

No. 2015-1146

IN THE
United States Court of Appeals
FOR THE FEDERAL CIRCUIT

THE TRUSTEES OF COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK,

Plaintiff-Appellant,

v.

SYMANTEC CORPORATION,

Defendant-Appellee.

APPEAL FROM THE UNITED STATES DISTRICT COURT
FOR THE EASTERN DISTRICT OF VIRGINIA
IN No. 3:13-cv-00808-JRS, SENIOR JUDGE JAMES R. SPENCER

**BRIEF OF PLAINTIFF-APPELLANT
THE TRUSTEES OF COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK**

DAVID I. GINDLER
JASON G. SHEASBY
RICHARD M. BIRNHOLZ
JOSEPH M. LIPNER
GAVIN SNYDER

IRELL & MANELLA LLP

1800 Avenue of the Stars, Suite 900
Los Angeles, California 90067
(310) 277-1010

Attorneys for Plaintiff-Appellant

January 20, 2015

CERTIFICATE OF INTEREST

Counsel for Plaintiff-Appellant The Trustees of Columbia University in the City of New York certifies the following:

1. The full name of every party or amicus represented by us is:

The Trustees of Columbia University in the City of New York.

2. The names of all real parties in interest (if the party named in the caption is not the real party in interest) represented by us are:

Not applicable.

3. All parent corporations and any publicly held companies that own 10% or more of the stock of the party or amicus curiae represented by us are:

None.

4. The names of all law firms and the partners or associates that have appeared for the party or amicus now represented by us in the trial court or agency or are expected to appear in this court are:

From Irell & Manella LLP: David I. Gindler, Jason G. Sheasby, Richard M. Birnholz, Joseph M. Lipner, Michael H. Strub, Jr., Gavin Snyder, C. Maclain Wells, Thomas C. Werner, Xinlin Li (no longer with firm), and Douglas A. Fretty (no longer with firm).

From Spotts Fain, P.C.: Dana D. McDaniel and John M. Erbach.

Dated: January 20, 2015

/s/ David I. Gindler

David I. Gindler

*Counsel for Plaintiff-Appellant
The Trustees of Columbia University
in the City of New York*

TABLE OF CONTENTS

PRELIMINARY STATEMENT	1
JURISDICTIONAL STATEMENT	4
STATEMENT OF THE ISSUES.....	5
STATEMENT OF THE CASE.....	6
I. The Parties	6
II. The Asserted Patents.....	7
A. Background	7
B. The '544 and '907 Patents	9
C. The '115 and '322 Patents	12
D. The '084 and '306 Patents	14
III. The District Court Proceedings	16
A. The District Court's Claim Construction.....	16
B. Columbia's Motion for Clarification	17
C. The Stipulated Final Judgment	18
SUMMARY OF ARGUMENT	20
ARGUMENT	24
I. Standard of Review.....	24
II. Claims Are Given Their Ordinary Meaning Absent Lexicography or Express Disavowal of Claim Scope.....	25
III. The District Court Erred in Construing the '544 and '907 Patents	29
A. The District Court Incorrectly Limited "Byte Sequence Feature" to an Exemplary Embodiment.....	29

1.	“Byte Sequence Features” Are Not Limited to “Machine Code Instructions”	31
2.	There Is No Lexicography or Disavowal of Claim Scope that Justifies Importing a “Machine Code Instructions” Limitation	35
3.	The Prosecution History Does Not Limit the Scope of Byte Sequence Features to “Machine Code Instructions”	39
B.	The District Court Erred in Holding Claims 1 and 16 of the ’544 Patent Invalid Under Section 112 ¶ 2.....	42
1.	The District Court’s Incorrect Byte Sequence Feature Construction Led to the Incorrect Invalidity Ruling.....	43
2.	The Claims Are Not Directed to Mutually Exclusive Embodiments	44
IV.	The District Court Erred in Construing the ’115 and ’322 Patents	47
A.	The District Court Incorrectly Relied on an Unrelated Patent Family in Construing “Anomalous”	48
B.	The Term “Anomalous” Does Not Include a Negative Limitation Preventing Analysis of Anything Other Than “Attack-Free” Data	49
1.	The Claims Confirm That the Model Need Not Be Generated Only With Attack-Free Data.....	50
2.	The Specification Does Not Require Models Built Only With Attack-Free Data	51
3.	The Prosecution History Describes Models Built Using Attack Data.....	53
V.	The District Court Erred in Construing the ’084 and ’306 Patents	55

A.	The District Court Incorrectly Narrowed the Claimed “Probabilistic Model of Normal Computer System Usage”	56
1.	The Claims Do Not Require a Model Generated With Only “Attack Free” Data	56
2.	The Specification Does Not Support the District Court’s Construction.....	57
3.	The Prosecution History Contains No Disclaimer and Does Not Support the District Court’s Construction.....	62
B.	The District Court Incorrectly Construed “Anomaly” and “Anomalous,” Which Should Have Received Their Plain Meaning	63
	CONCLUSION.....	64
	ADDENDUM	
	PROOF OF SERVICE	
	CERTIFICATE OF COMPLIANCE	

TABLE OF AUTHORITIES

Cases

Abbott Labs. v. Dey, L.P.,
287 F.3d 1097 (Fed. Cir. 2002) 48

AIA Eng’g Ltd. V. Magotteaux Int’l S/A,
657 F. 3d 1264 (Fed. Cir. 2011) 44

Alcon Research, Ltd. v. Apotex Inc.,
687 F.3d 1362 (Fed. Cir. 2012) 50

Allen Eng’g Corp. v. Bartell Indus., Inc.,
299 F.3d 1336 (Fed. Cir. 2002). 17, 43, 44, 45

Ancora Techs., Inc. v. Apple, Inc.,
744 F.3d 732 (Fed. Cir. 2014) 37, 47

Azure Networks, LLC v. CSR PLC,
771 F.3d 1336 (Fed. Cir. 2014) 26, 27, 38

Cordis Corp. v. Boston Sci. Corp.,
561 F.3d 1319 (Fed. Cir. 2009) 41

e.Digital Corp. v. Futurewei Techs., Inc.,
772 F.3d 723 (Fed. Cir. 2014) 48

Elkay Mfg. Co. v. Ebco Mfg. Co.,
192 F.3d 973 (Fed. Cir. 1999) 48

GE Lighting Solutions, LLC. v. Agilight, Inc.,
750 F.3d 1304 (Fed. Cir. 2014) 27, 38, 44, 51

Hill-Rom Servs., Inc. v. Stryker Corp.,
755 F.3d 1367 (Fed. Cir. 2014) 26, 27, 35, 41

Home Diagnostics, Inc. v. LifeScan, Inc.,
381 F.3d 1352 (Fed. Cir. 2004) 28

Innogenetics N.V. v. Abbott Labs.,
512 F.3d 1363 (Fed. Cir. 2008) 28

Liebel-Flarsheim Co. v. Medrad, Inc.,
358 F.3d 898 (Fed. Cir. 2004) 28

Micro Chem., Inc. v. Great Plains Chem. Co., Inc.,
194 F.3d 1250 (Fed. Cir. 1999) 63

Microsoft Corp. v. i4i Ltd. P’ship,
131 S. Ct. 2238 (2011)..... 25

Oatey Co. v. IPS Corp.,
514 F.3d 1271 (Fed. Cir. 2008) 25

Omega Eng’g v. Raytek Corp.,
334 F.3d 1314 (Fed. Cir. 2003) 57

Phillips v. AWH Corp.,
415 F.3d 1303 (Fed. Cir. 2005) passim

PSN Illinois, LLC v. Ivoclar Vivadent, Inc.,
525 F.3d 1159 (Fed. Cir. 2008) 34

Renishaw PLC v. Marposs Societa’ per Azioni,
158 F.3d 1243 (Fed. Cir. 1998) 31

Rexnord Corp. v. Laitram Corp.,
274 F.3d 1336 (Fed. Cir. 2001) 34

SciMed Life Sys., Inc. v. Advanced Cardiovascular Sys., Inc.,
242 F.3d 1337 (Fed. Cir. 2001) 27

Solomon v. Kimberly-Clark Corp.,
216 F.3d 1372 (Fed. Cir. 2000) 25, 44

Sun. Pharm. Indus., Ltd. v. Eli Lilly & Co.,
611 F.3d 1381 (Fed. Cir. 2010) 41

SunRace Roots Enters. Co., Ltd. v. SRAM Corp.,
336 F.3d 1298 (Fed. Cir. 2003) 38

Sun-Tiger, Inc. v. Scientific Research Funding Group,
189 F.3d 1327 (Fed. Cir. 1999) 57

<i>Talbert Fuel Sys. Patents Co. v. Unocal Corp.</i> , 275 F.3d 1371 (Fed. Cir. 2002)	44
<i>Teleflex, Inc. v. Ficosa N. Am. Corp.</i> , 299 F.3d 1313 (Fed. Cir. 2002)	57
<i>Teva Pharms. USA, Inc. v. Sandoz, Inc.</i> , 574 U.S. ___ (Jan. 20, 2015)	24
<i>Thorner v. Sony Computer Entm't Am. LLC</i> , 669 F.3d 1362 (Fed. Cir. 2012)	27
<i>Williamson v. Citrix Online LLC</i> , 770 F.3d 1371 (Fed. Cir. 2014)	38
<u>Statutes</u>	
28 U.S.C. § 1295(a)(1).....	4
28 U.S.C. § 1338(a)	4
28 U.S.C. § 2107(a)	4
35 U.S.C. § 112 ¶ 2	2, 44
35 U.S.C. § 282	25
<u>Rules</u>	
Fed. R. App. P. 4(a)	4
Fed. R. Civ. P. 54(b)	3, 4, 19

STATEMENT OF RELATED CASES

No other appeal in or from this same civil action was previously before this court or any other court of appeals. No other cases are pending between the same parties and there are no known or pending cases that will directly affect or be directly affected by this court's decision in this matter.

PRELIMINARY STATEMENT

This appeal concerns claim constructions that violate this Court's rules of claim interpretation. A construction may not import limitations restricting claims to particular embodiments or examples. Claims are presumed to have their full scope. This presumption is only overcome by a clear definition or words of manifest exclusion or restriction disavowing claim scope. Without finding any definition or disclaimer, the District Court strayed from these core principles and issued a two-page claim construction order that limits the claims to specific embodiments and examples from the specification.

This litigation involves six Columbia patents from three distinct families relating to important advances in computer security. The patents describe novel ways to use machine learning to detect previously unknown viruses and malicious computer intrusions, often called "zero day" attacks. Each patent family describes using models of program behavior to diagnose programs as malicious or benign. The patents explain how these models of program behavior may be created using different types of ingredients.

The first patent family, U.S. Patent Nos. 7,487,544 and 7,979,907, describes, among other things, extracting "byte sequence features" from executable email attachments to determine whether they are malicious. The District Court incorrectly construed the term "byte sequence feature" to be limited to only "a

representation of machine code instructions,” even though the specification expressly discloses byte sequence features that are not representations of machine code instructions. Using this faulty construction, the District Court compounded the error by holding claims 1 and 16 of the ’544 patent indefinite under section 112 ¶ 2 on the grounds that they cover mutually exclusive embodiments. This holding is incorrect—the claims track the express teaching of the specification.

The second family, U.S. Patent Nos. 8,074,115 and 8,601,322, teaches ways to determine if a program is “anomalous” by using machine learning to create models of function calls that programs make when they run. The meaning of “anomalous” in these patents is “behavior that deviates from normal and may correspond to an attack.” The District Court departed from this plain meaning in the intrinsic record, incorrectly reading the term “anomalous” to require a deviation from a “*model of typical, attack free computer system usage*” created with *only* “attack free” data. There is no justification for importing only “attack free” data or other restrictions into the claims. Indeed, rather than exclude the use of attack data in the model, the specification, the claims as filed, and the claims as issued all specify using attack data when creating the model. Moreover, in explaining this aspect of its construction, the District Court did not cite the ’115 and ’322 patents, but instead only referred to the unrelated third family in the case.

Applying a construction from one patent family to another is contrary to the well-settled principle that claims of unrelated patents are construed separately.

The third patent family at issue, U.S. Patent Nos. 7,448,084 and 7,913,306, describes how modeling activity on particular locations in a computer system – the Windows registry and file system – can improve detection of anomalous program behavior reflecting an intrusion. The District Court again improperly read in extraneous, negative limitations to the terms “anomaly/anomalous” and “probabilistic model of normal computer system usage,” requiring a model created using only “attack free” data. The claim language is not so limited and the specification teaches otherwise.

In order to redress promptly these incorrect claim constructions, Columbia stipulated to judgment of non-infringement under Rule 54(b). The District Court’s claim constructions are incorrect and should be reversed. The corresponding judgment should be vacated and the case remanded so it can proceed under proper claim constructions.

JURISDICTIONAL STATEMENT

The U.S. District Court for the Eastern District of Virginia had subject matter jurisdiction over this action pursuant to 28 U.S.C. § 1338(a).

On November 4, 2014, the District Court entered a Final Judgment Pursuant to Rule 54(b) of the Federal Rules of Civil Procedure pursuant to the parties' stipulation of non-infringement based on the District Court's claim constructions and ruling that certain claims of the '544 patent were invalid as indefinite. A1-8.

On November 10, 2014, Columbia timely filed its Notice of Appeal in accordance with 28 U.S.C. § 2107(a) and Rule 4(a) of the Federal Rules of Appellate Procedure. A314-16.

This Court has jurisdiction over this appeal pursuant to 28 U.S.C. § 1295(a)(1).

STATEMENT OF THE ISSUES

1. Whether the judgment of non-infringement of the '544 and '907 patents must be vacated because the District Court erred in limiting “byte sequence feature” to a “representation of machine code instructions,” when no lexicography or disavowal restricts claim scope.

2. Whether the judgment of indefiniteness of claims 1 and 16 of the '544 patent must be vacated because the District Court erred in concluding that the claim terms “byte sequence feature” and “byte string representative of resources” are directed to separate and mutually exclusive embodiments.

3. Whether the judgment of non-infringement of the '115 and '322 patents must be vacated because the District Court erred in construing “anomaly/anomalous” to require “deviation/deviating from a model” generated with only “typical, attack-free data” and, when doing so, relied on the disclosure of a different patent family.

4. Whether the judgment of non-infringement of the '084 and '306 patents must be vacated because the District Court erred in construing “probabilistic model of normal computer system usage” and “anomaly/anomalous” by imposing a negative limitation requiring a model generated with only “typical, attack-free data.”

STATEMENT OF THE CASE

I. THE PARTIES

Columbia is a leading university and research institution. The next generation computer security techniques at issue in this case were developed at Columbia's School of Engineering and Applied Sciences, which includes the Department of Computer Science—a department dedicated to training future leaders to solve important security challenges. A194-95. Symantec is based in Mountain View, California and sells popular computer security products and services, including “Norton” antivirus software. A195.

Columbia brought this action in December 2013 alleging that Symantec antivirus and computer security products and services infringed six Columbia patents from three separate families. A160-68; A173-190; A193-202; A209-28.¹ The asserted patent families are (1) U.S. Patent Nos. 7,487,544 (the '544 patent) and 7,979,907 (the '907 patent), (2) U.S. Patent Nos. 7,448,084 (the '084 patent) and 7,913,306 (the '306 patent), and (3) U.S. Patent Nos. 8,074,115 (the '115

¹ Columbia also alleged state law claims relating to Columbia intellectual property in “decoy” technology and to correct inventorship on a patent presently assigned to Symantec. A202-08; A228-36. These claims are separate from the Asserted Patents addressed in the judgment at issue on this appeal and have been stayed. A14.

patent) and 8,601,322 (the '322 patent) (collectively "Asserted Patents"). Each family has a distinct disclosure with distinct priority dates.

II. THE ASSERTED PATENTS

A. Background

Computer networks are continually becoming larger and more complex, and the amount of sensitive information exchanged and accessible on networked computers is increasing dramatically. As a result, there is a strong need for technologies preventing malicious viruses and other harmful intrusions to computers. The technology at issue concerns important advances meeting this need. '544 patent, A48 (1:30-2:67); '084 patent, A83-84 (1:41-3:10); '115 patent, A124 (1:20-44).

One challenge in computer security involves detecting attacks never previously seen by an intrusion detection system. These attacks sometimes are referred to as "zero-day" attacks because developers have "zero days" beforehand to patch the vulnerability. A382.

Professors Salvatore Stolfo and Angelos Keromytis and their students in Columbia's Intrusion Detection Lab and Network Security Lab pioneered next generation technologies based on, among other things, machine learning techniques providing vastly improved detection of new viruses and attacks. A196-

201. The Asserted Patents concern the application of machine learning techniques to various areas of computer security.

Machine learning is a subfield of artificial intelligence and statistics. A382-83. Machine learning systems are able to learn automatically from data. '544 patent, A48 (1:65-2:45); '084 patent, A87 (10:1-11). The first step involves collecting large amounts of data. '544 patent, A50 (5:16-27). Next, in a training phase, the collected data is fed into a machine learning algorithm to generate a model based on important parts of the data called features. '544 patent, A50 (5:28-6:6); '084 patent, A87-88 (10:1-11, 12:49-65); '115 patent, A125 (3:28-45). A model is the encapsulation of the automatically-generated insights that the machine learning algorithm derived from the training data. '544 patent, A49-50 (4:67-5:5). As additional training data is obtained, the model may be modified. A54-57 (14:4-22, cl. 5). The model is then deployed in the field. New programs not seen during the training phase are run through, or compared to, the model. '544 patent, A50 (5:6-15); '084 patent, A88 (12:36-48); '115 patent, A125 (3:46-56). The system can classify or evaluate the new program based on the comparison with the model. '544 patent, A51 (8:3-12).

Model creation using machine learning raises two questions: what the model represents and what ingredients are used to create the model. These are different concepts. For example, anomaly detection models can detect deviations

from normal behavior. The ingredients for such a model include attack-free data but also can include attack data. '115 patent, A133-34 (cls. 1, 8, 18, 29, 39); A674. By analogy, a model of a fresh apple could consider the attributes of a fresh apple in isolation but may be enriched by considering how fresh apples differ from rotten apples. A382-83; A398-402.

B. The '544 and '907 Patents

The '544 and '907 patents are entitled "System and Methods for Detection of New Malicious Executables." The '544 and '907 patents generally relate to "detecting malicious executable programs, and more particularly to the use of data mining techniques to detect such malicious executables in email attachments." '544 patent, A48 (1:33-37). The patents improved on prior art approaches relying on specific signatures of known viruses and were directed at a new technique for classifying executables as malicious or benign "which is not limited to particular types of files . . . and which provides the ability to detect new, previously unseen files." A48 (1:39-2:1, 2:64-67).

An executable fundamentally is a collection of bytes. A50 (6:29-35); A582-84 (describing sections of files in Windows Portable Executable (PE) format); A598 ("Initialized data for a section consists of simple blocks of bytes."). Both parties' experts agreed during claim construction that an executable may be organized into sections containing sequences of bytes representing different

functions. A1700-05; A1322-25. For example, an executable contains the program's instructions. Symantec's expert stated that these "machine code instructions" are byte sequences "that tell a computer's central processing unit what to do." A1321. Machine code instructions are not the only sequences of bytes in an executable. An executable also may contain information about "resources" that the program uses, such as dynamically linked libraries or "DLLs" the program may invoke, and may also contain "plain text strings" with other data. A50-51 (6:23-8:2); A389-92.² Resource information and plain text strings may be in a different section than the machine code instructions, such as in the program "header." A50-51 (6:48-58, 7:40-53). Although the machine code instructions, resource information, and plain text strings may reflect different functions and may be in different locations in the executable, they all are indisputably made up of sequences of bytes. A635-39 (example of byte sequences containing resource information); A594-97 (example of byte sequences that are plain text strings); A1700-05 (explaining how byte sequences may represent machine code instructions, resource information, and plain text strings).

² Dynamic link libraries (DLLs) generally are libraries of frequently used functions that an executable may call, for example, to interact with the operating system. A50-51 (6:35-7:39); A90-91. Plain text strings are sequences of printable characters that may appear in various places in an executable, such as the string "microsoft." A51 (7:40-8:2); A1704-1705.

The inventors of the '544 and '907 patents described a technique that could analyze the entirety, or selected parts, of an executable. The approach of the '544 and '907 patents involves extracting information from an executable for use in a machine learning system to classify the executable. The patents refer to the properties or attributes of the sequences of bytes in the executable that are analyzed as “byte sequence features.” The term “byte sequence feature” is used in the specification and claims to describe a broad category of information derived from sequences of bytes in the executable to be classified. That is, the data to be analyzed, like pieces of evidence in the dossier on a suspect, could be multiple types of information from the entire executable or just a part. The extraction of “byte sequence features” from executables is a concept in all claims of the '544 and '907 patents. A49 (3:23-24) (“A byte sequence feature is subsequently extracted from the executable attachment.”); *see, e.g.*, A57 (cl. 1). The byte sequence features are then used in a machine learning system to classify the executable. A49 (3:24-29); A51 (8:3-55).

Various examples of byte sequence features are described. The Summary describes how in one embodiment “[e]xtracting the byte sequence feature from the executable attachment may comprise converting the executable attachment from binary format to hexadecimal format.” A49 (3:34-37). In this embodiment, byte sequence features are extracted from the entire file and “each byte sequence in the

program is used as a feature.” A50 (6:21-22) (describing embodiment using utility known as “hexdump” to convert the entire file to hexadecimal). In other words, this embodiment includes byte sequence features representing machine code instructions but also includes byte sequence features representative of all other information in the executable.

The Summary also describes an embodiment in which “extracting the byte sequence features from the executable attachment may comprise creating a byte string representative of resources referenced by said executable attachment.” A49 (3:37-40). In other words, in this embodiment, the byte sequence features extracted must represent a specific type of information in the executable, the “resources” referenced by the executable, not all the information in the executable. These embodiments are described in greater detail in the “Detailed Description of Exemplary Embodiments,” which describes using as byte sequence features all the byte sequences in a file, sequences of machine code instructions, information generated by examining resource information in the executable, or information generated by examining plain text strings. A50-51 (5:57-8:2).

C. The '115 and '322 Patents

The '115 and '322 patents are entitled “Methods, Media and Systems for Detecting Anomalous Program Executions.” These patents recognize that certain activity – function calls made by running programs – can be a leading indicator of

intrusions or attacks. '115 patent, A125 (3:28-56). As the specification describes, “instrumenting, monitoring, and analyzing application-level program function calls and/or arguments . . . can be used to detect anomalous program executions that may be indicative of a malicious attack or program fault.” A125 (3:7-15). Other aspects of the claims not at issue on this appeal include the use of an “emulator” in analyzing the function calls and notifying an “application community” once malicious activity is detected. A126 (6:31-47); A133 (cl. 1).

The claims all require a “model of function calls.” A133 (cl. 1). The model of function calls is developed from running programs and inspecting what calls are made. A125 (3:46-56). This information is used to train a model, and once the model has been trained it can be applied to inspect programs in an emulator. *Id.*

The '115 and '322 patents make clear that both “normal” and “attack” data may be used to build the models. The independent claims of the '115 and '322 patents (both as originally filed and as issued) generally require a “model of function calls,” while the dependent claims (also as originally filed and as issued) require that specific types of data must be used in the model. Claim 7 of the '115 patent and claim 6 of the '322 patent recite that “the model reflects normal activity of the at least a part of the program.” A133; A158. Claim 8 of the '115 patent and claim 7 of the '322 patent recite that “the model reflects attacks against the at least a part of the program.” *Id.*

D. The '084 and '306 Patents

The '084 and '306 patents are entitled “System and Methods for Detecting Intrusions in a Computer System by Monitoring Operating System Registry Accesses.” Prior art designs of the inventors and other research groups involved the use of “anomaly detection algorithms [that] may build models of normal behavior in order to detect behavior that deviates from normal behavior and which may correspond to an attack.” '084 patent, A83 (2:34-37). These prior art anomaly detection systems were lacking because they looked at computer activity that was too irregular or required intensive computational overhead to monitor. A83-84 (2:65-3:7). The inventors realized that malicious software often tries to manipulate areas of the operating system known as the “registry” or “file system.” A85 (5:55-6:3); A91 (17:43-54). The inventors discovered that a model that included in the training data records of normal processes that access the registry or file system resulted in improved anomaly detection. A85 (5:61-6:15).

The “Background of the Invention” section describes different ways to detect intrusions. Some prior art systems looked for activity matching the signature of previous attacks. A83 (2:28-32). Because such systems only used data from known attacks, they were not effective in detecting new malicious programs not seen before. *Id.* Another approach involves the use of anomaly detection algorithms to “build models of normal behavior in order to detect

behavior that deviates from normal behavior and which may correspond to an attack.” A83 (2:34-37). By taking into account normal activity, a model can detect an anomaly even if the malicious program has never been encountered before. A83 (2:37-39); A85 (6:20-24) (“[A] program which substantially deviates from this normal activity may be easily detected as anomalous.”).

Anomaly detection models can be constructed with different types of ingredients. For example, the anomaly detection model may be generated with only data on normal processes free of attacks. A90 (15:38-46). Columbia researchers, however, explained in papers cited in the patent specification that more robust models of normal behavior could be constructed for detecting anomalies if the data used to build the model was supplemented with data regarding attacks or malicious programs. In other words, including data on attacks could enhance the ability to determine whether particular activity was anomalous. *See, e.g.*, A83 (2:39-64) (citing paper from named inventor describing “training over clean data (normal data containing no anomalies),” but noting “several inherent drawbacks to this approach” and describing “a technique for detecting anomalies without clean data,” which used normal data supplemented with data regarding abnormal processes); A674; A398-402.

The patents’ approach includes “gathering features from records of normal processes that access the operating system registry.” A93 (cl. 1). A “probabilistic

model of normal computer system usage” is generated based on the monitored data. A84 (3:21-30); A86 (8:7-21). Accesses are then analyzed to determine if they are anomalous and therefore indicative of attacks. A84 (3:21-30); A85 (5:2-10). While “records of normal processes” must be included in the data used to generate the model, the claims do not exclude the use of other ingredients or require *only* data reflecting “attack free” processes be used in the model.

III. THE DISTRICT COURT PROCEEDINGS

A. The District Court’s Claim Construction

After claim construction briefing, the District Court held a hearing on September 4, 2014. A32-34 (Dkts. 106, 107, 109, 110, 121). On October 7, 2014, the District Court issued a Claim Construction Order. A9-10.

With respect to the ’544 and ’907 patents, the District Court’s Claim Construction Order provides:

1. “Byte Sequence Feature”: Feature that is a representation of machine code instructions of the executable. “Feature” is a property or attribute of data which may take on a set of values.

...

3. “Wherein [the step of] extracting said byte sequence features from said executable attachment comprises creat[ing/e] a byte string representative of resources referenced by said executable attachment”: Indefinite. The Court finds that the “resource information” feature extraction embodiment is separate and distinct from the “byte sequence feature” extraction embodiment. Because Claims 1 and 16 of the ’544 patent conflate these terms, and thus are inconsistent with the specification, the Court holds that this term is indefinite. *See Allen*

Eng'g Corp. v. Bartell Indus., Inc., 299 F.3d 1336, 1349 (Fed. Cir. 2002).

A9.

With respect to the '115 and '322 patents, the Claim Construction Order provides:

1. "Anomalous": Deviation/deviating from a model of typical, attack-free computer system usage.

A10.

With respect to the '084 and '306 patents, the Court's Claim Construction Order provides:

2. "Probabilistic Model of Normal Computer System Usage": Model of typical attack-free computer system usage that employs probability. "Probability" is the likelihood that an event will occur or a condition will be present.

"Normal Computer System Usage": Typical attack-free computer system usage.

3. "Anomaly" / "Anomalous": Deviation/deviating from a model of typical, attack-free computer system usage.

Id.

B. Columbia's Motion for Clarification

On October 10, 2014, Columbia moved for clarification of the Claim Construction Order. A2199-2201; A2204-08. The constructions for both the '115 and '322 family and the '084 and '306 family required a model of "typical, attack free computer system usage." But the constructions left unresolved "a dispute

between the parties as to what ingredients can be used to create the model.”
A2204.

On October 23, 2014, the District Court issued a Memorandum Order granting Columbia’s Motion for Clarification of Claim Construction Order and clarifying certain constructions. A11-12. The District Court quoted from claim 1 of the ’084 patent, then wrote:

Logically, if the anomaly detection systems detect deviations from normal activity, that normal activity *must be* “attack-free” activity. Applying this logic to the rest of claim 1, which gathers “features from records of normal processes” and then generates “a probabilistic model of normal computer system usage based on [those] features,” it follows that the model is generated with *only* attack-free data.

A12 (emphasis added, other emphasis removed).

C. The Stipulated Final Judgment

On November 3, 2014, the parties filed a stipulation and joint motion for entry of final judgment of (a) non-infringement of all Asserted Patents and (b) invalidity of claims 1 and 16 of the ’544 patent based on indefiniteness. A272-82; A283-93; A304-13. The parties stipulated that, under the Court’s construction, the accused Symantec products did not practice the “byte sequence feature” limitation of the ’544 and ’907 patents, the “anomalous” limitation of the ’115 and ’322 patents, or the “probabilistic model of normal computer system usage” and “anomaly/anomalous” limitations of the ’084 and ’306 patents. A275-278.

Columbia noted its disagreement with the District Court's constructions and reserved the right to appeal. *Id.*

On November 4, 2014, the District Court entered final judgment on Columbia's patent infringement claims pursuant to Rule 54(b) of the Federal Rules of Civil Procedure. A13-14; A1-8.

Columbia timely filed a Notice of Appeal. A314-16.

SUMMARY OF ARGUMENT

The District Court improperly read into the claims limitations from specific embodiments and examples in the specification.

The '544 and '907 patents recite a method including the steps of “extracting a byte sequence feature” from an executable email attachment, “wherein” the extracting “comprises creating a byte string representative of resources referenced” by the executable. The District Court incorrectly construed “byte sequence feature” as being limited to only a particular kind of sequence of bytes—those representing “machine code instructions,” which are only a part of the content of an executable. The District Court’s construction is contrary to the Summary and the remainder of the specification, which teach that byte sequence features may be extracted from any portion of the executable. Examples of byte sequence features in the Summary and the specification are not limited to only machine code instructions, but include resource information, plain text strings, or instructions in the file. '544 patent, A49-51 (3:24-40, 6:23-8:2). The specification and the prosecution history contain no clear definition or disclaimer justifying the District Court’s narrow construction. The plain meaning of byte sequence feature is a property or attribute of a sequence of bytes which may take on a set of values. This is the construction that should have been adopted.

The District Court also erred in holding claims 1 and 16 of the '544 patent indefinite under section 112 ¶ 2. The court ruled that the “byte sequence feature” extraction embodiment is “separate and distinct” from the “resource information” extraction embodiment and that therefore the claims improperly covered two different embodiments. The court’s ruling ignores the logic and language of the claim, is contrary to the specification, and depends entirely on the court’s incorrect claim construction of “byte sequence feature” discussed above. The plain reading of claims 1 and 16 is that the “resource” information is a subset of the more general byte sequence feature, not a distinct and mutually exclusive embodiment. This is confirmed in the specification, which states expressly that “extracting the byte sequence features from the executable attachment *may comprise* creating a byte string representative of resources referenced by said executable attachment.” A49 (3:37-40). The sequence of bytes representing resource information is thus a specific example of a byte sequence feature. Claims 1 and 16 are directed to this embodiment, requiring the extraction of byte sequence features “wherein” the extracting the byte sequence feature “comprises creating a byte string representative of resources referenced by said executable.”

The '115 and '322 patents are directed to “detecting anomalous program executions” using a “model of function calls.” The District Court erred in construing the term “anomalous” in the '115 and '322 patents. “Anomalous”

means “behavior that deviates from normal and may correspond to an attack.” The District Court departed from this plain meaning and read in negative limitations *requiring* a deviation “from a model of typical, attack free computer system usage” in which the model must be created using *only* “attack free” data. Nothing in these patents justifies these requirements. The independent claims set out what must be present, such as a “model of function calls,” but impose no further constraints on the claimed model and do not exclude the use of data reflecting attacks. Indeed, the District Court disregarded dependent claim 8, which depends from the claim the court interpreted and expressly requires that the model “reflects attacks against” the program, just what the court said could not be reflected. ’115 patent, A133 (cl. 8). Moreover, the only citations in the court’s order were to the unrelated third family in the case. Construing one patent family solely by reference to another family is contrary to the well-settled principle that claims of unrelated patents must be construed separately.

The ’084 and ’306 patents recite detecting intrusions in the operation of a computer system by “gathering features from records of normal processes that access the operating registry” (or “file system” for the ’306 patent) and analyzing the features using a “probabilistic model of normal computer system usage” to determine if a registry or file system access “is an anomaly.” The District Court also wrongly read negative limitations into the ’084 and ’306 patent claims. One

ingredient that must be included in the claimed “probabilistic model of normal computer system usage” is “records of normal processes that access the operating system registry” or the “file system.” This model is used “to detect deviations from normal computer system usage to determine whether the access to the operating system registry [or file system] is an anomaly.” ’084 patent, A93 (cl. 1); ’306 patent, A110 (cl. 1). The District Court improperly added a negative limitation requiring a deviation from a “model of typical, attack-free computer system usage” created using “only attack-free data.” A12. Nothing in the plain language of the claims limits the data that may be used to create the claimed model, and no principle of patent law excludes the use of additional ingredients beyond normal access to the registry or file system.

The stipulated judgment of no infringement and indefiniteness is based on incorrect claim constructions. The District Court’s claim constructions should be reversed, the judgment vacated, and the case remanded.

ARGUMENT

I. STANDARD OF REVIEW

The Supreme Court today in *Teva Pharms. USA, Inc. v. Sandoz, Inc.*, 574 U.S. __ (Jan. 20, 2015) clarified the standard of review of claim construction. The Supreme Court confirmed that the District Court’s ultimate interpretation of patent claims remains a legal conclusion reviewed de novo. *Teva*, slip op. at 9 (“[T]he Federal Circuit will continue to review *de novo* the district court’s ultimate interpretation of the patent claims.”). As the Court explained, “when the district court reviews only evidence intrinsic to the patent (the patent claims and specifications, along with the patent’s prosecution history), the judge’s determination will amount solely to a determination of law, and the Court of Appeals will review that construction de novo.” *Id.*, slip op. at 11-12. In the event the District Court makes “subsidiary factual findings” about extrinsic evidence, “the Federal Circuit must apply clear error review when reviewing subsidiary factfinding in patent claim construction.” *Id.*

The District Court’s claim constructions here are reviewed de novo. The District Court did not make any factual findings in its Claim Construction Order and the Memorandum Order clarifying the claim construction. These orders recite the District Court’s claim constructions as a conclusion of law. When the District Court did provide a narrative explanation, the court relied entirely on the intrinsic

evidence. Even if any of the District Court's comments were treated as "subsidiary factual findings" under *Teva* (and they are not), such comments would constitute clear error because they directly contradict the intrinsic record.

The District Court entered a stipulated final judgment of non-infringement that was based solely on the court's claim constructions. If the District Court's claim constructions are incorrect, the judgment must be vacated. *Oatey Co. v. IPS Corp.*, 514 F.3d 1271, 1277-78 (Fed. Cir. 2008).

The District Court's ruling that claims 1 and 16 of the '544 patent are "indefinite" as "inconsistent with the specification" also is reviewed de novo. *Solomon v. Kimberly-Clark Corp.*, 216 F.3d 1372, 1377 (Fed. Cir. 2000) ("[A]s with claim construction, a determination under either portion of section 112, paragraph 2, is a question of law that we review de novo."). When the defendant challenges validity of a claim under this portion of section 112, the defendant must meet the heavy burden of 35 U.S.C. § 282. *Microsoft Corp. v. i4i Ltd. P'ship*, 131 S. Ct. 2238, 2242 (2011) (invalidity defenses must be proved by clear and convincing evidence).

II. CLAIMS ARE GIVEN THEIR ORDINARY MEANING ABSENT LEXICOGRAPHY OR EXPRESS DISAVOWAL OF CLAIM SCOPE

"It is a bedrock principle of patent law that the claims of a patent define the invention to which the patentee is entitled the right to exclude." *Phillips v. AWH Corp.*, 415 F.3d 1303, 1312 (Fed. Cir. 2005) (en banc) (quotations omitted). The

standards for construing claims are well-established. “Claim terms are generally given their plain and ordinary meanings to one of skill in the art when read in the context of the specification and prosecution history.” *Hill-Rom Servs., Inc. v. Stryker Corp.*, 755 F.3d 1367, 1371 (Fed. Cir. 2014) (reversing stipulated judgment of non-infringement because of erroneous claim construction). Indeed, there is a “heavy presumption that claim terms carry their accustomed meaning in the relevant community at the relevant time.” *Azure Networks, LLC v. CSR PLC*, 771 F.3d 1336, 1347 (Fed. Cir. 2014) (adopting patentee’s construction and vacating stipulated judgment of non-infringement) (quotations omitted).

This presumption can be overcome in only two circumstances: the patentee has expressly defined a term or has expressly disavowed the full scope of the claim in the specification or the prosecution history. *See, e.g., Phillips*, 415 F.3d at 1316 (“[T]he specification may reveal a special definition given to a claim term” or “an intentional disclaimer, or disavowal, of claim scope by the inventor”); *Azure Networks*, 771 F.3d at 1348 (“Departure from the ordinary and customary meaning is permissible only when the patentee has acted as his own lexicographer or disavowed claim scope in the specification or during the prosecution history.”); *Hill-Rom*, 755 F.3d at 1371 (“We depart from the plain and ordinary meaning of claim terms based on the specification in only two instances: lexicography and disavowal.”); *Thorner v. Sony Computer Entm’t Am. LLC*, 669 F.3d 1362, 1365

(Fed. Cir. 2012) (“There are only two exceptions to this general rule: 1) when a patentee sets out a definition and acts as his own lexicographer, or 2) when the patentee disavows the full scope of the claim term either in the specification or during prosecution.”).

The standards for finding lexicography or disavowal “are exacting.” *Thorner*, 669 F.3d at 1366; *Hill-Rom*, 755 F.3d at 1371 (same); *GE Lighting Solutions, LLC. v. Agilight, Inc.*, 750 F.3d 1304, 1309 (Fed. Cir. 2014) (same). For a patentee to give a term something other than its well-established meaning, the patentee must “clearly set forth a definition of the disputed claim term” and “clearly express an intent to redefine the term.” *Thorner*, 669 F.3d at 1365 (quotations omitted); *GE Lighting*, 750 F.3d at 1309 (same); *Azure Networks*, 771 F.3d at 1349 (same); *Hill-Rom*, 755 F.3d at 1371 (same). “The lexicography must appear with ‘reasonable clarity, deliberateness, and precision sufficient to narrow the definition of the claim term in the manner urged.’” *Azure Networks*, 771 F.3d at 1349 (quotation omitted). “Disavowal requires that the specification [or prosecution history] make[] clear that the invention does not include a particular feature, or is clearly limited to a particular form of the invention.” *Hill-Rom*, 755 F.3d at 1372 (brackets original; internal quotations and citations omitted); *SciMed Life Sys., Inc. v. Advanced Cardiovascular Sys., Inc.*, 242 F.3d 1337, 1341 (Fed. Cir. 2001) (same).

Phillips explains why courts are admonished against importing limitations into claims: “if we once begin to include elements not mentioned in the claim, in order to limit such claim . . . we should never know where to stop.” *Phillips*, 415 F.3d at 1312 (internal quotation and citation omitted). Courts therefore should not “at any time import limitations from the specification into the claims.” *Innogenetics N.V. v. Abbott Labs.*, 512 F.3d 1363, 1370 (Fed. Cir. 2008) (quotation omitted). This is the case even if a patent discloses only one embodiment. *Phillips*, 415 F.3d at 1323 (“[W]e have expressly rejected the contention that if a patent describes only a single embodiment, the claims of the patent must be construed as being limited to that embodiment.”); *Liebel-Flarsheim Co. v. Medrad, Inc.*, 358 F.3d 898, 906 (Fed. Cir. 2004) (“Even when the specification describes only a single embodiment, the claims of the patent will not be read restrictively unless the patentee has demonstrated a clear intention to limit the claim scope using ‘words or expressions of manifest exclusion or restriction.’”). “Absent a clear disavowal or contrary definition in the specification . . . the patentee is entitled to the full scope of its claim language.” *Home Diagnostics, Inc. v. LifeScan, Inc.*, 381 F.3d 1352, 1358 (Fed. Cir. 2004).

For each of the terms at issue on this appeal, the District Court improperly restricted the claims to a particular embodiment and imposed limitations that do not square with the claim language and intrinsic record.

III. THE DISTRICT COURT ERRED IN CONSTRUING THE '544 AND '907 PATENTS

The District Court erred in two ways: (1) it limited the term “byte sequence feature” to a “representation of machine code instructions”; and as a result (2) it invalidated claims 1 and 16 of the '544 patent as indefinite. In both instances, the District Court reached an incorrect result that is not faithful to the claim language or the specification's teachings.

A. The District Court Incorrectly Limited “Byte Sequence Feature” to an Exemplary Embodiment

The '544 and '907 patents disclose systems and methods for detecting malicious executables, such as harmful programs attached to emails. All claims and disclosed embodiments center around the extraction of “byte sequence features” from the executable to determine maliciousness. The byte sequence features are like a dossier on the executable. Just as a dossier on a person can contain different types of information about that person, byte sequence features can contain different types of information about the executable. In the specification, “byte sequence features” may represent the machine code instructions, resource information, and text strings in an executable. '544 patent, A49-50 (3:30-40, 5:57-8:2). The term “byte sequence feature” in the intrinsic record is an umbrella term for the properties or attributes of sequences of bytes that are extracted from any part of an executable.

“Byte sequence feature” is recited in all the claims of the ’544 and ’907 patents, including the claims as originally filed. A3874-81. For example, claim 1 reads:

1. A method for classifying an executable attachment in an email received at an email processing application of a computer system comprising:

- a) filtering said executable attachment from said email;
- b) extracting a byte sequence feature from said executable attachment; and
- c) classifying said executable attachment by comparing said byte sequence feature of said executable attachment with a classification rule set derived from byte sequence features of a set of executables having a predetermined class in a set of classes to determine the probability whether said executable attachment is malicious, wherein extracting said byte sequence features from said executable attachment comprises creating a byte string representative of resources referenced by said executable attachment.

A57 (cl. 1).

The District Court should have adopted Columbia’s proposed construction: a “property or attribute of a sequence of bytes, which may take on a set of values.” A347-50. Instead, the District Court construed “byte sequence feature” in two parts. The District Court first construed “feature” as “a property or attribute of data which may take on a set of values.” A9. The specification is consistent with this aspect of the construction, stating that “a feature is a property or attribute of data (such as ‘byte sequence feature’) which may take on a set of values.” A50

(5:63-64). But the District Court incorrectly limited “byte sequence feature” to a “[f]eature that is a representation of *machine code instructions* of the executable.”

A9. There is no lexicography or disavowal that would redefine “byte sequence feature” or limit the term to “a representation of machine code instructions,” which is only one part of an executable.

1. “Byte Sequence Features” Are Not Limited to “Machine Code Instructions”

The starting point for any claim construction is of course the claim language. *Phillips*, 415 F.3d at 1312 (“claims of a patent define the invention”); *Renishaw PLC v. Marposs Societa’ per Azioni*, 158 F.3d 1243, 1248 (Fed. Cir. 1998) (“claim construction inquiry, therefore begins and ends in all cases with the actual words of the claim”). The claim language “byte sequence feature” makes no reference to machine code instructions.

The phrase “byte sequence” is composed of familiar words. A “byte” is a collection of bits (commonly eight) and is a basic unit for representing information in computers. A568. A “byte sequence” is exactly that: a sequence of bytes. The intrinsic record does not artificially limit these concepts.

Throughout the specification, the inventors emphasize that byte sequence features may represent any portion of the executable, not just machine code instructions. The Summary states that one object of the invention is “a data mining technique which examines the entire file, rather than a portion of the file, such as a

header, to classify the executable as malicious or benign.” A49 (3:7-10). The Summary then explains in non-limiting terms how “extracting the byte sequence feature” can be accomplished in various ways. It states that “[e]xtracting the byte sequence feature . . . may comprise converting the executable attachment from binary format to hexadecimal format” (which creates byte sequence features from all information in the executable, not just the machine code instructions). A49 (3:30-37); A1700-04. The Summary then states how “[a]ccording to another embodiment, extracting the byte sequence features from the executable attachment may comprise creating a byte string representative of resources referenced by said executable attachment.” A49 (3:37-40). As discussed above, resources referenced by an executable (such as lists of DLLs that the program calls) are often in the program header, not in sections containing machine code instructions. A50 (6:30-58); A1702-03. In other words, the specification describes examples that can be used to generate byte sequence features that are not limited to machine code instructions, including byte strings representative of resources. A49-51 (3:30-40, 5:57-8:2).

The “Detailed Description of Exemplary Embodiments” describes several embodiments in which byte sequence features may be extracted from any part of an executable and not only from the machine code instructions. In one “exemplary embodiment,” byte sequence features are extracted from the executable using a

software utility called “hexdump.” Hexdump transforms the *entire* binary file into a hexadecimal file. A50 (6:7-12) (“Hexdump . . . is an open source tool that transforms binary files into hexadecimal files.”). In this hexdump embodiment, “each byte sequence in the program is used as a feature.” A50 (6:21-22). In other words, because hexdump acts on the entire executable, the byte sequence features that can be created in this embodiment are not just representations of the machine code instructions executed by the CPU, but any part of the program. A391; A1700-04.

The specification also describes how byte sequence features may be extracted using approaches other than hexdump. The Detailed Description of Exemplary Embodiments section makes clear that “[m]any additional methods of feature extraction are also useful.” A50 (6:23-24). For example, tools may be used to extract “resource information from the binary that provides insight to its behavior.” A50 (6:27-28). This may include extracting sequences of bytes from the program header “in object format,” such as information about the “dynamically linked libraries” (or “DLLs”) that the program calls to accomplish various functions. A50 (6:35-53). “From the object format, it is possible to extract a set of features to compose a feature vector, or string, for each binary representative of resources referenced by the binary.” A50 (6:56-58). Another approach is to use tools to extract “strings” of “plain text” from headers of executables. A51 (7:40-

53). As noted, the parties agree that resources and plain text include information that is not machine code instructions. A1700-05; A1322-25. All these approaches are examples of extracting byte sequence features from an executable that are not restricted to representations of machine code instructions.

The Summary never uses the phrase “machine code instructions,” nor does it define or limit byte sequence features to that particular portion of the executable. It expressly teaches that the term is not so limited. The District Court erred in disregarding this disclosure in the Summary and instead reading in limitations from one particular embodiment. *See PSN Illinois, LLC v. Ivoclar Vivadent, Inc.*, 525 F.3d 1159, 1166 (Fed. Cir. 2008) (“[W]e find that the District Court was incorrect in holding that the description of a preferred embodiment had more bite than the description in the summary of the invention”); *Rexnord Corp. v. Laitram Corp.*, 274 F.3d 1336, 1345-48 (Fed. Cir. 2001) (error to import characteristic from preferred embodiment, particularly when summary contained broader disclosure).

Given that the specification provides that “a feature is a property or attribute of data . . . which may take on a set of values,” A50 (5:57-6:6), the proper meaning of “byte sequence feature” based on the intrinsic record is “a property or attribute of a sequence of bytes which may take on a set of values.”

2. There Is No Lexicography or Disavowal of Claim Scope that Justifies Importing a “Machine Code Instructions” Limitation

In narrowing byte sequence features to “a representation of machine code instructions,” the District Court interpreted the intrinsic record such that “byte sequence features” applied only to a part of the hexdump embodiment and not to sequences of bytes representing resource information or strings in the file. Limiting the claims to a characteristic of an exemplary embodiment violates the express directive from *Phillips* that claim construction should not be used to limit claim scope to one particular embodiment, even if the specification discloses only one embodiment. *Phillips*, 415 F.3d at 1323. The *Hill-Rom* case is illustrative. In *Hill-Rom*, the patent claimed a “datalink” for conveying data. 755 F.3d at 1371-72. The specification disclosed a single embodiment, which used a cable to convey data, appeared to use the words “cable” and “datalink” interchangeably in describing the embodiment, and disclosed no alternative embodiment that used a wireless datalink. *See id.* at 1373. In reversing and setting aside the stipulated judgment of no infringement, this court declined to construe the term as requiring a wired connection, finding “no words of manifest exclusion or restriction” and “nothing in the language of the specification suggest[ing] that datalink should be limited to the cable used in the preferred embodiment.” *Id.* at 1372-73.

At claim construction, Symantec primarily relied on one particular paragraph in the “Detailed Description of Exemplary Embodiments” stating:

In the exemplary embodiment, hexdump was used in the feature extraction step. Hexdump, as is known in the art (Peter Miller, “Hexdump,” on line publication 2000, <http://gd.tuwien.ac.at/softeng/Aegis/hexdump.html> which is incorporated by reference in its entirety herein), is an open source tool that transforms binary files into hexadecimal files. ***The byte sequence feature is informative because it represents the machine code in an executable.*** After the “hexdumps” are created, features are produced in the form illustrated in FIG. 2 in which each line represents a short sequence of machine code instructions. In the analysis, a guiding assumption is made that similar instructions were present in malicious executables that differentiated them from benign programs, and the class of benign programs had similar byte code that differentiated them from the malicious executables. Each byte sequence in the program is used as a feature.

A50 (6:7-22) (emphasis added); *see also* A54 (13:24-26) (“This byte sequence is useful because it represents the machine code in an executable.”). Symantec isolated the phrase “represents the machine code in an executable” from the passage and then changed the language to craft the “machine code instructions” construction the District Court adopted.

The discussion of the hexdump embodiment is not a “clear definition” or “expression of manifest exclusion or restriction” needed to meet the “exacting” standards for finding a definition or disavowal. The language on which Symantec relied is itself non-limiting, beginning with “[i]n the exemplary embodiment.”

A50 (6:7). The patent also states expressly that the “Detailed Description of

Exemplary Embodiments” describes “illustrative embodiments” and how the accompanying “Brief Description of the Drawings” is not intended to limit or modify the claims scope. A49 (4:16, 4:46-50); A57 (19:6-9). The Summary similarly uses the words “may comprise” in describing examples of byte sequence features, not words of restriction or exclusion. A49 (3:34-40).

The sentence on which Symantec relied also lacks the clarity and precision that this Court looks for to justify limiting a claim. The passage never defines byte sequence features or says that it is “informative” because it represents *only* “machine code instructions.” The sentence also only uses the words “machine code instructions” when referring to Figure 2, which are part of the “figures showing illustrative embodiments.” A49 (4:15-16). “Under our claim construction law, a clear ordinary meaning is not properly overcome (and a relevant reader would not reasonably think it overcome) by a few passing references that do not amount to a redefinition or a disclaimer.” *Ancora Techs., Inc. v. Apple, Inc.*, 744 F.3d 732, 735 (Fed. Cir. 2014) (reversing construction when “nothing in the specification clearly narrows the term” “program”).

There is no language anywhere that the claimed byte sequence feature “is” or “must be” limited to “machine code instructions.” Rather, the specification makes plain that the byte sequence features are *not* limited to only the instructions portion of the file in the hexdump embodiment or any other. As set out in column

6, “[e]ach byte sequence in the program is used as a feature.” A50 (6:21-22). Again in column 13, the specification teaches that in the hexdump embodiment the entire executable is analyzed, not just machine code instructions: “In addition, this [hexdump] approach *involves analyzing the entire binary*, rather than portions such as headers, an approach which consequently provides a great deal of information about the executable.” A54 (13:24-29) (emphasis added).

When a District Court’s construction limits claims to a particular embodiment or example in the specification without a clear definition or unmistakable restriction in scope, this Court reverses. *See, e.g., Williamson v. Citrix Online LLC*, 770 F.3d 1371, 1377 (Fed. Cir. 2014) (“This court has repeatedly cautioned against limiting the claimed invention to preferred embodiments or specific examples in the specification.”) (internal quotations omitted); *GE Lighting Solutions*, 750 F.3d at 1310 (reversing stipulated judgment because “it was error to import the structural limitations of the preferred embodiment” into the claims); *Azure Networks*, 771 F.3d at 1350 (vacating stipulated judgment when “statements in the specification relied upon by the District Court neither define [the term] nor exclude” other examples); *SunRace Roots Enters. Co., Ltd. v. SRAM Corp.*, 336 F.3d 1298, 1302-06 (Fed. Cir. 2003) (statements descriptive of preferred embodiment did not constitute definition or disavowal and would not limit claims). Here, the lack of limiting language and the

disclosure of multiple embodiments not involving machine code instructions overcomes any possibility of disclaimer based on the exemplary embodiment.

3. The Prosecution History Does Not Limit the Scope of Byte Sequence Features to “Machine Code Instructions”

The prosecution history provides no support for limiting byte sequence features to machine code instructions. To the contrary, during prosecution, the patentee and the examiner understood the term to encompass sequences of bytes in a file beyond machine code instructions, further demonstrating that the District Court erred in its construction.

First, the patentee consistently used byte sequence feature as having its natural meaning not limited to any one embodiment in the specification. All the originally filed independent claims were directed to the extraction of byte sequence features from executables. A3874-81. Dependent claims in the originally-filed claims were directed at more particular characteristics of the byte sequence features. For example, one proposed dependent claim covered byte sequence features that included byte strings representative of resources, reciting “wherein the step of extracting said byte sequence features from said executable attachment comprises creating a byte string representative of resources referenced by said executable attachment.” A3874. Other proposed dependent claims were directed at byte sequence features representing the output of hexdump, reciting “wherein the step of extracting said byte sequence feature from said executable attachment

comprises converting said executable attachment from binary format to hexadecimal format.” *Id.* None of the original claims suggest that byte sequence features are limited to only the sequences of bytes representing the machine code instructions in the file.

The examiner also gave byte sequence feature its natural meaning not limited to a particular embodiment in the specification. In several office actions, the examiner consistently viewed the properties or attributes of any sequences of bytes in a program to be a byte sequence feature. *See* A4614 (citing a Morikawa reference about extracting “file attributes information” from the file header or plain text in the file); A4665-66 (citing a Kephart reference regarding different types of “data strings,” including portions of computer programs, metadata about the programs, and strings representing text); U.S. Patent No. 6,016,546 to Kephart et al. at 2:9-42; A4575-76 (citing a Chen reference concerning virus signatures; “It is obvious that this technique is done by extracting byte sequences from the content of e-mail attachments to compare with known virus signature, because a signature is a sequence of byte.”). The patentee’s response to this and several other office actions did not turn on the scope of byte sequence feature. If neither the patentee nor the examiner believed that byte sequence features were limited to machine code instructions, the prosecution history certainly does not “make[] clear that the invention does not include a particular feature, or is clearly limited to a particular

form of the invention,” as is required to show disavowal. *See Hill-Rom*, 755 F.3d at 1372.

To support its narrow construction in the District Court, Symantec took out of context a snippet from the early research paper constituting the provisional application. Symantec quoted from a section entitled “Byte Sequences Using Hexdump.” A3586. This described experiments involving outputting the entire file with the hexdump utility and stated “[t]he byte sequence feature is the most informative because it represents the machine code in an executable instead of resource information like libBFD features.” *Id.* When read in context, the provisional makes clear that the purpose of the hexdump embodiment is to capture information on the entire program, not just resource information or machine code instructions. *Id.* (“***analyzing the entire binary*** gives more information for non-PE format executables than the strings method”) (emphasis added). In any event, this passage, which does not even appear in the final specification, is far too thin a reed to justify a narrowing construction. The language and teaching in the final specification controls for claim construction. *Sun. Pharm. Indus., Ltd. v. Eli Lilly & Co.*, 611 F.3d 1381, 1388 (Fed. Cir. 2010) (“[T]he relevant specification for claim construction purposes is that of the issued patent, not an early version of the specification that may have been substantially altered”); *Cordis Corp. v. Boston Sci. Corp.*, 561 F.3d 1319, 1328-29 (Fed. Cir. 2009) (finding no disclaimer

where new description and figures introduced in the non-provisional application supported the broader construction).

B. The District Court Erred in Holding Claims 1 and 16 of the '544 Patent Invalid Under Section 112 ¶ 2

The District Court's incorrect construction of "byte sequence feature" led to an improper ruling that claims 1 and 16 of the '544 patent covered two different embodiments and were therefore indefinite under section 112 ¶ 2. Claims 1 and 16 each require the extraction of byte sequence features but also contain a "wherein" clause requiring a certain kind of byte sequence feature, namely one including a "byte string representative of resources" referenced by the executable. A57 (19:23-26) ("wherein extracting said byte sequence features from said executable attachment comprises creating a byte string representative of resources referenced by said executable attachment").

The natural reading of the claims is the correct one: a "byte string representative of resources" is a specific and more limited example of a "byte sequence feature." But because the District Court improperly narrowed "byte sequence features" to encompass only machine code instructions, the District Court consequently held that extracting "resource information" was not an example of a byte sequence feature as recited in the claims but a "separate and distinct" embodiment. The court then held that the claims were "inconsistent with the specification" and indefinite. A9 (citing *Allen Eng'g Corp. v. Bartell Indus., Inc.*,

299 F.3d 1336, 1349 (Fed. Cir. 2002)). The court’s ruling ignores the claim’s logic and language and the specification’s teaching and depends completely on the court’s incorrect claim construction of “byte sequence feature” as limited to machine code instructions.

1. The District Court’s Incorrect Byte Sequence Feature Construction Led to the Incorrect Invalidity Ruling

The District Court’s ruling that claims 1 and 16 were directed to separate and distinct embodiments stemmed entirely from the Court’s limitation of byte sequence features to machine code instructions. In other words, because the District Court viewed “byte sequence features” as limited to “machine code instructions,” the District Court incorrectly concluded that byte sequence features could not include information about “resources referenced by the executable.”

“Byte sequence feature” is used to refer generally to the properties or attributes of a sequence of bytes in the executable. As set forth in the Summary and in the originally filed claims, a byte sequence feature may be derived from byte strings representative of resources or other portions of the executable. A49 (3:1-4:9); A3874-81. Claims 1 and 16 cover this type of byte sequence feature “representative of resources.” A57 (cls. 1, 16). There is no inconsistency or indefiniteness in the claim.

The District Court should have taken the claim language at face value. As this Court has explained, “where claims can reasonably [be] interpreted to include

a specific embodiment, it is incorrect to construe the claims to exclude that embodiment, absent probative evidence on the contrary.” *GE Lighting*, 750 F.3d at 1311. Similarly, “[a] construction that renders the claimed invention inoperable should be viewed with extreme skepticism.” *AIA Eng’g Ltd. V. Magotteaux Int’l S/A*, 657 F. 3d 1264, 1278 (Fed. Cir. 2011) (quoting *Talbert Fuel Sys. Patents Co. v. Unocal Corp.*, 275 F.3d 1371, 1376 (Fed. Cir. 2002), *vacated and remanded on other grounds*, 537 U.S. 802 (2002)). A reversal of the construction of byte sequence feature requires reversal of the invalidity ruling.

2. The Claims Are Not Directed to Mutually Exclusive Embodiments

The only case cited in the District Court’s order to support the indefiniteness ruling is *Allen Engineering*. In *Allen*, the court invalidated claims under 35 U.S.C. § 112 ¶ 2, which requires that claims reflect what the inventor “regards as the invention”—a requirement separate from indefiniteness. 299 F.3d at 1348 (citing *Solomon v. Kimberly-Clark Corp.*, 216 F.3d 1372, 1377 (Fed. Cir. 2000)). The District Court erred in relying on *Allen Engineering*, which involves a different situation. The specification at issue there described a gearbox that “*cannot* pivot in a plane perpendicular to the biaxial plane,” while the claims recited a gearbox that pivots “*only* in a plane perpendicular to said biaxial plane.” *Id.* at 1349 (emphasis added). Based on this clear contradiction, the court found it “apparent from a simple comparison of the claims with the specification” that the inventor

did not regard the claimed gearbox to be his invention. *Id.* The patentee agreed, instead arguing that the claim suffered from a drafting error and that the term “perpendicular” should be read to mean “parallel.” *Id.*

There is no such logical inconsistency or contradiction in the claims at issue in this case. It is clear from the specification that there are multiple ways to extract byte sequence features from an executable attachment. The District Court’s holding that the intrinsic record discloses only one way to generate byte sequence features (extracting a representation of machine code instructions) cannot be squared with the express language of the Summary which states that “According to another embodiment, extracting the byte sequence feature may comprise creating a byte string representative of resources referenced by said executable attachment.” A49 (3:30-40); *see Rextord*, 274 F.3d at 1344-45 (finding that the term “portion” could refer to either a two-piece structure or a unified structure where the Summary of the Invention section disclosed both embodiments). The District Court’s holding also is inconsistent with the Detailed Description of Exemplary Embodiments which describe extracting byte sequences reflecting resource information. *E.g.*, A50 (6:26-28, 6:48-63) (“extract[ing] resource information from the binary that provides insight to its behavior,” such as DLLs and DLL function calls used by the binary). An example of a string reflecting these byte

sequences is depicted in Figure 3. A42 (Fig. 3). This byte string representative of resource information is an example of a byte sequence feature.

The prosecution history also confirms that neither the inventors nor the examiner ever considered resource information as anything other than a type of byte sequence feature. For example, in the originally-filed application, Claim 1 did not include the “creating a byte string representative of resources” clause. That limitation was originally in dependent Claim 4:

4. The method as defined in claim 1, wherein the step of extracting said byte sequence features from said executable attachment comprises creating a byte string representative of resources referenced by said executable attachment.

A3874. Columbia distinguished the prior art *not* on the basis that they lacked byte sequence features, but rather on the basis that they did not describe byte sequence features comprising “a byte string representative of resources referenced by said executable attachment.” A4681 (“neither Chen nor Kephart, whether considered separately or in combination, provides ‘creating a byte string representative of resources referenced by said executable attachment’”). Claim 4 then was allowed on the basis of the “byte string representative of resources” limitation and rewritten as an independent claim. This is stated directly in the Notice of Allowance:

[T]he above arts, singularly or in combination, fail to anticipate or render the following unique limitations of the independent claims in the instant invention:

“Claims 4 and 29: wherein extracting said byte sequence features from said executable attachment comprises creating a byte string representative of resources referenced by said executable attachment.”

A4750.

The meaning of “byte sequence feature” is not mutually exclusive with a “byte string representative of resources.” *See Ancora Techs., Inc. v. Apple, Inc.*, 744 F.3d 732, 739 (Fed. Cir. 2014) (distinguishing *Allen* as a case “where the patentee agreed that the claim language did not match what he regarded as his invention, as the intrinsic record unambiguously showed, and this court denied the patentee’s request to reject the claim language’s clear, ordinary meaning”). The District Court’s construction of “byte sequence feature,” and the corresponding holding that the claims are “inconsistent with the specification” is incorrect and should be reversed.

IV. THE DISTRICT COURT ERRED IN CONSTRUING THE ’115 AND ’322 PATENTS

The District Court’s construction of “anomalous” in the claims of the ’115 and ’322 patents should be reversed. The District Court should have construed “anomalous” according to its plain meaning in the intrinsic record as “behavior that deviates from normal and may correspond to an attack.” The District Court instead construed the term as requiring “[deviation/deviating] from a model of typical, attack-free computer system usage” generated with *only* “typical, attack-free” data. A10-12. There are no definitions or disclaimers that require limiting

the term to a model that is trained exclusively with “attack free” programs. The specification and claims of the ’115 and ’322 patents teach that the patents cover models trained using both attack-free and attack data.

A. The District Court Incorrectly Relied on an Unrelated Patent Family in Construing “Anomalous”

In ruling that “anomalous” required a deviation from a model of “normal computer system usage” trained only on “attack-free” data, the District Court did not cite the ’115 and ’322 patents’ specification. Instead, the District Court discussed only the unrelated ’084 and ’306 patents, which descend from a separate application. *Elkay Mfg. Co. v. Ebco Mfg. Co.*, 192 F.3d 973, 980 (Fed. Cir. 1999) (“related” patents must derive from a common application). The District Court also drew language for its construction from the ’084 and ’306 patents which appears nowhere in the ’115 and ’322 patents. The claims of the ’115 and ’322 patents recite a “model of function calls,” a different concept from the “computer system usage” discussed in the ’084 and ’306 patents and the court’s construction.

The District Court’s construction of terms in the ’115 and ’322 patents based on the Court’s construction of a different patent family is an independent ground for reversal. *e.Digital Corp. v. Futurewei Techs., Inc.*, 772 F.3d 723, 726-27 (Fed. Cir. 2014) (“[C]laims of unrelated patents must be construed separately.”); *see also Abbott Labs. v. Dey, L.P.*, 287 F.3d 1097, 1104-05 (Fed. Cir. 2002) (finding the relationship between two unrelated patents, although having common subject

matter, a common inventor, and the same assignee, “insufficient to render particular arguments made during prosecution of [one of the patents] equally applicable to the claims of [the other patent]”).

B. The Term “Anomalous” Does Not Include a Negative Limitation Preventing Analysis of Anything Other Than “Attack-Free” Data

The term “anomalous” does not require a model of typical, attack free computer system usage in which *only* attack free data is used to build the model.

“Anomalous” appears in all the claims of the ’115 and ’322 patents. For example, claim 1 reads:

1. A method for detecting anomalous program executions, comprising:

executing at least a part of a program in an emulator;

comparing a function call made in the emulator to a model of function calls for the at least a part of the program;

identifying the function call as anomalous based on the comparison; and

upon identifying the anomalous function call, notifying an application community that includes a plurality of computers of the anomalous function call.

’115 patent, A133 (cl. 1).

An anomaly is a deviation from normal. A566 (“anomalous” means “Deviating from the normal; irregular.”); A407-08.

The phrases “model of typical, attack-free computer system usage” or “employing only attack free data” appear nowhere in the ’115 and ’322 patents.

To be sure, other elements do require a model containing certain data— a “model of *function calls*.” But there is nothing in the claims or specification that justify interpreting “anomalous” to employ a model or requiring that the model be created solely and exclusively with data reflecting only “typical, attack free” “computer system usage.” The District Court should have adopted Columbia’s proposed construction and given “anomalous” its plain meaning in the specification as “behavior that deviates from normal and may correspond to an attack.”

1. The Claims Confirm That the Model Need Not Be Generated Only With Attack-Free Data

The structure of the claims themselves confirms that the District Court’s construction of “anomalous” cannot be correct. The District Court excluded the use of attack data to create the claim 1 model of function calls, but this is exactly the type of data that claim 8 requires be in the model. Dependent claim 8, an originally-filed claim, expressly specifies that “the model reflects attacks against the at least a part of the program.” A133 (cl. 8). Adopting and applying the District Court’s construction of “anomalous” creates an impossible result: a model restricted to exclusively attack-free data (claim 1, as construed) that must also include attack data (claim 8). Claim 8’s requirement that attack data be used in the model logically requires that the model of claim 1 can include that attack data. *Alcon Research, Ltd. v. Apotex Inc.*, 687 F.3d 1362, 1367 (Fed. Cir. 2012) (“It is

axiomatic that a dependent claim cannot be broader than the claim from which it depends.”).

Other claims likewise refute the District Court’s construction. In the claims as filed with the original parent application, the inventors made clear that “models of function calls” can be built using both normal and attack data. A6381-85. This concept remains in the claims as issued. For example, independent claim 1 of the ’115 patent refers to “a model of function calls for at least part of a program.” Claim 7 depends from claim 1 and specifies that “the model reflects normal activity of the at least a part of the program.” A133 (cl. 7). The reference to normal activity in claim 7 “creates a presumption that these dependent claim limitations are not included in the independent claim.” *GE Lighting Solutions, LLC v. Agilight, Inc.*, 750 F.3d 1304, 1310 (Fed. Cir. 2014).

2. The Specification Does Not Require Models Built Only With Attack-Free Data

Nothing in the specification meets the heavy standard for lexicography or disavowal required to limit the claim to attack free data. The “Summary” section of the specification does not use the words “normal,” “typical,” or “attack-free” in describing the invention. A124 (1:48-2:35). It describes the data that must be considered—function calls—but does not say they can *only* be function calls in attack-free programs.

The rest of the specification similarly describes examples in which the data set contains both attack-free and attack data. The “Detailed Description” section notes a purpose “to detect anomalous program executions that may be indicative of a malicious attack or program fault.” A125 (3:13-15). The section provides an “example” in which “the anomaly detector models normal program execution” and uses “normal data.” A125 (3:45-4:26). Another embodiment, called probabilistic anomaly detection (PAD), computes “consistency checks over the normal data.” A125 (4:9-17). These passages do not state that a model built exclusively using attack-free behavior always must be used.

To the contrary, the specification discusses using attack data to develop improved models and to repair software. The specification describes a system to learn from prior attacks: “once a vulnerability is detected, the system may use the detected vulnerability (and patch) to learn about other (e.g., similar) vulnerabilities” A127 (8:58-64). The specification continues:

A learning technique can be applied over multiple executions of a piece of code (e.g., a function or collection of functions) that may previously have been associated with a *failure*, or that is being proactively monitored. By retaining knowledge on program behavior across multiple executions, certain invariants (or probable invariants) may be learned, whose violation in future executions indicates an attack or imminent software fault.

A128 (9:41-48) (emphasis added). Data about previous attacks is used to build the model. In other words, the model is not formed exclusively from “typical, attack-free” data.

The specification also describes as an example a one-class support vector machine (OCSVM) in which “all the training data lies in the first class” but does not state that the class must be solely normal data. A126 (5:14-19). In fact, the original claims filed with the parent application specify that the class could be attack data. A6381-85.

3. The Prosecution History Describes Models Built Using Attack Data

During prosecution, neither the inventors nor the PTO examiner interpreted the term “anomalous” to require a deviation from a model of attack free computer system usage. Again, the record is just the opposite.

For example, the papers submitted as the provisional application reflect the inventors’ understanding that anomaly detection models are not limited to “attack free” data but could also include attack data. A3649-50 (describing determining a threshold value for a model using data “of approximately 300,000 records of which approximately 2,000 are labeled attacks”); A3724 (describing code repair techniques that “can only fix already-known attacks, *e.g.*, stack or heap-based buffer overflows”); A3739 (“[T]he normal data represents the regular flow of traffic through the network and, therefore, can include good data, potentially

harmful data, and noise.”); A3752 (“[T]he normal flow of data can conceivably include noise and/or malicious programs.”).

The examiner’s office actions also expressly acknowledged that attack data could be used as an ingredient in the model recited in the claims. In the first office action in the prosecution leading to the ’115 patent, the examiner rejected claim 8 of the application under a combination of two references, Vu and Chan. A6526. The examiner wrote: “Per claim 8, Vu does not explicitly disclose [that] the model reflects attacks against the at least a part of the program. Chan teaches using models that reflects [sic] attacks against a program during anomaly detection (see page I, section I, paragraphs 2-3).” *Id.* Thus, the examiner believed that the invention could encompass models trained using attack data. The applicant’s response to this rejection did not concern whether attack data could be used in the models—this was explicit from the claims as originally filed. Rather, Columbia amended independent claims to add an “application community” limitation without changing any claim language related to the composition of the model or arguing that the examiner’s citation of Chan in the context of claim 8 was improper. A6565; A6574.

There is no basis in the intrinsic record to limit the independent claims of the ’115 and ’322 patents to models created using only attack-free data. The District Court’s construction for the ’115 and ’322 patent should be reversed.

V. THE DISTRICT COURT ERRED IN CONSTRUING THE '084 AND '306 PATENTS

The District Court's constructions for terms of the '084 and '306 patents fail to apply well-established rules of claim interpretation. The District Court incorrectly converted claim elements reciting what must be present into negative limitations excluding anything else.

The '084 and '306 patent claims are directed to monitoring accesses to the operating system registry or file system, building a “probabilistic model of normal computer system usage” and then determining if a new access to these areas of the system “is an anomaly.” '084 patent, A93 (cl. 1). The District Court construed “probability” as having its plain meaning as “a likelihood that an event will occur or a condition will be present.” A10. In construing “probabilistic model of normal computer system usage,” the District Court should have done the same and construed the phrase as “a model of normal computer usage that employs probability.”³ A359-65; A1468-79. Similarly, the District Court should have given “anomaly / anomalous” its plain meaning in the intrinsic record as “behavior that deviates from normal and may correspond to an attack.” A365-66; A1479-80.

³ The District Court also construed the embedded phrase “normal computer system usage” to mean “typical, attack-free computer system usage.” A10. Although not a basis on which a judgment of non-infringement was entered, Columbia contends that this construction is incorrect for the same reasons as the other terms. “Normal” needed no separate construction. A360.

Instead, the District Court erred in construing *both terms* to require a “model” in which *only* attack-free data can be used as ingredients to build the model. The claims are not so limited and there is no lexicography or disclaimer that justifies reading such limitations into the claims.

A. The District Court Incorrectly Narrowed the Claimed “Probabilistic Model of Normal Computer System Usage”

1. The Claims Do Not Require a Model Generated With Only “Attack Free” Data

The claims spell out what must be present in the claimed systems and methods and they do not contain the negative limitation the District Court imposed. For example, claim 1 of the '084 patent specifies a “method for detecting intrusions” “comprising” —a term of inclusion, not exclusion—“gathering features from records of normal processes that access the operating system registry” and then “generating a probabilistic model of normal computer system usage based on the features.” A93 (22:21-39). The claims recite “gathering features from records of normal processes that access the operating system registry.” They do not require that the *only* “features” which can be used consist of an “attack-free” dataset.

Excluding anything in addition to what is set out in the claims is contrary to established precedent. The Federal Circuit has consistently held that doing *more* than what is claimed does not take activity outside the scope of a claim. *Sun-Tiger*,

Inc. v. Scientific Research Funding Group, 189 F.3d 1327, 1336 (Fed. Cir. 1999) (“It is fundamental that one cannot avoid infringement merely by adding elements if each element recited in the claims is found in the accused device.”) (quotation omitted). Certainly the claims require “records of normal processes that access the operating system registry” in generating the model (claim 1) but nothing in the claims makes this the *only* type of data that may be used or prohibits the use of anything else.

The inventors made a significant contribution in recognizing and explaining how normal registry and file system accesses could be used in an anomaly detection model. It is inconsistent with Federal Circuit precedent to hold that this critical insight can be used with impunity simply by adding to the claimed intrusion detection system data on events other than registry accesses or data that is not only “attack free.” This, however, is exactly what results under the District Court’s incorrect revision of the claims to require a model “based on *only*” normal processes that access the registry.

2. The Specification Does Not Support the District Court’s Construction

In order to depart from the claim language and import a negative limitation, there must be a “clear and unmistakable” disclaimer. *Teleflex, Inc. v. Ficosa N. Am. Corp.*, 299 F.3d 1313, 1325 (Fed. Cir. 2002); *Omega Eng’g v. Raytek Corp.*, 334 F.3d 1314, 1322-23 (Fed. Cir. 2003) (finding no disclaimer or lexicography

“that would justify adding that negative limitation”). The District Court’s ruling limiting the model ingredients to only “attack free” data is contrary to materials incorporated into the specification and in the prosecution history.

During claim construction briefing, Symantec argued that the model must be based on attack free data, relying on a remark in the specification referring to “generating a probabilistic model of normal computer system usage, e.g., free of attacks.” A84 (3:25-27). Symantec, however, conflated two questions: (a) what the model must reflect (“normal computer system usage”); and (b) what ingredients are used to create the model. The claims require the use of normal registry accesses as an ingredient but do not exclude supplemental data or specify that *only* “attack free” data can be used.

Certainly there are examples in the intrinsic record describing experiments in which attack free data was used. *See, e.g.*, A90 (15:5-16). For example, Symantec’s briefing cited the provisional application for the ’084 and ’306 patents, where the inventors wrote that they “used only attack free data for training.” A776; A932. But the inventors also emphasized that “a more sophisticated algorithm can be used” and how the one selected for the early experiments was “simple and efficient,” not the only option. A929. Neither the provisional application nor the specification state that using only an attack free dataset was a necessary part of the invention. Of course, the term “*only* attack free data” does

not appear in the claims. Nor does it appear in the specification. To be sure, the specification describes the importance of using normal access to registries and file systems as part of anomaly detection, but never excludes the use of other types of data. A85 (5:61-6:24).

Indeed, suggesting that the claims are limited to the use of an attack free dataset would make no sense. As set out in the Background of the Invention, the inventors had previously established that using attack free data supplemented by attack data in constructing an anomaly detection model makes for a more robust system. A83 (2:39-64); A398-402. The inventors explained in one of their papers referenced in the specification that “[t]ypical approaches to anomaly detection methods require training over clean data (normal data containing no anomalies) in order to build a model that detects anomalies. There are several inherent drawbacks to this approach.” A674. The paper describes constructing an anomaly detection model based on mixed data that “outperforms” other methods: “[t]he data provides normal *and intrusion traces* of system calls for several processes.” A677 (emphasis added); A394. Further, the paper describes constructing a model of normal computer system usage where “the normal data can effectively be modeled” by training on mixed data. A679.

Patent applications from the inventors incorporated into the specification, U.S. Patent Application No. 10/352,342 (the ’342 application) and 10/208,432 (the

'432 application), further support this point. A89 (14:1-10). The '084 and '306 patent specification says that the invention can use “the data stored in the database” described in the '342 application. A89 (14:3-5). The '342 application in turn gives examples of data sets that are compatible with the techniques in the '084 and '306 patents. The '342 application critiques data sets that are attack-free: “[t]his data can be very expensive because the process of manually cleaning the data is quite timing consuming. Also, some algorithms require a very large amount of normal data which increases the cost.” A959-60; *see also* A967 (describing use of “heterogeneous data”); A987 (database containing data “which is suspected to contain intrusions”).

At the District Court, Symantec argued that the '342 application supports its construction by distinguishing anomaly detection from “misuse detection.” A773-74. Symantec relied on a sentence in the '342 application that states “These algorithms need to train over data that contains no intrusions.” A776; A997. But Symantec ignored the context of that sentence. The '342 application introduces the “no intrusion” anomaly detection algorithm as just the “traditional” approach. A996. Moreover, in a paragraph that immediately follows, the '342 application discusses “unsupervised anomaly detection algorithms” which “examine unlabeled data and attempt to detect intrusions buried within the unlabeled data.” A997-98. This unlabeled data contains both normal and abnormal (intrusion) data. The

algorithm can discriminate between the two classes because “intrusions are very rare compared to the normal data and they are also quantitatively different.” *Id.* Nothing in the claims of the ’084 and ’306 patents excludes the use of unsupervised anomaly detection, which encompasses building models which are supplemented with data regarding abnormal processes.

Further, an exemplary embodiment described in the specification includes using attack data in building the model. The inventors observed that malicious programs often install quietly without using the Windows install shield program. A86 (7:60-8:1). The inventors teach adjusting the model to take this unique behavior into account. A86 (7:63-8:1). The model used to detect attacks in this situation thus incorporates insights about attacks. The District Court’s construction excludes this embodiment.

The specification also teaches that attack data is needed in determining when a program is sufficiently abnormal to trigger a prediction that the behavior is malicious. This is called a “threshold.” The specification gives an example of a training phase using “normal program executions *interspersed with attacks among normal process executions.*” A90 (15:18-21) (emphasis added). The number of false positives on the data is computed using different threshold scores. A90 (15:52-67). A final threshold value to use in a working system is selected based on

the performance of the system in training as applied to attack data. A91 (17:24-42). The District Court's constructions exclude this embodiment.

3. The Prosecution History Contains No Disclaimer and Does Not Support the District Court's Construction

Symantec also incorrectly argued to the District Court that statements during prosecution limit the claims to models created with only attack free data. The prosecution history contains no such assertions or disclaimer. Indeed, the statements Symantec relied on establish that the claims encompass models generated with normal data supplemented with attack data.

The statements at issue concern the examiner's rejection of claims in light of a reference to U.S. Patent Publication US2003/0070003 ("Chong"). Columbia argued that Chong did not disclose the claimed invention in part because Chong disclosed only using attack data: "attacker type," "attack objective," "attack intent," "attacker location," "attack methods," "target type," and "probing activity." A1313. Columbia therefore argued that "Chong provides no teaching relating to whether processes are 'normal'" A5804.

The examiner did not agree with Columbia's reading and concluded that Chong disclosed using *both* normal and attack data. The examiner wrote: "Chong states in paragraph 0014 that *any information* corresponding to events associated with the computer network may be collected." A5813 (emphasis added). If the '084 and '306 patents were limited to models using exclusively "attack free" data,

then the examiner could not have cited Chong as supplying the “records of normal processes” limitation. Columbia then overcame the rejection under Chong by pointing to other elements dealing with monitoring accesses to “operating system registry” and “determining the likelihood of observing an event that was not observed during the gathering of features.” A5837. The PTO agreed these were not present and allowed the claims. A6083-84.

There is no disclaimer. *See Micro Chem., Inc. v. Great Plains Chem. Co., Inc.*, 194 F.3d 1250, 1260-1261 (Fed. Cir. 1999) (no disclaimer when argued distinction over prior art was not basis for allowance).

B. The District Court Incorrectly Construed “Anomaly” and “Anomalous,” Which Should Have Received Their Plain Meaning

The claims and the specification do not contain words of manifest exclusion or restriction that limit the terms “anomaly” and “anomalous” to deviations from a model generated *only* with “attack free” data. The meaning of “anomaly” and “anomalous” is a divergence from normal. A566 (defining “anomalous” as “Deviating from the normal; irregular.”); A407-08. The specification similarly describes “anomalous” as “behavior that deviates from normal behavior and which may correspond to an attack.” A83 (2:34-37). This is the proper construction the District Court should have adopted.

CONCLUSION

The District Court's claim construction rulings should be reversed. Columbia stipulated to a final judgment of non-infringement and invalidity of claims 1 and 16 of the '544 patent based solely on the District Court's incorrect claim constructions. This Court should vacate the final judgment and remand for further proceedings in view of the correct claim constructions.

Dated: January 20, 2015

Respectfully Submitted,

/s/ David I. Gindler

David I. Gindler

IRELL & MANELLA LLP

David I. Gindler

Jason G. Sheasby

Richard M. Birnholz

Joseph M. Lipner

Gavin Snyder

*Attorneys for Plaintiff-Appellant
The Trustees of Columbia University
in the City of New York*

ADDENDUM

ADDENDUM TABLE OF CONTENTS

1. Order Granting Final Judgment Pursuant to Rule 54(b) of the Federal Rules of Civil Procedure, entered on November 4, 2014, in *The Trustees of Columbia University in the City of New York v. Symantec Corporation*, No. 3:13-CV-00808-JRS, docket no. 151 (E.D. Va.).
2. Claim Construction Order, entered on October 7, 2014, docket no. 123.
3. Memorandum Order Granting Motion for Clarification of Claim Construction Order, entered on October 23, 2014, docket no. 146.
4. Order Granting Joint Motion for Entry of Final Judgment, entered on November 4, 2014, docket no. 150.
5. U.S. Patent No. 7,487,544.
6. U.S. Patent No. 7,979,907.
7. U.S. Patent No. 7,448,084.
8. U.S. Patent No. 7,913,306.
9. U.S. Patent No. 8,074,115.
10. U.S. Patent No. 8,601,322.

1

IN THE UNITED STATES DISTRICT COURT
FOR THE EASTERN DISTRICT OF VIRGINIA
RICHMOND DIVISION

THE TRUSTEES OF COLUMBIA
UNIVERSITY IN THE CITY OF NEW
YORK,

Plaintiff

vs.

SYMANTEC CORPORATION,

Defendant

Civil Action No. 3:13-cv-00808-JRS

JURY TRIAL DEMANDED

**FINAL JUDGMENT PURSUANT TO RULE 54(b) OF THE
FEDERAL RULES OF CIVIL PROCEDURE**

THIS MATTER is before the Court on the Stipulation And Joint Motion For Entry Of Final Judgment On All Infringement Claims Based On The Court’s Claim Construction And Indefiniteness Rulings, And For An Order Staying Remaining Decoy Claims Pending Appeal of plaintiff The Trustees of Columbia University in the City of New York (“Columbia”) and defendant Symantec Corporation (“Symantec”) (Dkt. 147) (“Stipulation”). Good cause appearing, it is ORDERED and ADJUDGED that:

1. On December 5, 2013, Columbia filed its Complaint against Symantec alleging, among other things, that Symantec infringed United States Patent No. 7,487,544 (the ‘544 patent), 7,979,907 (the ‘907 patent), 7,448,084 (the ‘084 patent), 7,913,306 (the ‘306 patent), and 8,074,115 (the ‘115 patent).

2. Columbia also alleged a claim for Correction of Inventorship with respect to United States Patent No. 8,549,643 (the '643 patent), a patent presently assigned to Symantec. The '643 patent relates to "decoy" technology.

3. On December 24, 2013, Columbia filed its First Amended Complaint against Symantec alleging in its First through Sixth Claims for Relief that Symantec infringed the '544 patent, the '907 patent, the '084 patent, the '306 patent, the '115 patent, and United States Patent No. 8,601,322 (the '322 patent) (collectively, the "Asserted Patents"). Dkt. 12.

4. Columbia also alleged in its Seventh through Eleventh Claims for Relief claims for Fraudulent Concealment, Unjust Enrichment, and Conversion relating to Columbia intellectual property in "decoy" technology along with claims for Correction of Inventorship with respect to the '643 patent.

5. On January 14, 2014, Symantec answered Columbia's First Amended Complaint, denying the material allegations and asserting, among other things, affirmative defenses of non-infringement, invalidity and unenforceability of the Asserted Patents. Dkt. 20.

6. On October 7, 2014, following briefing and a hearing, the Court issued a Claim Construction Order construing certain claims of the Asserted Patents. Dkt. 123.

7. With respect to the '544 and '907 patents, the Court's Claim Construction Order provided, among other things, the following constructions and rulings:

- a. "Byte Sequence Feature": Feature that is a representation of machine code instructions of the executable. "Feature" is a property or attribute of data which may take on a set of values.
- b. "Wherein [the step of] extracting said byte sequence features from said executable attachment comprises creat[ing/e] a byte string representative of

resources referenced by said executable attachment”: Indefinite. The Court finds that the “resource information” feature extraction embodiment is separate and distinct from the “byte sequence feature” extraction embodiment. Because Claims 1 and 16 of the ‘544 patent conflate these terms, and thus are inconsistent with the specification, the Court holds that this term is indefinite. *See Allen Eng’g Corp. v. Bartell Indus., Inc.*, 299 F.3d 1336, 1349 (Fed. Cir. 2002).

8. With respect to the ‘084 and ‘306 patents, the Court’s Claim Construction Order provided, among other things, the following constructions for the ‘084 and ‘306 patents:

- a. “Probabilistic Model of Normal Computer System Usage”: Model of typical attack-free computer system usage that employs probability. “Probability” is the likelihood that an event will occur or a condition will be present.
- b. “Normal Computer System Usage”: Typical, attack free computer system usage.
- c. “Anomaly” / “Anomalous”: Deviation/deviating from a model of typical, attack-free computer system usage.

9. With respect to the ‘115 and ‘322 patents, the Court’s Claim Construction Order provided, among other things, the following constructions: “Anomalous”: Deviation/deviating from a model of typical, attack-free computer system usage.

10. On October 23, 2014, the Court issued a Memorandum Order granting Columbia’s Motion for Clarification of Claim Construction Order. (Dkt. 146). Columbia sought clarification on whether the Court intended its construction of the terms “probabilistic model of normal computer system usage” in the ‘084 / ‘306 patents and “anomalous” in the ‘084 / ‘306

patents and '115 / '322 patents – both of which the Court construed as requiring a “model of typical, attack-free computer system usage” – meant that the model may be generated with normal data and also attack data or required that the model must be generated with *only* “typical, attack free” data. (Dkt. 128). The Court’s Order on the clarification motion stated that the Court’s construction for both the '084/'306 patents and the '115/'322 patents required that the model “is generated with only attack-free data.” (Dkt. 146).

11. Based on the Court’s October 7, 2014 Claim Construction Order (Dkt. 123) with respect to the '544 patent and '907 patent, and based on the Stipulation of the Parties, this final judgment of non-infringement of claims 6, 10-14, 28-34, 37-41, and 43 of the '544 patent and claims 10-15 and 18-20 of the '907 patent is entered against Columbia on the grounds that the MalHeur, Insight, SONAR, and MutantX systems employed in or with Norton Internet Security, Norton AntiVirus, Norton 360, Symantec Endpoint Protections, Symantec Endpoint Protection Small Business Edition, and Symantec Mail Security through the date of this judgment (i.e., all the accused products) do not satisfy the “byte sequence feature” limitations of those claims under the Court’s construction of the term “byte sequence feature” to mean a “Feature that is a representation of machine code instructions of the executable.”

12. Based on the Court’s October 7, 2014 Claim Construction Order (Dkt. 123) with respect to the '544 patent, and based on the Stipulation of the Parties, this final judgment of invalidity of claims 1 and 16 of the '544 patent (and their dependent claims 2-5 and 17-27) is entered against Columbia on the ground that the Court has ruled that the following term recited in these claims is indefinite under 35 U.S.C. § 112 ¶ 2: “Wherein [the step of] extracting said byte sequence features from said executable attachment comprises creat[ing/e] a byte string representative of resources referenced by said executable attachment.” The final judgment in

Paragraphs 11 and 12 herein fully resolves Columbia's First and Second Claims for Relief in the District Court, making the Claims ripe for appellate review of the Claim Construction Orders.

13. Based on the Court's October 7, 2014 and October 23, 2014 Claim Construction Orders (Dkt. 123 & 146), and based on the Stipulation of the parties, this final judgment of non-infringement of claims 1, 3-10, 14, 16-26, and 28 of the '084 patent and claims 1-4 and 7-8 of the '306 patent is entered against Columbia on the grounds that the MalHeur, Insight, SONAR, and MutantX systems employed in or with Norton Internet Security, Norton AntiVirus, Norton 360, Symantec Endpoint Protections, and Symantec Endpoint Protection Small Business Edition through the date of this judgment (i.e., all the accused products) do not satisfy the "probabilistic model of normal computer system usage" and "anomaly / anomalous" limitations of those patents under the Court's construction of these terms to require a "model" "generated with only attack-free data" (Dkt. 146). Columbia also has reserved the right to address on appeal the Court's construction of the subsidiary phrase "normal computer system usage" in all locations in which it appears in the relevant claims. This final judgment fully resolves Columbia's Third and Fourth Claims for Relief in the District Court, making the Claims ripe for appellate review of the Claim Construction Orders.

14. Based on the Court's October 7, 2014 and October 23, 2014 Claim Construction Orders (Dkt. 123 & 146), and the Stipulation of the parties, this final judgment of non-infringement of claims 1-42 of the '115 patent and claims 1-27 of the '322 patent is entered against Columbia on the grounds that the MalHeur, Insight, SONAR, and MutantX systems employed in or with Norton Internet Security, Norton AntiVirus, Norton 360, Symantec Endpoint Protections, and Symantec Endpoint Protection Small Business Edition through the date of judgment (i.e., all of the accused products) do not satisfy the "anomalous" limitation of

those patents under the Court's construction of this term to require "a model" that is "generated with only attack-free data" (Dkt. 146). This final judgment fully resolves Columbia Fifth and Sixth Claims for Relief in the District Court, making the Claims ripe for appellate review of the Claim Construction Orders.

15. The present action presents more than one claim for relief. The final judgments referenced above are only directed to the First through Sixth Claims for Relief and do not alter Columbia's and Symantec's respective rights in Columbia's Seventh through Eleventh Claims for Relief regarding the "decoy" technology. The decoy technology claims relate to a distinct patent presently assigned to Symantec, which is not part of the same family as the patents in the First through Sixth Claims for Relief, and is not related to the Symantec products at issue in Columbia's First through Sixth Claims for Relief. As a result, the United States Court of Appeals for the Federal Circuit would not have to consider the same issues more than once if and when the Seventh through Eleventh Claims for Relief are appealed.

16. In light of the foregoing, there is no just reason to delay the entry of a final, appealable judgment on Columbia's First through Sixth Claims for Relief pursuant to Rule 54(b) of the Federal Rules of Civil Procedure. Therefore, this Court enters final judgment as to the First through Sixth Claims for Relief as set forth above in order to conserve judicial resources, to avoid the time and expense of further discovery, motion practice and other proceedings in this Court, and to allow Columbia to appeal this Court's claim construction and indefiniteness rulings.

17. Columbia and Symantec reserve their rights to raise on appeal any and all issues concerning claim interpretation and/or claim definiteness raised by either Party in the District

2

UNITED STATES DISTRICT COURT
EASTERN DISTRICT OF VIRGINIA
RICHMOND DIVISION

THE TRUSTEES OF COLUMBIA
UNIVERSITY IN THE CITY OF NEW YORK,

Plaintiff,

Action No. 3:13-CV-808

v.

SYMANTEC CORPORATION,

Defendant.

CLAIM CONSTRUCTION ORDER

Pursuant to the directions of *Markman v. Westview Instruments, Inc.*, 517 U.S. 370 (1996), the Court hereby construes the following disputed terms of the allegedly infringed patents in the above referenced case:

I. First Family of Patents ('544 and '907)

1. "Byte Sequence Feature": Feature that is a representation of machine code instructions of the executable. "Feature" is a property or attribute of data which may take on a set of values.
2. "Email Interface": The component that reintegrates filtered email back into normal email traffic and may send the model generator 240 each attachment to be analyzed further.
3. "Wherein [the step of] extracting said byte sequence features from said executable attachment comprises creat[ing/e] a byte string representative of resources referenced by said executable attachment": Indefinite. The Court finds that the "resource information" feature extraction embodiment is separate and distinct from the "byte sequence feature" extraction embodiment. Because Claims 1 and 16 of the '544 patent conflate these terms, and thus are inconsistent with the specification, the Court holds that this term is indefinite. *See Allen Eng'g Corp. v. Bartell Indus., Inc.*, 299 F.3d 1336, 1349 (Fed. Cir. 2002).

II. Second Family of Patents ('084 and '306)

1. "Operating System Registry": A database of information about a computer's configuration, utilized by an operating system, organized hierarchically as a tree, with entries consisting of keys and values.

- 2. “Probabilistic Model of Normal Computer System Usage”: Model of typical attack-free computer system usage that employs probability. “Probability” is the likelihood that an event will occur or a condition will be present.

“Normal Computer System Usage”: Typical, attack-free computer system usage.

- 3. “Anomaly” / “Anomalous”: Deviation/deviating from a model of typical, attack-free computer system usage.

III. Third Family of Patents (‘115 and ‘322)

- 1. “Anomalous”: Deviation/deviating from a model of typical, attack-free computer system usage.
- 2. “Emulator”: Software, alone or in combination with hardware, that permits the monitoring and selective execution of certain parts, or all, of a program.
- 3. “Application Community”: Members of a community running the same program or a selected portion of the program.

Let the Clerk send a copy of this Order to all counsel of record.

It is SO ORDERED.

<p>_____/s/_____ James R. Spencer Senior U. S. District Judge</p>

ENTERED this 7th day of October 2014.

3

IN THE UNITED STATES DISTRICT COURT
FOR THE EASTERN DISTRICT OF VIRGINIA
RICHMOND DIVISION

THE TRUSTEES OF COLUMBIA
UNIVERSITY IN THE CITY OF NEW
YORK,

Plaintiff,

v.

SYMANTEC CORPORATION,

Defendant.

Civil Action No. 3:13-CV-808

JURY TRIAL DEMANDED

MEMORANDUM ORDER

THIS MATTER is before the Court on Plaintiff's Motion for Clarification of Claim Construction Order ("Motion") (ECF No. 128). Defendant filed a brief opposition on October 10, 2014. The parties have waived oral argument on this matter. Therefore, the issue is ripe for disposition. The Motion is hereby GRANTED and the Claim Construction Order issued on October 7, 2014 ("Order") (ECF No. 123) is clarified below.

On October 9, 2010, Columbia filed the instant Motion asking the Court to clarify its Order with respect to the terms "probabilistic model of normal compute system usage" in the '084/'306 patents and "anomalous" in the '115/'322 patents. This Court's Order defined "probabilistic model of normal computer system usage" as a "[m]odel of typical attack-free computer system usage that employs probability." The Court defined "anomalous" as "[d]eviation/deviating from a model of typical, attack-free computer system usage." Specifically, Columbia now seeks clarification on whether the Court intended its construction to mean that the claimed model may be generated with normal data and also attack data or whether the model must be generated with *only* "typical, attack free" data. In the claim construction briefs, Columbia argued the former interpretation, while Symantec argued the latter.

As an initial matter, the Court notes that its Order construed the specific disputed terms

that were originally presented in the parties' claim construction briefs. Columbia now seeks a further interpretation of the disputed terms to answer the question Columbia posed at the claim construction hearing, that is, "What information is used to construct the model?" (See Tr. Markman Hr'g 75:13–14.) Despite not originally requesting a construction of such issue, the Court chooses to now clarify its ruling with respect to this question. See *U.S. Surgical Corp. v. Ethicon, Inc.*, 103 F.3d 1554, 1568 (Fed. Cir. 1997) (emphasis added) ("Claim construction is a matter of resolution of disputed meanings and technical scope, to *clarify* and when necessary to explain what the patentee covered by the claims, for use in the determination of infringement.").

"Anomaly detectors . . . do not operate by looking for malicious activity directly. Rather, they look for deviations from normal activity." ('084 patent at 7:47–49.) Claim 1 of the '084 patent mirrors this concept: "[A]nalyzing features from a record of a process that accesses the operating system registry *to detect deviations from normal computer system usage to determine whether the access to the operating system registry is an anomaly.*" (*Id.* at 22:30–34) (emphasis added). Logically, if the anomaly detection systems detect *deviations* from normal activity, that normal activity must be "attack-free" activity. Applying this logic to the rest of claim 1, which gathers "features from records of *normal* processes" and then *generates* "a probabilistic model of normal computer system usage based on [those] features," it follows that the model is generated with only attack-free data.

Let the Clerk send a copy of this Order to all counsel of record

It is SO ORDERED.

<p>_____/s/_____ James R. Spencer Senior U. S. District Judge</p>

ENTERED this 23rd day of October 2014.

4

IN THE UNITED STATES DISTRICT COURT
FOR THE EASTERN DISTRICT OF VIRGINIA
RICHMOND DIVISION

THE TRUSTEES OF COLUMBIA
UNIVERSITY IN THE CITY OF NEW
YORK,

Plaintiff

vs.

SYMANTEC CORPORATION,

Defendant

Civil Action No. 3:13-cv-00808-JRS

JURY TRIAL DEMANDED

ORDER

THIS MATTER is before the Court on the Joint Motion for Entry of Final Judgment On All Infringement Claims Based On The Court’s Claim Construction And Indefiniteness Rulings And For An Order Staying Remaining Decoy Claims Pending Appeal brought by plaintiff The Trustees of Columbia University in the City of New York (“Columbia”) and defendant Symantec Corporation (“Symantec”). Having considered the pleadings filed by the parties and being fully informed in the matter, and for other good cause, the Joint Motion is GRANTED.

Based on the parties’ stipulation and as set forth therein, the Court hereby finds that there is no just reason to delay the entry of a final, appealable judgment on Columbia’s First through Sixth Claims for Relief pursuant to Rule 54(b) of the Federal Rules of Civil Procedure. The Court will separately enter a form of Final Judgment on Columbia’s First through Sixth Claims for Relief under Rule 54(b) of the Federal Rules of Civil Procedure. The prosecution histories of the asserted patents are made of record. The Court further finds that due to their size, such

prosecution histories are not to be electronically filed, imaged or maintained in the ECF system, and shall be filed with the Clerk on DVD or other appropriate removable storage medium.

Columbia's Seventh through Eleventh Claims for Relief are hereby ordered STAYED pending resolution of any appeal on Columbia's First through Sixth Claims for Relief or further order from this Court.

Let the Clerk file this Order electronically, notifying all counsel of record accordingly.

It is so ORDERED.

Date: 11-4-14
Richmond, Virginia

 /s/
James R. Spencer
Senior U. S. District Judge

5

(12) **United States Patent**
Schultz et al.

(10) **Patent No.:** **US 7,487,544 B2**
 (45) **Date of Patent:** **Feb. 3, 2009**

(54) **SYSTEM AND METHODS FOR DETECTION OF NEW MALICIOUS EXECUTABLES**

5,832,208 A * 11/1998 Chen et al. 726/24
 6,016,546 A * 1/2000 Kephart et al. 726/24
 6,161,130 A * 12/2000 Horvitz et al. 709/206
 6,275,850 B1 * 8/2001 Beyda et al. 709/206

(75) Inventors: **Matthew G. Schultz**, Ithaca, NY (US);
Eleazar Eskin, Santa Monica, CA (US);
Erez Zadok, Middle Island, NY (US);
Manasi Bhattacharyya, Flushing, NY (US);
Stolfo J. Salvatore, Ridgewood, NJ (US)

(Continued)

(73) Assignee: **The Trustees of Columbia University in the city of New York**, NY, NY (US)

OTHER PUBLICATIONS

Jeffrey O. Kephart and William C. Arnold, "Automatic Extraction of Computer Virus Signatures," *4th Virus Bulletin International Conference*, pp. 178-184, 1994.

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 1122 days.

(Continued)

(21) Appl. No.: **10/208,432**

Primary Examiner—Gilberto Barron, Jr.
Assistant Examiner—Abdulkhakim Nobahar
 (74) *Attorney, Agent, or Firm*—Baker Botts LLP

(22) Filed: **Jul. 30, 2002**

(65) **Prior Publication Data**

(57) **ABSTRACT**

US 2003/0065926 A1 Apr. 3, 2003

Related U.S. Application Data

(60) Provisional application No. 60/308,622, filed on Jul. 30, 2001, provisional application No. 60/308,623, filed on Jul. 30, 2001.

(51) **Int. Cl.**
G06F 21/00 (2006.01)

(52) **U.S. Cl.** **726/24**; 713/188

(58) **Field of Classification Search** 726/13,
 726/22-25; 713/156, 188; 709/206, 207,
 709/225

See application file for complete search history.

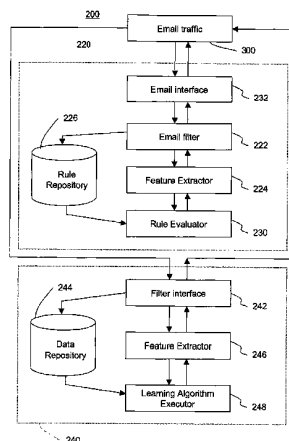
(56) **References Cited**

U.S. PATENT DOCUMENTS

5,452,442 A 9/1995 Kephart et al.
 5,485,575 A * 1/1996 Chess et al. 714/38
 5,675,711 A 10/1997 Kephart et al.
 5,765,170 A * 6/1998 Morikawa 707/200

A system and methods for detecting malicious executable attachments at an email processing application of a computer system using data mining techniques. The email processing application may be located at the server or at the client or host. The executable attachments are filtered from said email, and byte sequence features are extracted from the executable attachment. The executable attachments are classified by comparing the byte sequence feature of the executable attachment to a classification rule set derived from byte sequence features of a data set of known executables having a predetermined class in a set of classes, e.g., malicious or benign. The system is also able to classify executable attachments as borderline when the difference between the probability that the executable is malicious and the probability that the executable is benign are within a predetermined threshold. The system can notify the user when the number of borderline attachments exceeds the threshold in order to refine the classification rule set.

43 Claims, 7 Drawing Sheets



US 7,487,544 B2

Page 2

U.S. PATENT DOCUMENTS

6,598,076	B1 *	7/2003	Chang et al.	709/206
6,778,995	B1	8/2004	Gallivan	
6,820,081	B1	11/2004	Kawai et al.	
6,826,609	B1 *	11/2004	Smith et al.	709/225
6,888,548	B1	5/2005	Gallivan	
6,978,274	B1	12/2005	Gallivan et al.	
7,035,876	B2	4/2006	Kawai et al.	
7,080,076	B1	7/2006	Williamson et al.	
2002/0059383	A1 *	5/2002	Katsuda	709/206
2002/0065892	A1 *	5/2002	Malik	709/206

OTHER PUBLICATIONS

R Kohavi, "A study of cross-validation and boot-strap for accuracy estimation and model selection," *IJCAI*, 1995.

Ronald L. Rivest. "The MD5 Message Digest Algorithm." published as Internet RFC 1321, Apr. 1992. <http://www.freesoft.org/CIE/RFC/1321/>.

Stephen R. van den Berg and Philip Guenther, "Procmail." online publication, 2001. <http://www.procmail.org>.

Steve R. White, Morton Swimmer, Edward J. Pring, William C. Arnold, David M. Chess, and John F. Morar, "Anatomy of a Commercial-Grade Immune System," IBM Research White Paper, 1999.

Eleazar Eskin et al. "System and Methods for Intrusion Detection with Dynamic Window Sizes," filed Jul. 30, 2000, U.S. Appl. No. 10/208,402.

U.S. Appl. No. 10/352,343, filed Jan. 27, 2003 claiming priority to P34981 (070050.1936) U.S. Appl. No. 60/351,857, filed Jan. 25, 2001, entitled "Behavior Based Anomaly Detection For Host-Based Systems For Detection Of Intrusion In Computer Systems," of Frank Apap, Andrew Honig, Shlomo Hershkop, Eleazar Eskin and Salvatore J. Stolfo.

U.S. Appl. No. 10/352,342, filed Jan. 27, 2003 claiming priority to U.S. Appl. No. 60/351,913, filed Jan. 25, 2002, entitled "Data Warehouse Architecture For Adaptive Model Generation Capability In Systems For Detecting Intrusion In Computer Systems," of Salvatore J. Stolfo, Eleazar Eskin, Matthew Miller, Juxin Zhang and Zhi-Da Zhong.

U.S. Appl. No. 10/327,811, filed Dec. 19, 2002 claiming priority to U.S. Appl. No. 60/342,872, filed Dec. 20, 2001, entitled "System And Methods for Detecting A Denial-Of-Service Attack On A Computer System" of Salvatore J. Stolfo, Shlomo Hershkop, Rahul Bhan, Suhail Mohiuddin and Eleazar Eskin.

U.S. Appl. No. 10/320,259, filed Dec. 16, 2002 claiming priority to U.S. Appl. No. 60/328,682, filed Oct. 11, 2001 and U.S. Appl. No. 60/352,894, filed Jan. 29, 2002, entitled "Methods of Unsupervised Anomaly Detection Using A Geometric Framework" of Eleazar Eskin, Salvatore J. Stolfo and Leonid Portnoy.

U.S. Appl. No. 10/269,718, filed Oct. 11, 2002 claiming priority to U.S. Appl. No. 60/328,682, filed Oct. 11, 2001 and U.S. Appl. No. 60/340,198, filed Dec. 14, 2001, entitled "Methods For Cost-Sensitive Modeling For Intrusion Detection" of Dec. 14, 2001, entitled "Methods For Cost-Sensitive Modeling For Intrusion Detection" of Salvatore J. Stolfo, Wenke Lee, Wei Fan and Matthew Miller.

U.S. Appl. No. 10/269,694, filed Oct. 11, 2002 claiming priority to U.S. Appl. No. 60/328,682, filed Oct. 11, 2001 and U.S. Appl. No. 60/339,952, filed Dec. 13, 2001, entitled "System And Methods For Anomaly Detection And Adaptive Learning" of Wei Fan, Salvatore J. Stolfo.

U.S. Appl. No. 10/222,632, filed Aug. 16, 2002 claiming priority to U.S. Appl. No. 60/312,703, filed Aug. 16, 2001 and U.S. Appl. No. 60/340,197, filed Dec. 14, 2001, entitled "System And Methods For Detecting Malicious Email Transmission" of Salvatore J. Stolfo, Eleazar Eskin, Manasi Bhattacharyya and Matthew G. Schultz.

* cited by examiner

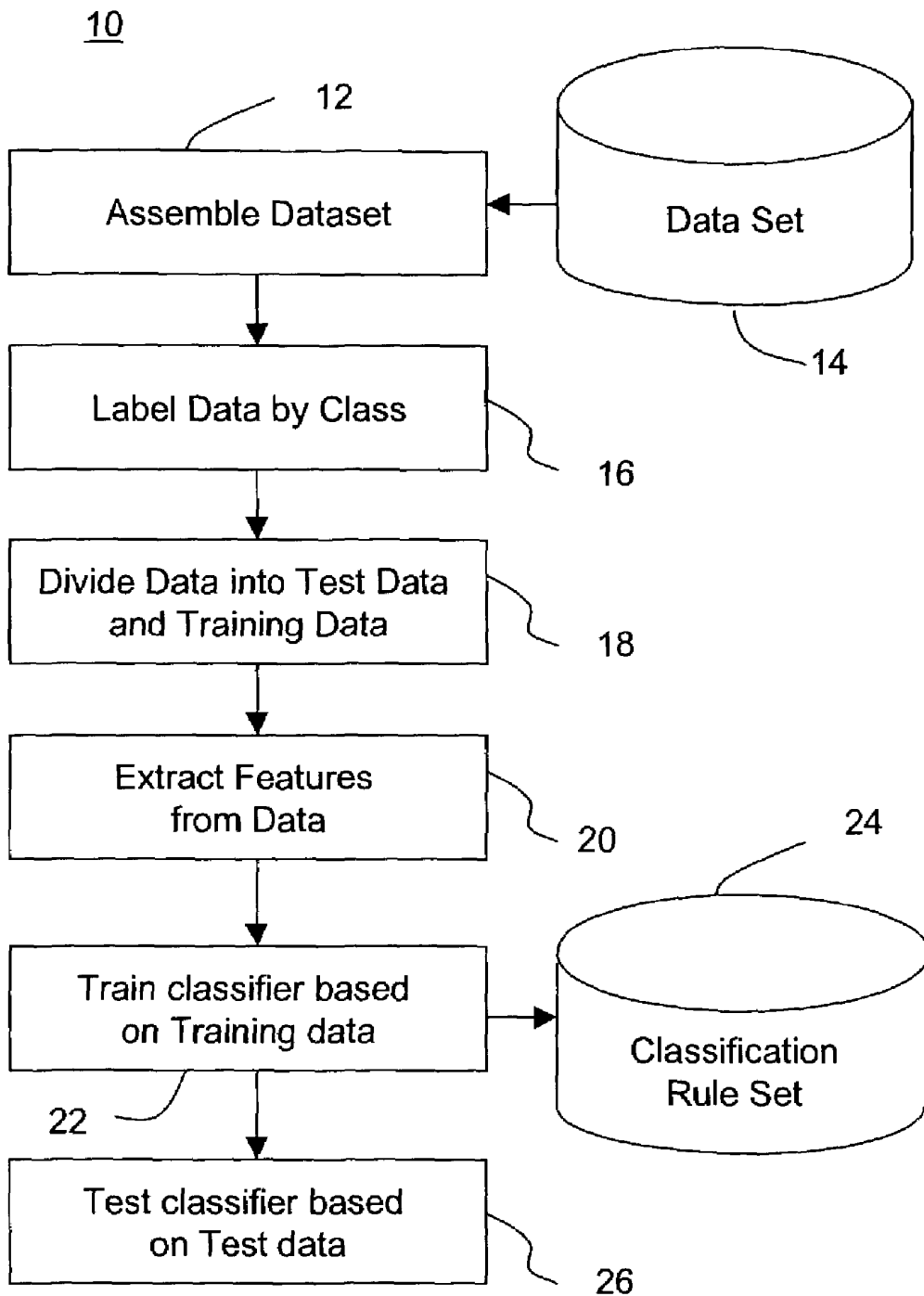


FIG. 1

646e	776f	2e73	0a0d	0024	0000	0000	0000
454e	3c0f	026c	0009	0000	0000	0302	0004
0400	2800	3924	0001	0000	0004	0004	0006
000c	0040	0060	021e	0238	0244	02f5	0000
0001	0004	0000	0802	0032	1304	0000	030a

FIG. 2

$\neg advapi32 \wedge avicap32 \wedge \dots \wedge winmm \wedge \neg wsock32$

FIG. 3

advapi32.AdjustTokenPrivileges()
 $\wedge advapi32.GetFileSecurityA() \wedge \dots$
 $\wedge wsock32.recv() \wedge wsock32.send()$

FIG. 4

$advapi32 = 2 \wedge avicap32 = 10 \wedge \dots$
 $\wedge winmm = 8 \wedge wsock32 = 2$

FIG. 5

malicious := \neg *user32.EndDialog()* \wedge
 kernel32.EnumCalendarInfoA()
malicious := \neg *user32.LoadIconA()* \wedge
 \neg *kernel32.GetTempPathA()* \wedge \neg *advapi32.*
malicious := *shell32.ExtractAssociatedIconA()*
malicious := *msvbvm.*
Benign := *otherwise*

FIG. 6

P(" windows" \benign) = 45/47
P(" windows" \malicious) = 2/47
P(".COM" \benign)* = 1/12
P(".COM" \malicious)* = 11/12

FIG. 7

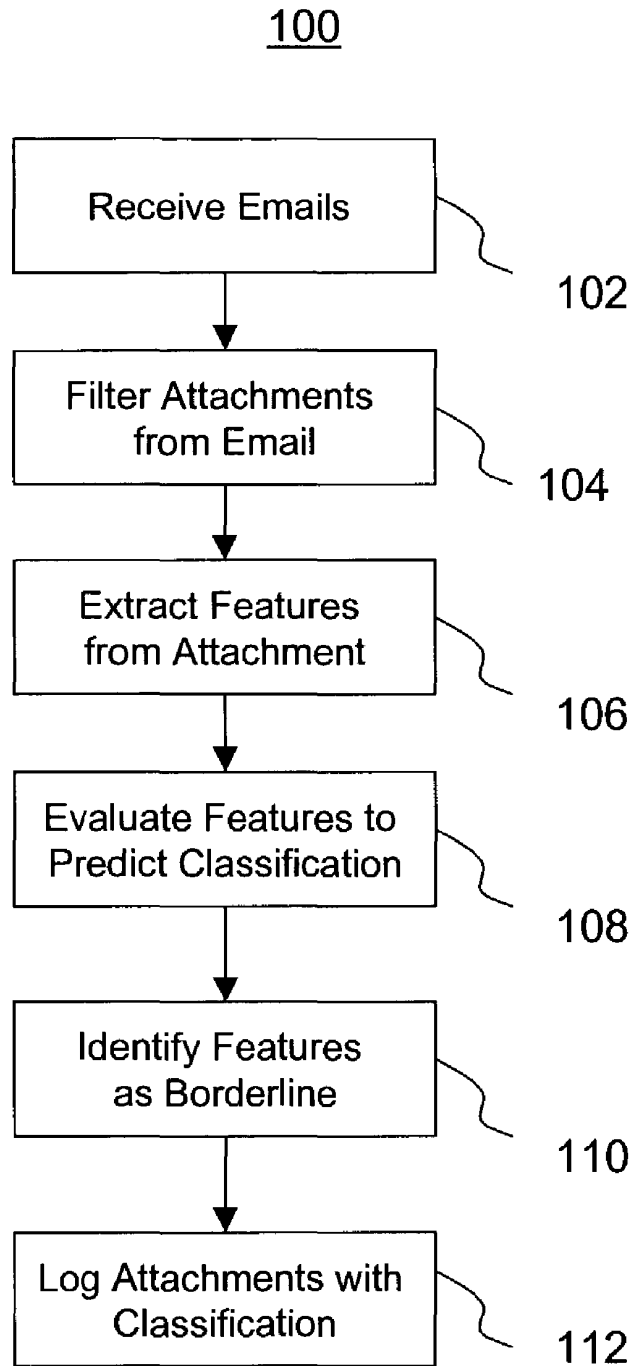


FIG. 8

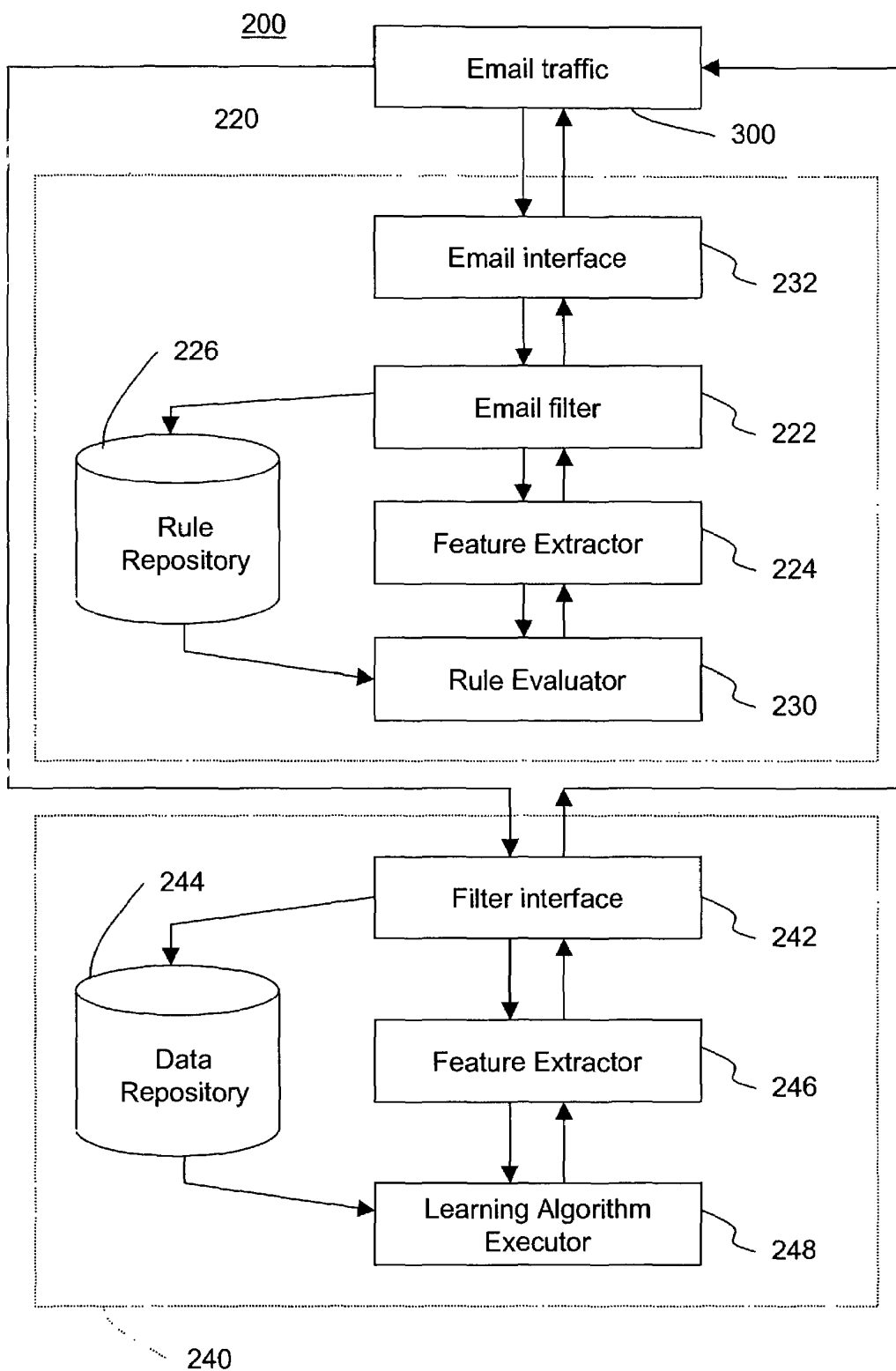


FIG. 9

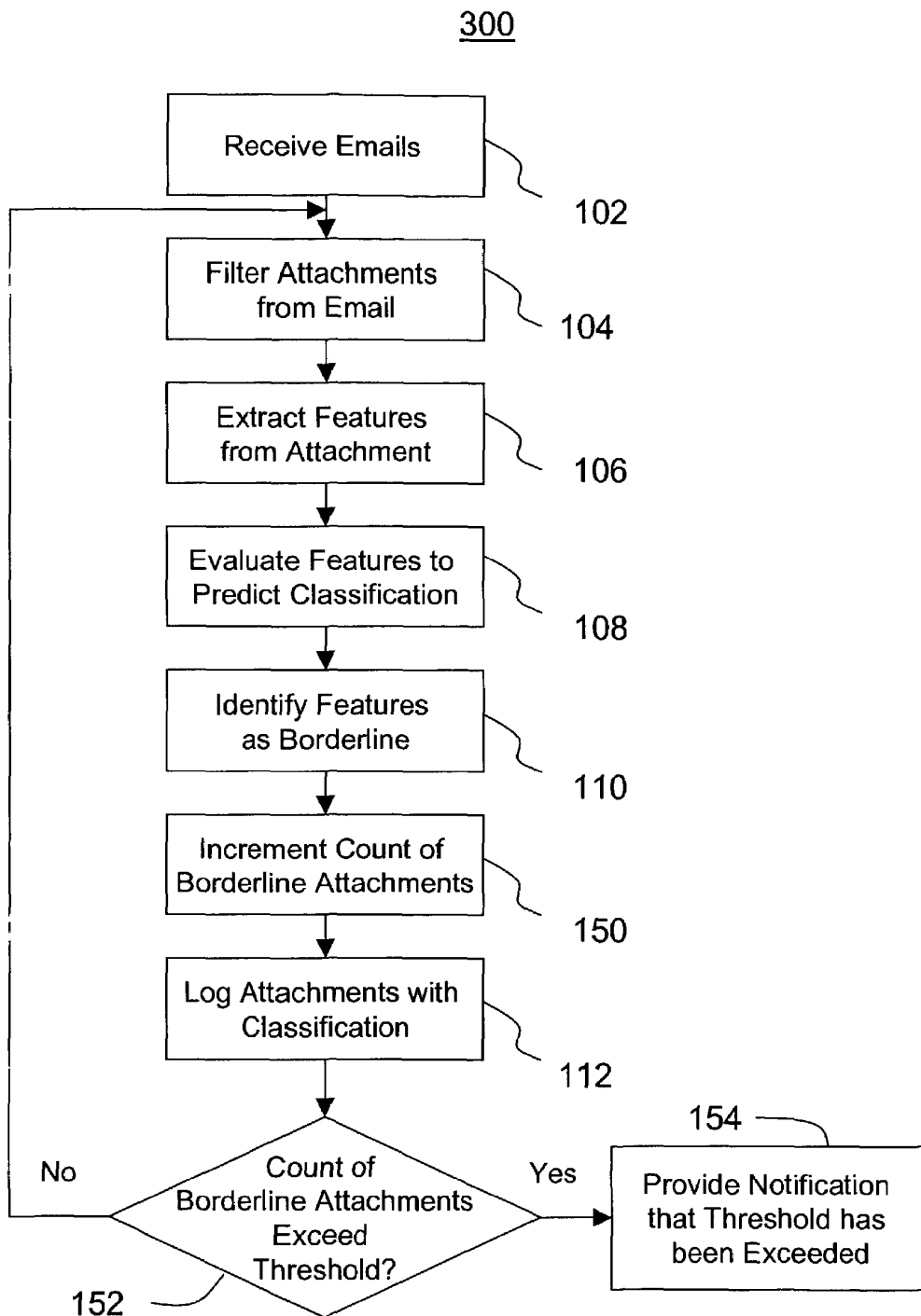


FIG. 10

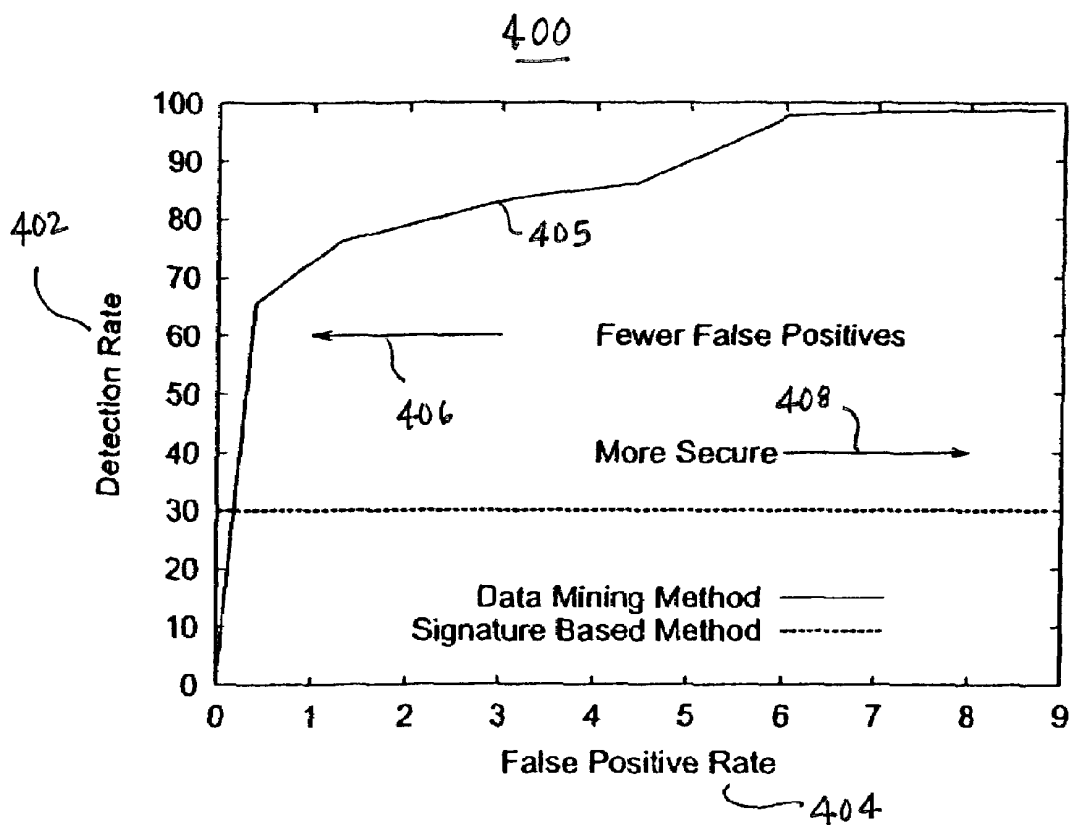


FIG. 11

US 7,487,544 B2

1

SYSTEM AND METHODS FOR DETECTION OF NEW MALICIOUS EXECUTABLES

CLAIM FOR PRIORITY TO RELATED APPLICATIONS

This application claims the benefit of U.S. Provisional Patent Application Ser. Nos. 60/308,622, filed on Jul. 30, 2001, entitled "Data Mining Methods for Detection of New Malicious Executables" and 60/308,623, filed on Jul. 30, 2001, entitled "Malicious Email Filter," which are hereby incorporated by reference in their entirety herein.

STATEMENT OF GOVERNMENT RIGHT

The present invention was made in part with support from the United States Defense Advanced Research Projects Agency (DARPA) grant nos. FAS-526617 and SRTSC-CU019-7950-1. Accordingly, the United States Government may have certain rights to this invention.

COPYRIGHT NOTICE

A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by any one of the patent disclosure, as it appears in the Patent and Trademark Office patent files or records, but otherwise reserves all copyright rights whatsoever.

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates to systems and methods for detecting malicious executable programs, and more particularly to the use of data mining techniques to detect such malicious executables in email attachments.

2. Background

A malicious executable is a program that performs a malicious function, such as compromising a system's security, damaging a system or obtaining sensitive information without the user's permission. One serious security risk is the propagation of these malicious executables through e-mail attachments. Malicious executables are used as attacks for many types of intrusions. For example, there have been some high profile incidents with malicious email attachments such as the ILOVEYOU virus and its clones. These malicious attachments are capable of causing significant damage in a short time.

Current virus scanner technology has two parts: a signature-based detector and a heuristic classifier that detects new viruses. The classic signature-based detection algorithm relies on signatures (unique telltale strings) of known malicious executables to generate detection models. Signature-based methods create a unique tag for each malicious program so that future examples of it can be correctly classified with a small error rate. These methods do not generalize well to detect new malicious binaries because they are created to give a false positive rate as close to zero as possible. Whenever a detection method generalizes to new instances, the tradeoff is for a higher false positive rate.

Unfortunately, traditional signature-based methods may not detect a new malicious executable. In an attempt to solve this problem, the anti-virus industry generates heuristic classifiers by hand. This process can be even more costly than generating signatures, so finding an automatic method to generate classifiers has been the subject of research in the

2

anti-virus community. To solve this problem, different IBM researchers applied Artificial Neural Networks (ANNs) to the problem of detecting boot sector malicious binaries. (The method of detection is disclosed in G. Tesauro et al., "Neural Networks for Computer Virus Recognition, *IEE Expert*, 11(4): 5-6, August 1996, which is incorporated by reference in its entirety herein.) An ANN is a classifier that models neural networks explored in human cognition. Because of the limitations of the implementation of their classifier, they were unable to analyze anything other than small boot sector viruses which comprise about 5% of all malicious binaries.

Using an ANN classifier with all bytes from the boot sector malicious executables as input, IBM researchers were able to identify 80-85% of unknown boot sector malicious executables successfully with a low false positive rate (<1%). They were unable to find a way to apply ANNs to the other 95% of computer malicious binaries.

In similar work, Arnold and Tesauro applied the same techniques to Win32 binaries, but because of limitations of the ANN classifier they were unable to have the comparable accuracy over new Win32 binaries. (This technique is described in Arnold et al., "Automatically Generated Win 32 Heuristic Virus Detection," *Proceedings of the 2000 International Virus Bulletin Conference*, 2000, which is incorporated by reference in its entirety herein.)

The methods described above have the shortcoming that they are not applicable to the entire set of malicious executables, but rather only boot-sector viruses, or only Win32 binaries.

The technique is similar to data mining techniques that have already been applied to Intrusion Detection Systems by Lee et al. Their methods were applied to system calls and network data to learn how to detect new intrusions. They reported good detection rates as a result of applying data mining to the problem of IDS. A similar framework is applied to the problem of detecting new malicious executables. (The techniques are described in W. Lee et al., "Learning Patterns From UNIX Processes Execution Traces for Intrusion Detection, *AAAI Workshop in AI Approaches to Fraud Detection and Risk Management*, 1997, pages 50-56, and W. Lee et al., "A Data Mining Framework for Building Intrusion Detection Models," *IEEE Symposium on Security and Privacy*, 1999, both of which are incorporated by reference in their entirety herein.)

Procmail is a mail processing utility which runs under UNIX, and which filters email; and sorts incoming email according to sender, subject line, length of message, keywords in the message, etc. Procmail's pre-existent filter provides the capability of detecting active-content HTML tags to protect users who read their mail from a web browser or HTML-enabled mail client. Also, if the attachment is labeled as malicious, the system "mangles" the attachment name to prevent the mail client from automatically executing the attachment. It also has built in security filters such as long filenames in attachments, and long MIME headers, which may crash or allow exploits of some clients.

However, this filter lacks the ability to automatically update its list of known malicious executables, which may leave the system vulnerable to attacks by new and unknown viruses. Furthermore, its evaluation of an attachment is based solely on the name of the executable and not the contents of the attachment itself.

Accordingly, there exists a need in the art for a technique which is not limited to particular types of files, such as boot-sector viruses, or only Win32 binaries, and which provides the ability to detect new, previously unseen files.

US 7,487,544 B2

3

SUMMARY

An object of the present invention is to provide a technique for predicting a classification of an executable file as malicious or benign which is not dependent upon the format of the executable.

Another object of the present invention is to provide a data mining technique which examines the entire file, rather than a portion of the file, such as a header, to classify the executable as malicious or benign.

A further object of the present invention is to provide an email filter which can detect executables that are borderline, i.e., executables having features indicative of both malicious and benign executables, which may be detrimental to the model if misclassified.

These and other objects of the invention, which will become apparent with reference to the disclosure herein, are accomplished by a system and methods for classifying an executable attachment in an email received by an email processing application or program, which includes filtering the executable attachment from said email. The email processing application may be executed at an email server or a client or host email application. A byte sequence feature is subsequently extracted from the executable attachment. The executable attachment is classified by comparing said byte sequence feature of the executable attachment with a classification rule set derived from byte sequence features of a set of executables having a predetermined class in a set of classes.

According to a preferred embodiment, extracting the byte sequence feature from said executable attachment comprises extracting static properties of the executable attachment, which are properties that do not require the executable to be run in order to discern. Extracting the byte sequence feature from the executable attachment may comprise converting the executable attachment from binary format to hexadecimal format. According to another embodiment, extracting the byte sequence features from the executable attachment may comprise creating a byte string representative of resources referenced by said executable attachment.

Advantageously, classifying the executable attachment may comprise predicting the classification of the executable attachment as one class in a set of classes consisting of malicious and benign. The set of classes may also include a borderline class. Classifying the executable attachment may comprise determining a probability or likelihood that the executable attachment is a member of each class in said set of classes based on said byte sequence feature. In one embodiment, this probability is determined by use of a Naive Bayes algorithm. In another embodiment, the probability may be determined by use of a Multi-Naive Bayes algorithm. The determination of the probability may be divided into a plurality of processing steps. These processing steps may then be performed in parallel. The executable attachment is classified as malicious if the probability that the executable attachment is malicious is greater than said probability that the executable attachment is benign. The executable attachment is classified as benign if the probability that the executable attachment is benign is greater than said probability that said executable attachment is malicious. The executable attachment is classified as borderline if a difference between the probability the executable attachment is benign and the probability the executable attachment is malicious is within a predetermined threshold.

A further step in accordance with the method may include logging the class of the executable attachment. The step of logging the class of the executable attachment may further

4

include incrementing a count of the executable attachments classified as borderline. If the count of executable attachments classified as borderline exceeds a predetermined threshold, the system may provide a notification that the threshold has been exceeded.

In accordance with the invention, the objects as described above have been met, and the need in the art for a technique which can analyze previously unseen malicious executables, without regard to the type of file, has been satisfied.

BRIEF DESCRIPTION OF THE DRAWINGS

Further objects, features and advantages of the invention will become apparent from the following detailed description taken in conjunction with the accompanying figures showing illustrative embodiments of the invention, in which:

FIG. 1 is a flow chart illustrating an overview of a method of detection model generation in accordance with the present invention.

FIGS. 2-4 illustrate a several approaches to binary profiling.

FIG. 5 illustrates sample classification rules determined from the features represented in FIG. 3.

FIG. 6 illustrates sample classification rules found by a RIPPER algorithm.

FIG. 7 illustrates sample classification rules found by a Naive Bayes algorithm.

FIG. 8 is a flow chart illustrating a method of detecting malicious executables in accordance with the present invention.

FIG. 9 is a simplified diagram illustrating the architecture of the malicious email detector and model generator in accordance with the present invention.

FIG. 10 is a flow chart, similar to FIG. 8, illustrating another method of detecting malicious executables in accordance with the present invention.

FIG. 11 is a plot illustrating the interactive effect of false positive rate and detection rate on the performance of the detection model or classifier in accordance with the present invention.

Throughout the figures, the same reference numerals and characters, unless otherwise stated, are used to denote like features, elements, components or portions of the illustrated embodiments. Moreover, while the subject invention will now be described in detail with reference to the figures, it is done so in connection with the illustrative embodiments. It is intended that changes and modifications can be made to the described embodiments without departing from the true scope and spirit of the subject invention as defined by the appended claims.

DETAILED DESCRIPTION OF EXEMPLARY EMBODIMENTS

This invention will be further understood in view of the following detailed description.

An exemplary system and methods for detecting malicious email attachments was implemented in UNIX with respect to Sendmail (a message transfer agent (MTA) which ensures messages get from source message servers to destination message servers for recipients to access their email, as produced by Sendmail, Inc. or Emeryville, Calif.), using Procmail (a publicly available program that processes e-mail messages received by the server, as further described in Stephen R. van den Berg and Philip Guenther, "Procmail," online publication as viewed on <http://www.procmail.org>, 2001). This system and methods uses data mining methods in order

US 7,487,544 B2

5

to create the detection model. The data mining methods are used to create classifiers to detect the malicious executables. A classifier is a classification rule set, or detection model, generated by the data mining algorithm that was trained over, i.e., derived from, a given set of training data.

In accordance with the exemplary embodiment, a data mining-based filter integrates with Procmail's pre-existent security filter to detect malicious executable attachments. The filter uses a scoring system based on a data mining classifier to determine whether or not an attachment may be malicious. If an attachment's score is above a certain threshold it is considered malicious. The data mining classifier provides the ability to detect both the set of known malicious executables and a set of previously unseen, but similar malicious executables.

A flowchart illustrating the process 10 of creating of the classification rule set is illustrated in FIG. 1. An early stage in the process is to assemble the dataset (step 12) which will be used for training, and for optionally testing the detection model. In the exemplary embodiment, this step included gathering a large set of executables 14 from public sources. In addition, each example program in the data set is a Windows or MS-DOS format executable, although the framework is applicable to other formats. In the exemplary embodiment, the programs were gathered either from FTP sites, or personal computers in the Data Mining Lab at Columbia University. A total of 4,031 programs were used.

In a subsequent stage, each data item, or executable, is labeled by class (step 16). The learning problem in the exemplary embodiment is defined with two classes, e.g., malicious and benign. As discussed above, a malicious executable is defined to be a program that performs a malicious function, such as compromising a system's security, damaging a system, or obtaining sensitive information without the user's permission. A benign program does not perform such malicious functions. Thus, the data set was divided into two groups: (1) malicious and (2) benign executables. In order to train the classification rule set, the classes of the executables must be known in advance. Of the 4,031 programs used in the data set, 3,301 were malicious executables and 1,000 were benign executables. The malicious executables consisted of viruses, Trojans, and cracker/network tools. There were no duplicate programs in the data set. To standardize the data-set, an updated McAfee's virus scanner, produced by McAfee.com Corporation of Sunnyvale, Calif., was used to label the programs as either malicious or benign executables. All labels were assumed to be correct for purposes of the analysis.

Another step, which may be performed concurrently with or subsequent to the above step, is to divide the dataset into two subsets which include a training set and a test set (step 18). The data mining algorithms use the training set to generate the classification rule sets. After training, a test set may be used to test the accuracy of the classifiers on a set of unseen examples. It is understood that testing the detection model is an optional step to determine the accuracy of the detection model, and, as such, may be omitted from the process.

The next step of the method is to extract features from each executable (Step 20). Features in a data mining framework are defined as properties extracted from each example program in the data set, e.g., byte sequences, that a classifier uses to generate detection models. (Signatures, as distinguished from features, typically refer to a specific feature value, while a feature is a property or attribute of data (such as "byte sequence feature") which may take on a set of values. Signature based methods are methods that inspect and test data to determine whether a specific feature value is present in that data, and then classify or alarm accordingly.) In the present

6

invention, the presence of specific feature values is used by the learning algorithms to calculate a probability or likelihood of classification of the data. The features which are extracted in the exemplary embodiment are static properties, which are properties that do not require executing the binary in order to be detected or extracted.

In the exemplary embodiment, hexdump was used in the feature extraction step. Hexdump, as is known in the art (Peter Miller, "Hexdump," on line publication 2000, <http://gd.tuwien.ac.at/softeng/Aegis/hexdump.html> which is incorporated by reference in its entirety herein), is an open source tool that transforms binary files into hexadecimal files. The byte sequence feature is informative because it represents the machine code in an executable. After the "hexdumps" are created, features are produced in the form illustrated in FIG. 2 in which each line represents a short sequence of machine code instructions. In the analysis, a guiding assumption is made that similar instructions were present in malicious executables that differentiated them from benign programs, and the class of benign programs had similar byte code that differentiated them from the malicious executables. Each byte sequence in the program is used as a feature.

Many additional methods of feature extraction are also useful to carry out step 20, above, and are described herein. For example, octale encoding may be used rather than hexadecimal encoding. According to another approach to feature extraction is to extract resource information from the binary that provides insight to its behavior, which is also referred to herein as "binary profiling." According to this approach, a subset of the data may be examined which is in Portable Executable (PE) format (which is described in "Portable Executable Format," online publication as viewed on, <http://support.microsoft.com/support/kb/articles/Q121/4/60.asp>, 1999, which is incorporated by reference in its entirety herein.) For instance, an executable in a standard Windows user interface may normally call the User Interfaces Dynamically Linked Library (USER32.DLL). This approach assumes that if an executable being evaluated does not call USER32.DLL, then the program does not have the standard Windows user interface. To extract resource information from Windows executables, GNU's Bin-Utils may be used (as described in "GNU Binutils Cygwin, online publication as viewed on <http://sourceware.cygwin.com/cygwin>, 1999, which is incorporated by reference in its entirety herein). GNU's Bin-Utils suite of tools can analyze PE binaries within Windows. In PE, or Common Object File Format (COFF), program headers are composed of a COFF header, an Optional header, at MS-DOS stub, and a file signature. All of the information about the binary is obtained from the program header without executing the unknown program but by examining the static properties of the binary, using libbfd, which is a library within Bin-Utils, to extract information in object format. Object format for a PE binary gives the file size, the names of DLLs, and the names of function calls within those DLLs and Relocation Tables. From the object format, it is possible to extract a set of features to compose a feature vector, or string, for each binary representative of resources referenced by the binary.

Three types of features may be analyzed to determine how resources affect a binary's behavior:

1. The list of DLLs used by the binary
2. The list of DLL function calls made by the binary
3. The number of different function calls within each DLL

A first approach to binary profiling used the DLLs loaded by the binary as features. Data can be modeled by extracting a feature or a set of features, and each set of features may be represented as a vector of feature values. The feature vector

7

comprised of a number of boolean values, e.g., 30, representing whether or not a binary used a DLL. Typically, not every DLL was used in all of the binaries, but a majority of the binaries called the same resource. For example, almost every binary called GDI32.DLL, which is the Windows NT Graphics Device Interface and is a core component of WinNT. The example vector given in FIG. 3 is composed of at least two unused resources: ADVAPI32.DLL, the Advanced Windows API, and WSOCK32.DLL, the Windows Sockets API. It also uses at least two resources: AVI-CAP32.DLL, the AVI capture API, and WINMM.DLL, the Windows Multimedia API.

A second approach to binary profiling uses DLLs and their function calls as features. This approach is similar to the first, described above, but with added function call information. The feature vector is composed of a greater number, e.g., 2,229, of boolean values. Because some of the DLL's had the same function names it was important to record which DLL the function came from. The example vector given in FIG. 4 is composed of at least four resources. Two functions were called in AD-VAP132.DLL: AdjustTokenPrivileges() and GetFileSecurityA(), and two functions were called in WSOCK32.DLL: recv() and send().

A third approach to binary profiling counts the number of different function calls used within each DLL, and uses such counts as features. The feature vector included several, e.g., 30, integer values. This profile provides an approximate measure of how heavily a DLL is used within a specific binary. This is a macro-resource usage model because the number of calls to each resource is counted instead of detailing referenced functions. For example, if a program only called the recv() and send() functions of WSOCK32.DLL, then the count would be 2. It should be noted that this third approach does not count the number of times those functions might have been called. The example vector given in FIG. 5 describes an example that calls two functions in ADVAPI32.DLL, ten functions in AVICAP32.DLL, eight functions in WINMM.DLL, and two functions from WSOCK32.DLL.

Another method useful for feature extraction (step 20) does not require PE format for the executables, and therefore is applicable to Non-PE executables. Headers in PE format are in plain text, which allows extraction of the same information from the PE executables by extracting the plain text headers. Non-PE executables also have strings encoded in them. This information is used to classify the entire data set, rather than being limited only to the subset of data including libBFD, described above. To extract features from the data set according to this third method, the GNU strings program was used. The strings program extracts consecutive printable characters from any file. Typically there are many printable strings in binary files. Some common strings found in the dataset are illustrated in Table 1.

TABLE 1

kernel	microsoft	windows	getmodulehandlea
getversion	getstartupinfoa	win	getmodulefilenamea
messageboxa	closehandle	null	dispatchmessagea
library	getprocaddress	advapi	getlasterror
loadlibrarya	exitprocess	heap	getcommandlinea
reloc	createfilea	writefile	setfilepointer
application	showwindow	time	regclosekey

Through testing it was observed that similar strings were present in malicious executables that distinguished them from benign programs, and similar strings in benign programs that distinguished them from malicious executables.

8

According to this technique, each string in the binary was used as a feature for the classifier.

Once the features were extracted using hexdump, or any other feature extraction method, such as those described herein, a classifier was trained to label a program as malicious or benign (Step 22). The classifier computes the probability or likelihood that a program is a member of a certain classification given the features or byte strings that are contained in that program. (Throughout the description herein, the term "probability" will generally refer to probability or likelihood, except where specified.)

In the exemplary embodiment, the classifier was a Naive Bayes classifier that was incorporated into Procmail, as will be described in greater detail herein. A Naive Bayes classifier is one exemplary machine learning algorithm that computes a model of a set of labeled training data and subsequently may use that model to predict the classification of other data. Its output is a likelihood (based on mathematical probability theory) associated with each classification possible for the other data. The Naive Bayes algorithm computes the likelihood that a program is a member each classification, e.g., malicious and benign, given the features or byte strings that are contained in that program. For instance, if a program contained a significant number of malicious byte sequences and a few or no benign sequences, then it labels that binary as malicious. Likewise, a binary that was composed of many benign features and a smaller number of malicious features is labeled benign by the system. In accordance with the invention, the assumption was made that there were similar byte sequences in malicious executables that differentiated them from benign programs, and the class of benign programs had similar sequences that differentiated them from the malicious executables.

In particular, the Naive Bayes algorithm first computes (a) the probability that a given feature is malicious and (b) the probability that the feature is benign, by computing statistics on the set of training data. Then to predict whether a binary, or collection of hex strings, was malicious or benign, those probabilities were computed for each hex string in the binary, and then the Naive Bayes independence assumption was used. The independence assumption was applied in order to efficiently compute the probability that a binary was malicious and the probability that the binary was benign.

Specifically, the Naive Bayes algorithm computes the class C of a program, given that the program contains a set of features F. (The Naive Bayes algorithm is described, for example, in T. Mitchell, "Naive Bayes Classifier," *Machine Learning*, McGraw-Hill, 1997, pp. 177-180, which is incorporated by reference in its entirety herein.) The term C is defined as a random variable over the set of classes: benign and malicious executables. That is, the classifier computes P(C|F), the probability that a program is in a certain class C given the program contains the set of features F. According to the Bayes rule, the probability is expressed in equation (1):

$$P(C|F) = \frac{P(F|C) * P(C)}{P(F)} \tag{1}$$

To use the Naive Bayes rule, it is assumed that the features occur independently from one another. If a program F include the features F₁, F₂, F₃, . . . , F_n, then equation (1) may be re-written as equation (2). (In this description, subscripted features F_x refers to a set of code strings.)

US 7,487,544 B2

9

$$P(C|F) = \frac{\prod_{i=1}^n P(F_i|C) * P(C)}{\prod_{j=1}^n P(F_j)} \tag{2}$$

Each $P(F_i|C)$ is the frequency that feature string F_i occurs in a program of class C . $P(C)$ is the proportion of the class C in the entire set of programs.

The output of the classifier is the highest probability class for a given set of strings. Since the denominator of equation (1) is the same for all classes, the maximum class is taken over all classes C of the probability of each class computed in (2) to get equation (3):

$$\text{Most Likely Class} = \max_c \left(P(C) \prod_{i=1}^n P(F_i|C) \right) \tag{3}$$

In equation (3), the term \max_c denotes the function that returns the class with the highest probability. "Most Likely Class" is the class in C with the highest probability and hence the most likely classification of the example with features F .

To train the classifier, a record was made for how many programs in each class contained each unique feature. This information was used to classify a new program into an appropriate class. Feature extraction, as described above, was used to determine the features contained in the program. Then, equation (3) was applied to compute the most likely class for the program.

The Naive Bayes method is a highly effective technique, but also requires significant amounts of main memory, such as RAM, e.g., greater than 1 gigabyte, to generate a detection model when the data or the set of features it analyzes is very large. To make the algorithm more efficient, the problem may be divided into smaller pieces that would fit in memory and generate a classifier for each of the subproblems. For example, the subproblem was to classify based on 16 subsets of the data organized according to the first letter of the hex string. This data mining algorithm is referred to as "Multi-Naive Bayes." This algorithm was essentially a collection of Naive Bayes algorithms that voted on an overall classification for an example. The Multi-Naive Bayes calculations are advantageously executed in a parallel and distributed computing system for increased speed.

According to this approach, several Naive Bayes classifiers may be trained so that all hex strings are trained on. For example, one classifier is trained on all hex strings starting with an "A", and another on all hex strings starting with an "0". This is done 16 times and then a voting algorithm is used to combine their outputs. Each Naive Bayes algorithm classified the examples in the test set as malicious or benign, and this counted as a vote. The votes are combined by the Multi-Naive Bayes algorithm to output a final classification for all the Naive Bayes.

According to the exemplary embodiment, the data may be divided evenly into several sets, e.g. six sets, by putting each i th line in the binary into the $(i \bmod n)$ th set where n is the number of sets. For each set, a Naive Bayes classifier is trained. The prediction for a binary is the product of the predictions of the n classifiers. In the exemplary embodiment, 6 classifiers ($n=6$) were used.

10

More formally, the Multi-Naive Bayes promotes a vote of confidence between all of the underlying Naive Bayes classifiers. Each classifier determines a probability of a class C given a set of bytes F which the Multi-Naive Bayes uses to generate a probability for class C given features F over all the classifiers.

The likelihood of a class C given feature F and the probabilities learned by each classifier NaiveBayes, are determined. In equation (4) the likelihood $L_{NB}(C|F)$ of class C given a set of feature F was computed:

$$L_{NB}(C|F) = \prod_{i=1}^{|NB|} P_{NB_i}(C|F) / P_{NB_i}(C) \tag{4}$$

where NB_i is a Naive Bayes classifier and NB is the set of all combined Naive Bayes classifiers (in this case 6). $P_{NB_i}(C|F)$ (generated from equation (2)) is the probability for class C computed by the classifier NaiveBayes, given F divided by the probability of class C computed by NaiveBayes_{*i*}. Each $P_{NB_i}(C|F)$ was divided by $P_{NB_i}(C)$ to remove the redundant probabilities. All the terms were multiplied together to compute $L_{NB}(C|F)$, the final likelihood of C given F . $|NB|$ is the size of the set NB such that $NB_i \in NB$.

The output of the multi-classifier given a set of bytes F is the class of highest probability over the classes given $L_{NB}(C|F)$ and $P_{NB}(C)$ the prior probability of a given class, which is represented by equation (5), below.

$$\text{Most Likely Class} = \max_c (P_{NB}(C) * L_{NB}(C|F)) \tag{5}$$

Most Likely Class is the class in C with the highest probability hence the most likely classification of the example with features F , and \max_c returns the class with the highest likelihood.

Additional embodiments of classifiers are described herein, which are also useful to classify an executable as benign or malicious. Alternatively, inductive rule learners may be used as classifiers. Another algorithm, RIPPER, is an inductive rule learner (RIPPER is described in W. Cohen, "Learning Trees and Rules with Set-Valued Features," *American Association for Artificial Intelligence*, 1996, which is incorporated by reference in its entirety herein). This algorithm generates a detection model composed of resource rules that was built to detect future examples of malicious executables. RIPPER is a rule-based learner that builds a set of rules that identify the classes while minimizing the amount of error. The error is defined by the number of training examples misclassified by the rules. This algorithm may be used with libBFD information as features, which were described above.

As is known in the art, an inductive algorithm learns what a malicious executable is, given a set of training examples. Another useful algorithm for building a set of rules is Find-S. Find-S finds the most specific hypothesis that is consistent with the training examples. For a positive training example the algorithm replaces any feature in the hypothesis that is inconsistent with the training example with a more general feature. For example, four features seen in Table 2 are:

TABLE 2

- | | |
|----|---|
| 1. | "Does it have a GUI?" |
| 2. | "Does it perform a malicious function?" |
| 3. | "Does it compromise system security?" |
| 4. | "Does it delete files?" |

US 7,487,544 B2

11

TABLE 2-continued

and finally the classification label "Is it malicious?"					
	Has a GUI?	Malicious Function?	Compromise Security?	Deletes Files?	Is it malicious?
1	yes	yes	yes	no	yes
2	no	yes	yes	yes	yes
3	yes	no	no	yes	no
4	yes	yes	yes	yes	yes

The defining property of any inductive learner is that no a priori assumptions have been made regarding the final concept. The inductive learning algorithm makes as its primary assumption that the data trained over is similar in some way to the unseen data.

A hypothesis generated by an inductive learning algorithm for this learning problem has four features. Each feature will have one of these values:

1. \top , truth, indicating any value is acceptable in this position,
2. a value, either yes, or no, is needed in this position, or
3. a \perp , falsity, indicating that no value is acceptable for this position.

For example, the hypothesis $\langle \top, \top, \top, \top \rangle$ and the hypothesis (yes, yes, yes, no) would make the first example in Table 2 true. $\langle \top, \top, \top, \top \rangle$ would make any feature set true and $\langle \text{yes, yes, yes, no} \rangle$ is the set of features for example one.

Of all the hypotheses, values 1 is more general than 2, and 2 is more general than 3. For a negative example, the algorithm does nothing. Positive examples in this problem are defined to be the malicious executables and negative examples are the benign programs.

The initial hypothesis that Find-S starts with is $\langle \perp, \perp, \perp, \perp \rangle$. This hypothesis is the most specific because it is true over the fewest possible examples, none. Examining the first positive example in Table 2, $\langle \text{yes, yes, yes, no} \rangle$, the algorithm chooses the next most specific hypothesis $\langle \text{yes, yes, yes, no} \rangle$. The next positive example, $\langle \text{no, no, no, yes} \rangle$, is inconsistent with the hypothesis in its first and fourth attribute ("Does it have a GUI?" and "Does it delete files?") and those attributes in the hypothesis get replaced with the next most general attribute, \top .

The resulting hypothesis after two positive examples is $\langle \top, \text{yes, yes, } \top \rangle$. The algorithm skips the third example, a negative example, and finds that this hypothesis is consistent with the final example in Table 2. The final rule for the training data listed in Table 2 is $\langle \top, \text{yes, yes, } \top \rangle$. The rule states that the attributes of a malicious executable, based on training data, are that it has a malicious function and compromises system security. This is consistent with the definition of a malicious executable stated above. Thus, it does not matter in this example if a malicious executable deletes files, or if it has a GUI or not.

RIPPER looks at both positive and negative examples to generate a set of hypotheses that more closely approximate the target concept while Find-S generates one hypothesis that approximates the target concept.

Each of the data mining algorithms generated its own classification rule set **24** to evaluate new examples. The classification rule sets are incorporated in the filter to detect malicious executables, as will be described below in the exemplary embodiment. For purposes herein, a classification rule set is considered to have the standard meaning in data mining terminology, i.e., a set of hypotheses that predict the classification, e.g., malicious or benign, of an example, i.e.,

12

an executable, given the existence of certain features. The Naive Bayes rules take the form of $P(F|C)$, the probability of an feature F given a class C. The probability for a string occurring in a class is the total number of times it occurred in that class's training set divided by the total number of times that the string occurred over the entire training set. An example of such hypotheses are illustrated in FIG. 7. Here, the string "windows" was predicted to more likely occur in a benign program and string "*.COM" was more than likely in a malicious executable program.

This approach compensates for those instances where a feature, e.g., a hex string, occurs in only one class in the training data. When this occurs, the probability is arbitrarily increased from $0/n$, where n is the number of occurrences, to $1/n$. For example, a string (e.g. "AAAA") may occur in only one set, e.g., in the malicious executables. The probability of "AAAA" occurring in any future benign example is predicted to be 0, but this is an incorrect assumption. If a program was written to print out "AAAA" it will always be tagged a malicious executable even if it has other strings in it that would have labeled it benign. In FIG. 6, the string "*.COM" does not occur in any benign programs so the probability of "*.COM" occurring in class benign is approximated to be $1/12$ instead of $0/11$. This approximates real world probability that any string could occur in both classes even if during training it was only seen in one class.

The rule sets generated by the Multi-Naive Bayes algorithm are the collection of the rules generated by each of the component Naive Bayes classifiers. For each classifier, there is a rule set such as the one in FIG. 6. The probabilities in the rules for the different classifiers may be different because the underlying data that each classifier is trained on is different. The prediction of the Multi-Naive Bayes algorithm is the product of the predictions of the underlying Naive Bayes classifiers.

RIPPER's rules were built to generalize over unseen examples so the rule set was more compact than the signature-based methods. For the data set that contained 3,301 malicious executables the RIPPER rule set contained the five rules in FIG. 6.

Here, a malicious executable was consistent with one of four hypotheses:

1. it did not call `user32.EndDialog()` but it did call `kernel32.EnumCalendarInfoA()`
2. it did not call `user32.LoadIconA()`, `kernel32.GetTempPathA()`, or any function in `advapi32.dll`
3. it called `shell32.FExtractAssociatedIconA()`,
4. it called any function in `msvbbm.dll`, the Microsoft Visual Basic Library

A binary is labeled benign if it is inconsistent with all of the malicious binary hypotheses in FIG. 6.

Each data mining algorithm generated its own rule set **24** to evaluate new examples. The detection models are stored for subsequent application to classify previously unseen examples, as will be described below. An optional next step is to test the classification rule set **24** against the test data (step **26**). This step is described in greater detail below.

The process **100** of detecting malicious emails in accordance with the invention is illustrated in FIG. 8. A first step is to receive the emails at the server (step **102**). In the exemplary embodiment, the mail server is Sendmail. Procmail is a publicly available program that processes e-mail messages received by the server and looks for particular information in the headers or body of the message, and takes action on what it finds.

Subsequently, the emails are filtered to extract attachments or other components from the email (step **104**). The execut-

US 7,487,544 B2

13

able attachments may then be saved to a file. In the exemplary embodiment, `html_trap.procmail`, a commonly-available routine, has been modified to include a call to a novel routine, `parser3`, which performs the functions of filtering attachments. The routine `parser3` includes a call to the routine `extractAttachments`, for example, which extracts executable attachments and other items of the email and saves them to a file, e.g., `$files_full_ref`, and also provides a string containing a directory of where the executable attachments are saved, e.g., `$dir`, and a reference to an array containing a list of the file names, e.g., `$files_ref`.

Features in the executable attachment are extracted (step 106), and those features are subsequently analyzed and used to classify the executable attachment as malicious or benign. In the exemplary embodiment, the routine `parser3` also includes a call to the routine `scanAttachments`, which in turn calls the routine `hexScan`, which performs the feature extraction of step 106. In particular, `hexScan` includes a function call to `hexdump`, a commonly-available routine which transforms the binary files in the attachment into a byte sequence of hexadecimal characters, as described above and illustrated in FIG. 2. The resulting hexadecimal string is saved as `"/tmp/$$hex,"` These strings of hexadecimal code are the "features" which are used in the classification, described below. This byte sequence is useful because it represents the machine code in an executable. In addition, this approach involves analyzing the entire binary, rather than portions such as headers, an approach which consequently provides a great deal of information about the executable. It is understood that the feature extraction step described herein is alternatively performed with a binary profiling method in another embodiment, as described above and illustrated in FIGS. 3-4, and with a GNU strings method, also described above and illustrated in Table 1. In these embodiments, the step of calling the routine `hexScan` in `scanAttachments` is replaced by calls to routines that perform the binary profiling or GNU strings analysis.

The features extracted from the attachment in step 106 are evaluated using the classification rule set as described above, and the attachment is classified as malicious or benign (step 108). In the exemplary embodiment, the routine `hexScan` subsequently calls the routine `senb`, which calculates "scores" associated with the attachments. (As will be described below, such scores are representative of whether a binary is malicious or benign.) The routine `senb` evaluates the features, e.g., the hexadecimal string `"/tmp/$$hex"` produced by `hexdump` against the rules in the Classification Rule Set, e.g., `"/etc/procmail/senb/aids_model.txt,"` and returns with a first score associated with the probability that the string is malicious and a second score associated with the probability that the string is benign. In order to obtain these scores, the routine `senb` invokes the routine `check_file`, which performs the Naive Bayes analysis on the features as described above in equation (1)-(5), and calculates scores associated with the probability that the program is malicious and benign. Where the Multi-Naive Bayes algorithm is used, the data is partitioned into components which are processed in parallel to increase processing speed. The routine `hexScan` then determines which of the scores is greater, e.g., malicious or benign. In other embodiments, a different classification algorithm may be implemented, such as a function call to the RIPPER algorithm which will evaluate the features extracted in step 106 to determine whether they are malicious or benign.

A further step may be to identify the programs as borderline (step 110). Borderline executables are defined herein as programs that have similar probabilities of being benign and malicious (e.g., 50% chance it is malicious, and 50% chance

14

it is benign; 60% chance malicious and 40% chance benign; 45% chance malicious and 55% chance benign, etc.). Due to the similar probabilities, borderline executables are likely to be mislabeled as either malicious or benign. Since borderline cases could potentially lower the detection and accuracy rates by being misclassified, it is desirable to identify these borderline cases, properly classify them as malicious or benign, and update the classification rule set to provide increased accuracy to the detection of malicious executables. The larger the data set that is used to generate models, then the more accurate the detection models will be. To execute this process, the system identifies programs as borderline using the criteria described below, and archives the borderline cases. At periodic intervals, the system sends the collection of these borderline cases to a central server, by the system administrator. Once at a central repository, such as data repository 244, these binaries can then be analyzed by experts to determine whether they are malicious or not, and subsequently included in the future versions of the detection models. Preferably, any binary that is determined to be a borderline case will be forwarded to the repository and wrapped with a warning as though it were a malicious attachment.

An exemplary metric to identify borderline cases, which may be implemented in `hexScan` or a similar routine is to define a borderline case to be a case where the difference between the probability, or score, that the program is malicious and the probability, or score, it is benign is below a threshold. This threshold may be set based on the policies of the host. For example, in a secure setting, the threshold could be set relatively high, e.g., 20%. In this case, all binaries that have a 60/40 split are labeled as borderline. In other words, binaries with a 40-60% chance (according to the model) of being malicious and 40-60% chance of being benign would be labeled borderline. This setting can be determined by the system administrator. An exemplary default setting of 51.25/48.75 may be used with a threshold of 2.5%, which was derived from testing.

The routine `scanAttachments` receives the output of `hexScan` which is a determination of whether the program is malicious or benign, and assigns the string a boolean "0" or "1." (Where the probabilities of being malicious and of being benign are similar, it may be labeled borderline, as discussed above.) Subsequently, `scanAttachments` invokes the routine `md5log` to associate a unique identifier for each attachment in by using the MD5 algorithm, (as described in R. Rivest, "The MD5 Message Digest Algorithm," Internet RFC1321, Paril 1992, which is incorporated by reference in its entirety herein.) The input to MD5 is the hexadecimal representation of the binary. These identifiers are then kept in a log along with other information such as whether the attachment was malicious, benign, or borderline and with what certainty the system made those predictions (Step 112).

The results of this analysis are sent from `parser3` to `html_trap.procmail`, which inserts warnings that the file may be malicious and may quarantine the attachment. The routine `html_trap.procmail` reintegrates filtered email back into normal email traffic.

An exemplary system 200 in accordance with the invention is illustrated in FIG. 9. The system 200 includes a malicious email detector 220 and model generator 240. The system may reside on the server of a computer or on a host or client of the computer system to receive emails before they are forwarded to users of the system.

The malicious email detector may include an email filter 222, a feature extractor 224, a rule repository 226, a rule evaluator 230, and an email interface 232. In the exemplary embodiment, the email filter 222 may include the routine

Parser3 which filters attachments from the emails as described above. Parser3 calls the routine extractAttachments, for example, which extracts attachments and other items of the email. The email filter 222 may also filter out updated classification rules sent by the model generator 240, and forward them to the rule repository 226.

The feature extractor 224 receives the executable attachments and extracts those byte sequence features which will be analyzed and used to classify the program as malicious or benign. In the exemplary embodiment, the routine scanAttachments calls the routine hexScan, which performs the function of the feature extractor 224. In particular, hexScan includes a function call to hexdump, which transforms the binary files into hexadecimal strings. The rule repository 226 may be a database which contains the classification rule set generated by a data mining model in a process such as that illustrated in FIG. 1. The rule evaluator 230 evaluates the byte sequence features extracted from the attachments by the feature extractor 224 using the classification rule set provided by the rule repository 226.

In the exemplary embodiment, the routine hexScan calls the routine senb, which performs the function of rule evaluator 230. The routine senb evaluates the byte sequence features, e.g., the hexadecimal string against the classification rule set in the rule repository 26, e.g., “/etc/procmail/senb/aids_model.txt,” and returns with a score that the string is malicious and a score that the string is benign. The rule evaluator 230 may also provide an indication that the string is borderline.

The results of this analysis may be sent to the email interface 232, which reintegrates filtered email back into normal email traffic 300, and which may send the model generator 240 (described below) each attachment to be analyzed further. If the program was considered malicious by the rule evaluator 230, the email interface 232 may add warnings to the email or quarantine the email. In the exemplary embodiment, the routine html-trap.procmail performs this function.

The classification rule set may require updates periodically. For example, after a number of borderline cases have been identified by the rule evaluator 230, it may be desirable to generate a new detection model, and subsequently distribute the updated models. This embodiment of the system 300, which is illustrated in FIG. 10, is substantially identical to system 100, with the differences noted herein. For example, the email filter 222 may maintain a running counter of the number of borderline executables identified (step 150 of FIG. 9). When a predetermined threshold is exceeded (proportional to the overall traffic of email received) (step 152 of FIG. 9), a notification may be sent that the threshold has been exceeded (step 154 of FIG. 9). Subsequently, the model generator 240 may be invoked to generate an updated classification rule set

A new classification rule set is generated at the model generator 240 by running the data mining algorithm on the new data set that contains the borderline cases along with their correct classification (as determined by expert analysis), and the existing training data set. As described herein, the data mining algorithm may be a Naive Bayes or a Multi-Naive Bayes algorithm, or any other appropriate algorithm for calculating the probability or likelihood that a feature is a member of a class. When the Multi-Naive Bayes analysis is used herein, the data is partitioned into several components and all the components may be processed in parallel to increase speed. This updated model may then be distributed to the malicious email detector 220. The filter interface 242 may receive copies of all attachments from the email interface 232. In the exemplary embodiment, the routine senb may perform

the function of the filter interface 242. The data repository 244 receives copies of attachments from the filter interface 42, and stores the attachments. In the exemplary embodiment, the attachments may be stored as a datafile.

The feature extractor 246 accesses attachments stored in the data repository 244, and then extracts features from the attachments. This function may be performed by invoking the hexdump routine, as described above. The learning algorithm executor 248 receives features from the feature extractor 246 and executes learning algorithms on the features extracted from the attachments to generate an updated classification rule set. In the exemplary embodiment, the routine senb calls the routine test_table, which in turn invokes test_class. Test_class invokes test_file, which performs the function of creating the updated classification rule set, including performing the Naive Bayes calculations, described above in equations (1)-(5).

In an exemplary embodiment, the filter interface 242 receives the classification model rule set from the learning algorithm executor 248 and transmits the classification model rule set to the malicious email detector 220, where it is used to update the classification rule set stored in the rule repository 226. According to the exemplary embodiment, portions of the classification model rule set that have changed may be distributed, rather than the entire classification model rule set, to improve efficiency. In order to avoid constantly sending a large model from the model generator 240 to the malicious email detector 220, the administrator is provided with the option of receiving this smaller file. Using the update algorithm, the older model can then be updated. The full model will also be available to provide additional options for the system administrator. Efficient update of the model is possible because the underlying representation of the models is probabilistic. Thus, the model is a count of the number of times that each byte string appears in a malicious program versus the number of times that it appears in a benign program. An update model can then be easily summed with the older model to create a new model. From these counts the algorithm computes the probability that an attachment is malicious in a method described above. In order to combine the models, the counts of the old model are summed with the new information.

As shown in Table 3, in model A, the old detection model, a byte string occurred 99 times in the malicious class, and one time in the benign class. In model B, the update model, the same byte string was found three times in the malicious class and four times in the benign class. The combination of models A and B would state that the byte string occurred 102 times in the malicious class and five times in the benign class. The combination of A and B would be the new detection model after the update.

TABLE 3

Model A (old)
The byte string occurred in 99 malicious executables The byte string occurred in 1 benign executable
Model B (new)
The byte string occurred in 3 malicious executables The byte string occurred in 4 benign executables.
Model C (update)
The byte string occurred in 102 malicious executables The byte string occurred in 5 benign executables.

To compare the results of the methods and system described herein with traditional methods, a prior art signa-

US 7,487,544 B2

17

ture-based method was implemented (step 26, of FIG. 1). First, the byte-sequences that were only found in the malicious executable class were calculated. These byte-sequences were then concatenated together to make a unique signature for each malicious executable example. Thus each malicious executable signature contained only byte-sequences found in the malicious executable class. To make the signature unique, the byte-sequences found in each example were concatenated together to form one signature. This was done because a byte-sequence that was only found in one class during training could possibly be found in the other class during testing, and lead to false positives when deployed.

The virus scanner that was used to label the data set (step 16, above) contained signatures for every malicious example in the data set, so it was necessary to implement a similar signature-based method. This was done to compare the two algorithms' accuracy in detecting new malicious executables. In the tests, the signature-based algorithm was only allowed to generate signatures for the same set of training data that the data mining method used. This allowed the two methods to be fairly compared. The comparison was made by testing the two methods on a set of binaries not contained in the training set.

To quantify the performance of the method described herein, statistics were computed on the performance of the data mining-based method, tables 4 and 5 are included herein which include counts for true positives, true negatives, false positives and false negatives. A true positive, TP, is a malicious example that is correctly classified as malicious, and a true negative, TN, is a benign example that is correctly classified as benign. A false positive, FP, is a benign program that has been mislabeled by an algorithm as malicious, while a false negative, FN, is a malicious executable that has been mis-classified as a benign program.

The overall accuracy of the algorithm is calculated as the number of programs the system classified correctly divided by the total number of binaries tested. The detection rate is the number of malicious binaries correctly classified divided by the total number of malicious programs tested.

The results were estimated over new executables by using 5-fold cross validation technique, as described in R. Kohavi, "A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection," *IJCAI*, 1995. Cross-validation, as is known in the art, is the standard method to estimate the performance of predictions over unseen data in Data Mining. For each set of binary profiles the data was partitioned into five equal size partitions. Four of the partitions were used for training a model and then evaluating that model on the remaining partition. Then the process was repeated five times leaving out a different partition for testing each time. This provided a measure of the method's accuracy on unseen data. The results of these five tests were averaged to obtain a measure of how the algorithm performs over the entire set.

To evaluate the algorithms over new executables, the algorithms generated their detection models over the set of training data and then tested their models over the set of test data. This was done five times in accordance with cross fold validation.

Tables 4 displays the results. The data mining algorithm had the highest detection rate, 97.76%, compared with the signature-based method's detection rate of 33.96%. Along with the higher detection rate the data mining method had a higher overall accuracy, 96.88% vs. 49.31%. The false positive rate of 6.01% though was higher than the signature-based method, 0%.

18

TABLE 4

Profile Type	Detection Rate	False Positive Rate	Overall Accuracy
Signature Method	33.96%	0%	49.31%
Data Mining Method	97.76%	6.01%	96.88%

FIG. 11 displays the plot 400 of the detection rate 402 vs. false positive rate 404 using Receiver Operation Characteristic curves, as described in K. H. Zou et al., "Smooth Non-Parametric ROC Curves for Continuous Diagnostic Tests," *Statistics in Medicine*, 1997. Receiver Operating Characteristic (ROC) curves are a way of visualizing the trade-offs between detection and false positive rates. In this instance, the ROC curve shows how the data mining method (illustrated in dashed line 405) can be configured for different environments. For a false positive rate less than or equal to 1% the detection rate would be greater than 70%, and for a false positive rate greater than 8% the detection rate would be greater than 99%. Thus, more secure settings would select a threshold setting associated with a point on the data mining line towards the right (indicated by arrow 408), and applications needing fewer false alarms should choose a point towards the left (indicated by arrow 406).

The performance of the models in detecting known executables was also evaluated. For this task, the algorithms generated detection models for the entire set of data. Their performance was then evaluated by testing the models on the same training set.

As shown in Table 5, both methods detected over 99% of known executables. The data mining algorithm detected 99.87% of the malicious examples and misclassified 2% of the benign binaries as malicious. However, the signatures for the binaries that the data mining algorithm misclassified were identified, and the algorithm can include those signatures in the detection model without lowering accuracy of the algorithm in detecting unknown binaries. After the signatures for the executables that were misclassified during training had been generated and included in the detection model, the data mining model had a 100% accuracy rate when tested on known executables.

TABLE 5

Profile Type	Detection Rate	False Positive Rate	Overall Accuracy
Signature Method	100%	0%	100%
Data Mining Method	99.87%	2%	99.44%

In order for the data mining algorithm to quickly generate the models, it is advantageous for all calculations to be done in memory. The algorithm consumed space in excess of a gigabyte of RAM. By splitting the data into smaller pieces, the algorithm was done in memory with no loss in accuracy. In addition, the calculations may be performed in parallel. The training of a classifier took 2 hours 59 minutes and 49 seconds running on Pentium III 600 Linux machine with 1 GB of RAM. The classifier took on average 2 minutes and 28 seconds for each of the 4,301 binaries in the data set. The amount of system resources taken for using a model are equivalent to the requirements for training a model. So on a Pentium III 600 Linux box with 1 GB of RAM it would take on average 2 minutes 28 seconds per attachment. Another advantageous of splitting the data into smaller partitions (in

US 7,487,544 B2

19

connection with the Multi-Naive Bayes analysis) is that the Naive Bayes algorithm is executed on each partition on parallel hardware, which reduces the total training time from 2 hours and 59 minutes, to 2 minutes and 28 seconds if each piece is concurrently executed.

It will be understood that the foregoing is only illustrative of the principles of the invention, and that various modifications can be made by those skilled in the art without departing from the scope and spirit of the invention.

What is claimed is:

1. A method for classifying an executable attachment in an email received at an email processing application of a computer system comprising:

- a) filtering said executable attachment from said email;
- b) extracting a byte sequence feature from said executable attachment; and
- c) classifying said executable attachment by comparing said byte sequence feature of said executable attachment with a classification rule set derived from byte sequence features of a set of executables having a predetermined class in a set of classes to determine the probability whether said executable attachment is malicious, wherein extracting said byte sequence features from said executable attachment comprises creating a byte string representative of resources referenced by said executable attachment.

2. The method as defined in claim 1, wherein extracting said byte sequence feature from said executable attachment comprises extracting static properties of said executable attachment.

3. The method as defined in claim 1, wherein extracting said byte sequence feature from said executable attachment comprises converting said executable attachment from binary format to hexadecimal format.

4. The method as defined in claim 1, wherein classifying said executable attachment comprises determining a probability that said executable attachment is a member of each class in a set of classes consisting of malicious and benign.

5. The method as defined in claim 1, further comprising updating the classification rule set based on executable attachments classified in said classifying.

6. A method for classifying an executable attachment in an email received at an email processing application of a computer system comprising:

- a) filtering said executable attachment from said email;
- b) extracting a byte sequence feature from said executable attachment; and
- c) classifying said executable attachment by comparing said byte sequence feature of said executable attachment with a classification rule set derived from byte sequence features of a set of executables having a predetermined class in a set of classes to determine a probability that said executable attachment is a member of each class in a set of classes consisting of malicious, benign, and borderline.

7. The method as defined in claim 6, wherein classifying said executable attachment comprises determining said probability that said executable attachment is a member of each class in said set of classes with a Naive Bayes algorithm.

8. The method as defined in claim 6, wherein classifying the executable attachment comprises determining said probability that said executable attachment is a member of a class in said set of classes with a Multi-Naive Bayes algorithm.

9. The method as defined in claim 8, which further comprises dividing said determining said probability into a plurality of processing steps and executing said processing steps in parallel.

20

10. The method as defined in claim 6, wherein classifying the executable attachment comprises classifying said executable attachment as malicious if said probability that said executable attachment is malicious is greater than said probability that said executable attachment is benign.

11. The method as defined in claim 6, wherein classifying the executable attachment comprises classifying said executable attachment as benign if said probability that said executable attachment is benign is greater than said probability that said executable attachment is malicious.

12. The method as defined in claim 6, wherein classifying the executable attachment comprises classifying said executable attachment as borderline if a difference between said probability that said executable attachment is benign and said probability that said executable attachment is malicious is within a predetermined threshold.

13. The method as defined in claim 6, which further comprises logging said class of said executable attachment classified in said step c).

14. The method as defined in claim 13, wherein logging said class of said executable attachment further comprising incrementing a count of said executable attachments classified as borderline.

15. The method defined in claim 14, which further comprises, if said count of executable attachments exceeds a predetermined threshold, providing a notification that said threshold has been exceeded.

16. A method for classifying an executable program comprising:

- a) training a classification rule set based on a predetermined set of known executable programs having a predetermined class and one or more byte sequence features by recording the number of known executable programs in each said predetermined class that has each of said byte sequence features;
- b) extracting a byte sequence feature from said executable program comprising converting said executable program from binary format to hexadecimal format, wherein extracting said byte sequence features from said executable attachment comprises create a byte string representative of resources referenced by said executable attachment; and
- c) determining the probability that the executable program is within each said predetermined class, based on said one or more byte sequence features in said executable and said classification rule set to determine whether said executable program is malicious.

17. The method as defined in claim 16, wherein extracting said byte sequence feature from said executable program comprises extracting static properties of said executable program.

18. The method as defined in claim 16, wherein determining the probability that the executable program is within each said predetermined class comprises determining the probability that the executable program is within said predetermined class in a set of classes consisting of malicious and benign.

19. The method as defined in claim 16, wherein determining said probability that the executable program is within each said predetermined class comprises determining said probability that the executable program is within each said predetermined class with a Naive Bayes algorithm.

20. The method as defined in claim 16, wherein determining said probability that the executable program is within each said predetermined class comprises determining said probability that the executable program is within each said predetermined class with a multi-Naive Bayes algorithm.

21

21. The method as defined in claim 16, wherein determining said probability that the executable program is within each said predetermined class comprises classifying said executable program as malicious if said probability that said executable program is malicious is greater than said probability that said executable program is benign.

22. The method as defined in claim 16, wherein determining said probability that the executable program is within each said predetermined class comprises classifying said executable program as benign if said probability that said executable program is benign is greater than said probability that said executable program is malicious.

23. The method as defined in claim 16, wherein determining said probability that the executable program is within each said predetermined class comprises classifying said executable program as borderline if a difference between said probability that said executable program is benign and said probability that said executable program is malicious is within a predetermined threshold.

24. The method as defined in claim 16, which further comprises logging said class of said executable determined in said step c).

25. The method as defined in claim 24, wherein logging said class of said executable further comprising incrementing a count of said executable classified as borderline.

26. The method defined in claim 25, which further comprises, if said count of executable exceeds a predetermined threshold, providing a notification that said threshold has been exceeded.

27. The method as defined in claim 16, further comprising updating the classification rule set based on executable attachments classified in said determining.

28. A system for classifying an executable attachment in an email received at a server of a computer system comprising:

- a) an email filter configured to filter said executable attachment from said email;
- b) a feature extractor configured to extract a byte sequence feature from said executable attachment, wherein said feature extractor is further configured to create a byte string representative of resources referenced by said executable attachment; and
- c) a rule evaluator configured to classify said executable attachment by comparing said byte sequence feature of said executable attachment to a classification rule set derived from byte sequence features of a set of executables having a predetermined class in a set of classes to determine the probability whether said executable attachment is malicious.

29. The system as defined in claim 28, wherein the feature extractor is configured to extract static properties of said executable attachment.

30. The system as defined in claim 28, wherein the feature extractor is configured to convert said executable attachment from binary format to hexadecimal format.

31. The system as defined in claim 28, wherein the rule evaluator is configured to predict the classification of said executable attachment as one class of a set of classes consisting of malicious and benign.

32. The system as defined in claim 28, which further comprises an email interface configured to log said class of said executable attachment classified in said step c).

33. The system as defined in claim 28, further comprising a model generator configured to update the classification rule set based on classified executable attachments.

22

34. A system for classifying an executable attachment in an email received at a server of a computer system comprising:

- a) an email filter configured to filter said executable attachment from said email;

- b) a feature extractor configured to extract a byte sequence feature from said executable attachment; and

- c) a rule evaluator is configured to predict the classification of said executable attachment as one class of a set of classes consisting of malicious, benign, and borderline by comparing said byte sequence feature of said executable attachment to a classification rule set derived from byte sequence features of a set of executables having a predetermined class in a set of classes.

35. The system as defined in claim 34, wherein the rule evaluator is configured to determine said probability that said executable attachment is a member of one class of said set of classes with a Naive Bayes algorithm.

36. The system as defined in claim 34, wherein the rule evaluator is configured to determine said probability that said executable attachment is a member of a class of said set of classes with a multi-Naive Bayes algorithm.

37. The system as defined in claim 34, wherein the rule evaluator is configured to divide a determination said probability into a plurality of processing steps and to execute said processing steps in parallel.

38. The system as defined in claim 34, wherein the rule evaluator is configured to classify said executable attachment as malicious if said probability that said executable attachment is malicious is greater than said probability that said executable attachment is benign.

39. The system as defined in claim 34, wherein the rule evaluator is configured to classify said executable attachment as benign if said probability that said executable attachment is benign is greater than said probability that said executable attachment is malicious.

40. The system as defined in claim 34, wherein the rule evaluator is configured to classify said executable attachment as borderline if a difference between said probability that said executable attachment is benign and said probability that said executable attachment is malicious is within a predetermined threshold.

41. The system as defined in claim 32, wherein said email interface is configured to increment a count of said executable attachments classified as borderline.

42. The system defined in claim 41, wherein said email interface is configured to, if said count of executable attachments exceeds a predetermined threshold, provide a notification that said threshold has been exceeded.

43. A method for classifying an executable program comprising:

- a) training a classification rule set based on a predetermined set of known executable programs having a predetermined class and one or more byte sequence features by recording the number of known executable programs in each said predetermined class that has each of said byte sequence features;

- b) extracting a byte sequence feature from said executable program comprising converting said executable program from binary format to hexadecimal format

- c) determining the probability that the executable program is within each said predetermined class in a set of classes consisting of malicious, benign, and borderline, based on said one or more byte sequence features in said executable and said classification rule set.

* * * * *

6

(12) **United States Patent**
Schultz et al.

(10) **Patent No.:** **US 7,979,907 B2**
 (45) **Date of Patent:** ***Jul. 12, 2011**

(54) **SYSTEMS AND METHODS FOR DETECTION OF NEW MALICIOUS EXECUTABLES**

(58) **Field of Classification Search** 726/13, 726/22-25; 713/156, 188; 709/206, 207, 709/225
 See application file for complete search history.

(75) Inventors: **Matthew G. Schultz**, Ithaca, NY (US);
Eleazar Eskin, Santa Monica, CA (US);
Erez Zadok, Middle Island, NY (US);
Manasi Bhattacharyya, Flushing, NY (US);
Stolfo Salvatore J., Ridgewood, NJ (US)

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,832,208 A * 11/1998 Chen et al. 726/24
 6,016,546 A * 1/2000 Kephart et al. 726/24
 6,161,130 A * 12/2000 Horvitz et al. 709/206
 6,732,149 B1 * 5/2004 Kephart 709/206
 2004/0073617 A1 * 4/2004 Milliken et al. 709/206
 2009/0132669 A1 * 5/2009 Milliken et al. 709/206
 * cited by examiner

(73) Assignee: **The Trustees of Columbia University in the City of New York**, New York, NY (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 129 days.
 This patent is subject to a terminal disclaimer.

Primary Examiner — Gilberto Barron, Jr.
Assistant Examiner — Abdulhakim Nobahar
 (74) *Attorney, Agent, or Firm* — Baker Botts LLP

(21) Appl. No.: **12/338,479**

(22) Filed: **Dec. 18, 2008**

(65) **Prior Publication Data**

US 2009/0254992 A1 Oct. 8, 2009

Related U.S. Application Data

(63) Continuation of application No. 10/208,432, filed on Jul. 30, 2002, now Pat. No. 7,487,544.
 (60) Provisional application No. 60/308,622, filed on Jul. 30, 2001, provisional application No. 60/308,623, filed on Jul. 30, 2001.

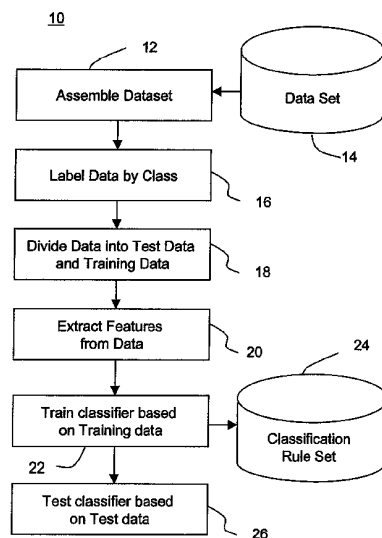
(51) **Int. Cl.**
G06F 11/00 (2006.01)
G06F 12/14 (2006.01)

(52) **U.S. Cl.** **726/24; 726/13; 713/188**

(57) **ABSTRACT**

A system and methods for detecting malicious executable attachments at an email processing application of a computer system using data mining techniques. The email processing application may be located at the server or at the client or host. The executable attachments are filtered from said email, and byte sequence features are extracted from the executable attachment. The executable attachments are classified by comparing the byte sequence feature of the executable attachment to a classification rule set derived from byte sequence features of a data set of known executables having a predetermined class in a set of classes, e.g., malicious or benign. The system is also able to classify executable attachments as borderline when the difference between the probability that the executable is malicious and the probability that the executable is benign are within a predetermined threshold. The system can notify the user when the number of borderline attachments exceeds the threshold in order to refine the classification rule set.

20 Claims, 7 Drawing Sheets



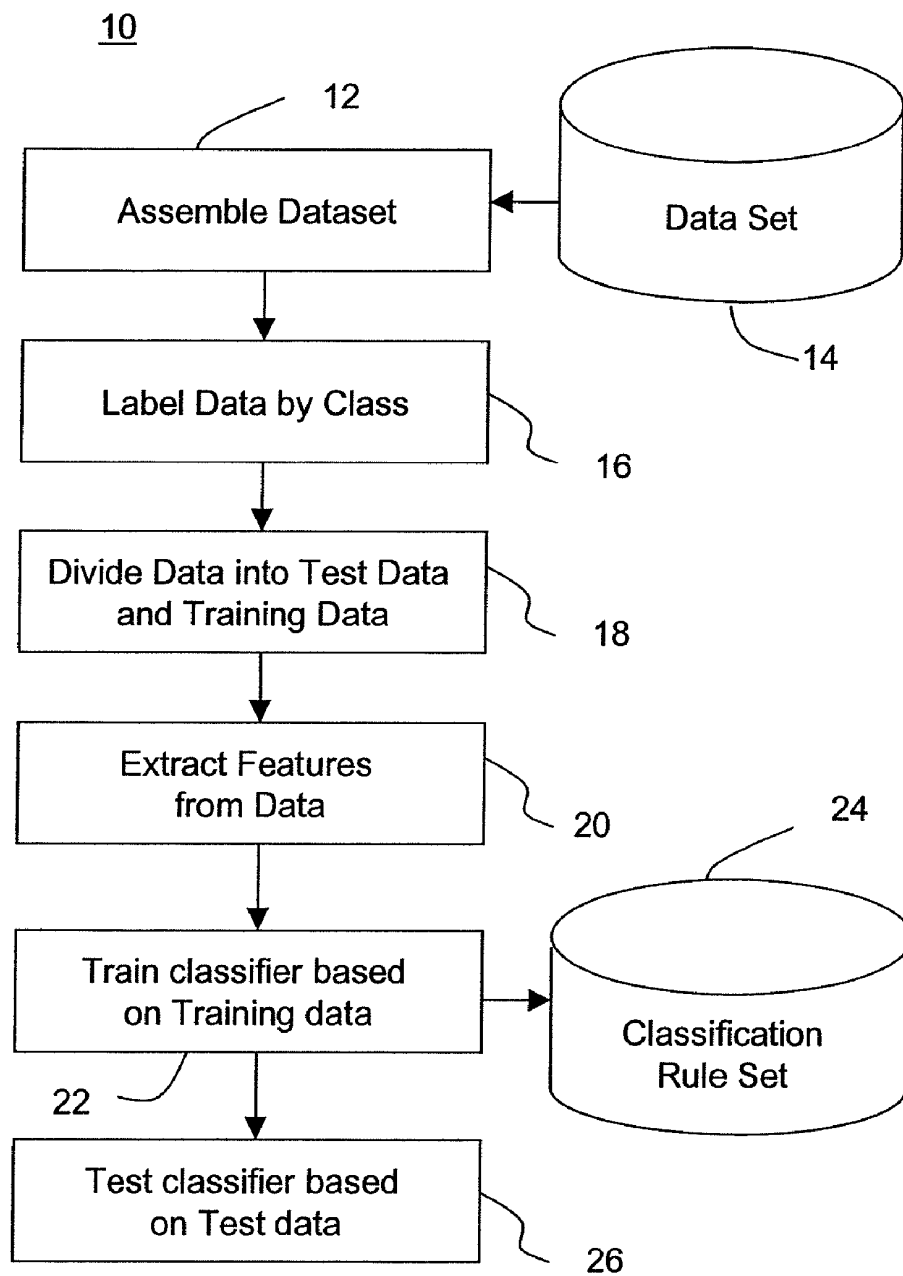


FIG. 1

```

646e 776f 2e73 0a0d 0024 0000 0000 0000
454e 3c0f 026c 0009 0000 0000 0302 0004
0400 2800 3924 0001 0000 0004 0004 0006
000c 0040 0060 021e 0238 0244 02f5 0000
0001 0004 0000 0802 0032 1304 0000 030a
    
```

FIG. 2

$\neg advapi32 \wedge avicap32 \wedge \dots \wedge winmm \wedge \neg wsock32$

FIG. 3

advapi32.AdjustTokenPrivileges()
 $\wedge advapi32.GetFileSecurityA() \wedge \dots$
 $\wedge wsock32.recv() \wedge wsock32.send()$

FIG. 4

$advapi32 = 2 \wedge avicap32 = 10 \wedge \dots$
 $\wedge winmm = 8 \wedge wsock32 = 2$

FIG. 5

malicious := $\neg user32.EndDialog() \wedge$
 $kernel32.EnumCalendarInfoA()$
malicious := $\neg user32.LoadIconA() \wedge$
 $\neg kernel32.GetTempPathA() \wedge \neg advapi32.$
malicious := $shell32.ExtractAssociatedIconA()$
malicious := $msvbvm.$
Benign := $otherwise$

FIG. 6

$P(" windows" \backslash benign)$ = 45/47
 $P(" windows" \backslash malicious)$ = 2/47
 $P(" *.COM" \backslash benign)$ = 1/12
 $P(" *.COM" \backslash malicious)$ = 11/12

FIG. 7

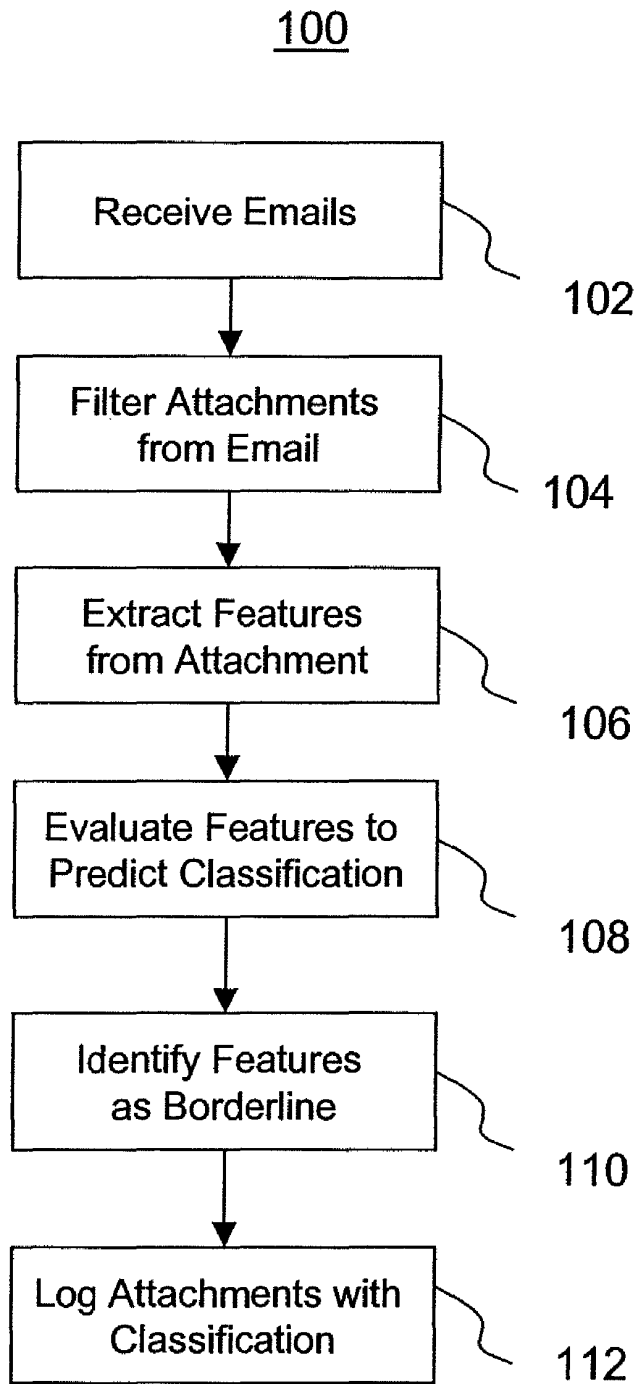


FIG. 8

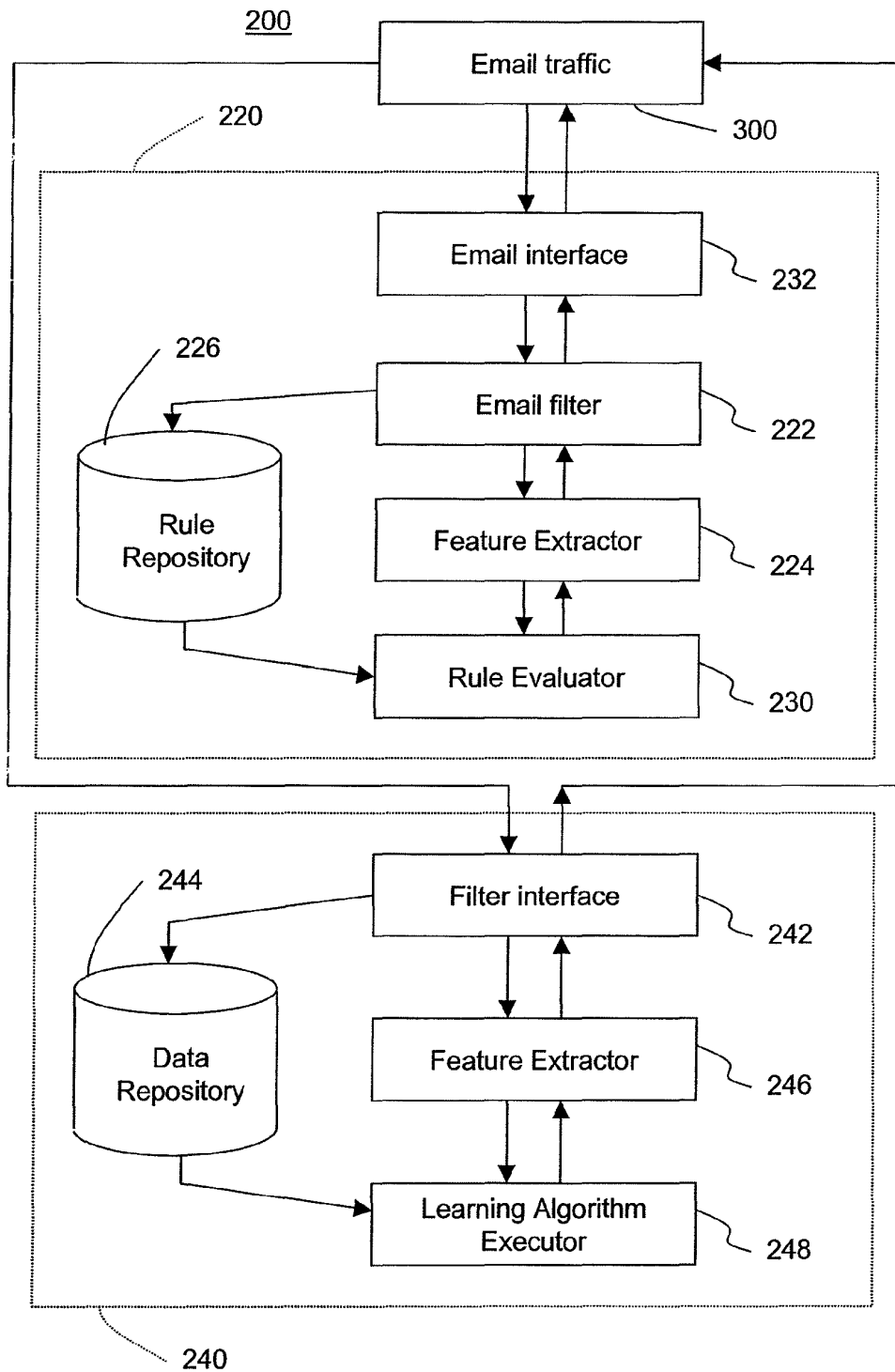


FIG. 9

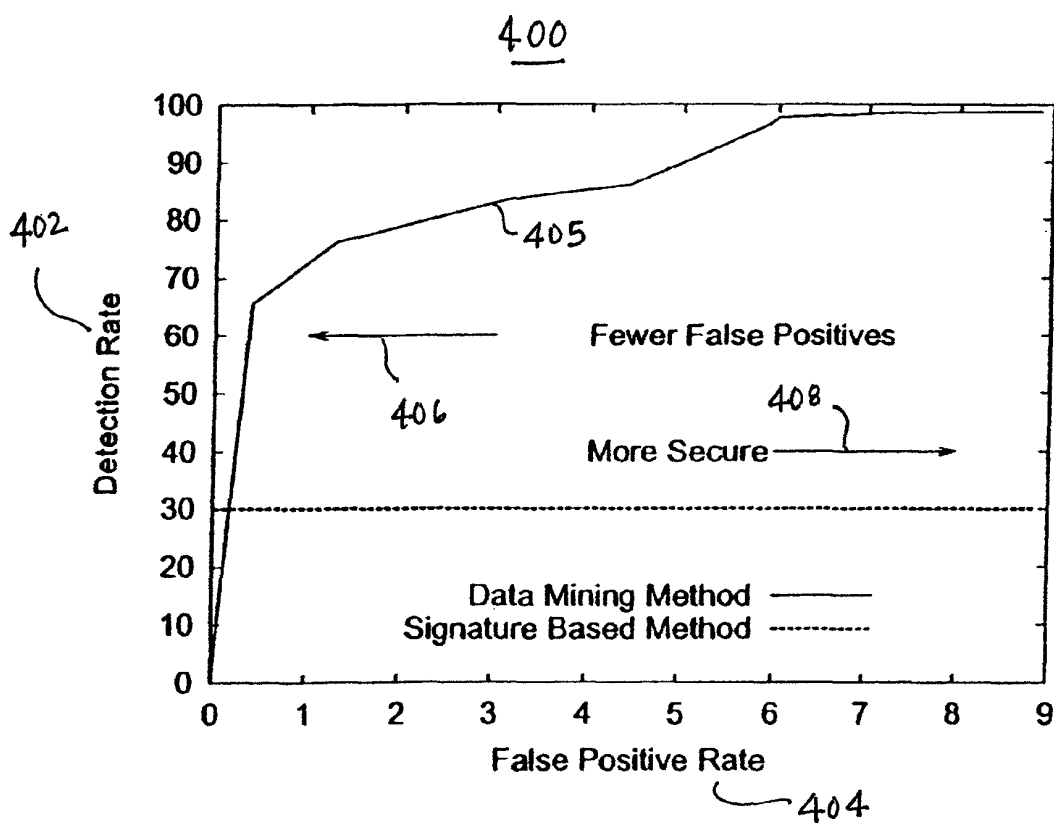


FIG. 11

US 7,979,907 B2

1

SYSTEMS AND METHODS FOR DETECTION OF NEW MALICIOUS EXECUTABLES

CLAIM FOR PRIORITY TO RELATED APPLICATIONS

This application claims the benefit of U.S. Provisional Patent Application Ser. Nos. 60/308,622, filed Jul. 30, 2001, entitled "Data Mining Methods for Detection of New Malicious Executables" and 60/308,623, filed on Jul. 30, 2001, entitled "Malicious Email Filter" which are incorporated by reference in their entirety herein. This application is a continuation of and claims the priority from U.S. patent application Ser. No. 10/208,432, filed Jul. 30, 2002, now U.S. Pat. No. 7,487,544 which is incorporated by reference in its entirety herein.

STATEMENT OF GOVERNMENT RIGHT

The present invention was made in part with support from the United States Defense Advanced Research Projects Agency (DARPA) grant nos. FAS-526617 and SRTSC-CU019-7950-1. Accordingly, the United States Government may have certain rights to this invention.

COPYRIGHT NOTICE

A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by any one of the patent disclosure, as it appears in the Patent and Trademark Office patent files or records, but otherwise reserves all copyright rights whatsoever.

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates to systems and methods for detecting malicious executable programs, and more particularly to the use of data mining techniques to detect such malicious executables in email attachments.

2. Background

A malicious executable is a program that performs a malicious function, such as compromising a system's security, damaging a system or obtaining sensitive information without the user's permission. One serious security risk is the propagation of these malicious executables through e-mail attachments. Malicious executables are used as attacks for many types of intrusions. For example, there have been some high profile incidents with malicious email attachments such as the ILOVEYOU virus and its clones. These malicious attachments are capable of causing significant damage in a short time.

Current virus scanner technology has two parts: a signature-based detector and a heuristic classifier that detects new viruses. The classic signature-based detection algorithm relies on signatures (unique telltale strings) of known malicious executables to generate detection models. Signature-based methods create a unique tag for each malicious program so that future examples of it can be correctly classified with a small error rate. These methods do not generalize well to detect new malicious binaries because they are created to give a false positive rate as close to zero as possible. Whenever a detection method generalizes to new instances, the tradeoff is for a higher false positive rate.

Unfortunately, traditional signature-based methods may not detect a new malicious executable. In an attempt to solve

2

this problem, the anti-virus industry generates heuristic classifiers by hand. This process can be even more costly than generating signatures, so finding an automatic method to generate classifiers has been the subject of research in the anti-virus community. To solve this problem, different IBM researchers applied Artificial Neural Networks (ANNs) to the problem of detecting boot sector malicious binaries. (The method of detection is disclosed in G. Tesauro et al., "Neural Networks for Computer Virus Recognition, *IEE Expert*, 11(4):5-6, August 1996, which is incorporated by reference in its entirety herein.) An ANN is a classifier that models neural networks explored in human cognition. Because of the limitations of the implementation of their classifier, they were unable to analyze anything other than small boot sector viruses which comprise about 5% of all malicious binaries.

Using an ANN classifier with all bytes from the boot sector malicious executables as input, IBM researchers were able to identify 80-85% of unknown boot sector malicious executables successfully with a low false positive rate (<1%). They were unable to find a way to apply ANNs to the other 95% of computer malicious binaries.

In similar work, Arnold and Tesauro applied the same techniques to Win32 binaries, but because of limitations of the ANN classifier they were unable to have the comparable accuracy over new Win32 binaries. (This technique is described in Arnold et al.: "Automatically Generated Win 32 Heuristic Virus Detection," *Proceedings of the 2000 International Virus Bulletin Conference*, 2000, which is incorporated by reference in its entirety herein.)

The methods described above have the shortcoming that they are not applicable to the entire set of malicious executables, but rather only boot-sector viruses, or only Win32 binaries.

The technique is similar to data mining techniques that have already been applied to Intrusion Detection Systems by Lee et al. Their methods were applied to system calls and network data to learn how to detect new intrusions. They reported good detection rates as a result of applying data mining to the problem of IDS. A similar framework is applied to the problem of detecting new malicious executables. (The techniques are described in W. Lee et al., "Learning Patterns From UNIX Processes Execution Traces for Intrusion Detection and Risk Management, 1997, pages 50-56, and W. Lee et al., "A Data Mining Framework for Building Intrusion Detection Models," *IEEE Symposium on Security and Privacy*, 1999, both of which are incorporated by reference in their entirety herein.)

Procmail is a mail processing utility which runs under UNIX, and which filters email; and sorts incoming email according to sender, subject line, length of message, keywords in the message, etc. Procmail's pre-existent filter provides the capability of detecting active-content HTML tags to protect users who read their mail from a web browser or HTML-enabled mail client. Also, if the attachment is labeled as malicious, the system "mangles" the attachment name to prevent the mail client from automatically executing the attachment. It also has built in security filters such as long filenames in attachments, and long MIME headers, which may crash or allow exploits of some clients.

However, this filter lacks the ability to automatically update its list of known malicious executables, which may leave the system vulnerable to attacks by new and unknown viruses. Furthermore, its evaluation of an attachment is based solely on the name of the executable and not the contents of the attachment itself.

US 7,979,907 B2

3

Accordingly, there exists a need in the art for a technique which is not limited to particular types of files, such as boot-sector viruses, or only Win32 binaries, and which provides the ability to detect new, previously unseen files.

SUMMARY

An object of the present invention is to provide a technique for predicting a classification of an executable file as malicious or benign which is not dependent upon the format of the executable.

Another object of the present invention is to provide a data mining technique which examines the entire file, rather than a portion of the file, such as a header, to classify the executable as malicious or benign.

A further object of the present invention is to provide an email filter which can detect executables that are borderline, i.e., executables having features indicative of both malicious and benign executables, which may be detrimental to the model if misclassified.

These and other objects of the invention, which will become apparent with reference to the disclosure herein, are accomplished by a system and methods for classifying an executable attachment in an email received by an email processing application or program, which includes filtering the executable attachment from said email. The email processing application may be executed at an email server or a client or host email application. A byte sequence feature is subsequently extracted from the executable attachment. The executable attachment is classified by comparing said byte sequence feature of the executable attachment with a classification rule set derived from byte sequence features of a set of executables having a predetermined class in a set of classes.

According to a preferred embodiment, extracting the byte sequence feature from said executable attachment comprises extracting static properties of the executable attachment, which are properties that do not require the executable to be run in order to discern. Extracting the byte sequence feature from the executable attachment may comprise converting the executable attachment from binary format to hexadecimal format. According to another embodiment, extracting the byte sequence features from the executable attachment may comprise creating a byte string representative of resources referenced by said executable attachment.

Advantageously, classifying the executable attachment may comprise predicting the classification of the executable attachment as one class in a set of classes consisting of malicious and benign. The set of classes may also include a borderline class. Classifying the executable attachment may comprise determining a probability or likelihood that the executable attachment is a member of each class in said set of classes based on said byte sequence feature. In one embodiment, this probability is determined by use of a Naive Bayes algorithm. In another embodiment, the probability may be determined by use of a Multi-Naive Bayes algorithm. The determination of the probability may be divided into a plurality of processing steps. These processing steps may then be performed in parallel. The executable attachment is classified as malicious if the probability that the executable attachment is malicious is greater than said probability that the executable attachment is benign. The executable attachment is classified as benign if the probability that the executable attachment is benign is greater than said probability that said executable attachment is malicious. The executable attachment is classified as borderline if a difference between the

4

probability the executable attachment is benign and the probability the executable attachment is malicious is within a predetermined threshold.

A further step in accordance with the method may include logging the class of the executable attachment. The step of logging the class of the executable attachment may further include incrementing a count of the executable attachments classified as borderline. If the count of executable attachments classified as borderline exceeds a predetermined threshold, the system may provide a notification that the threshold has been exceeded.

In accordance with the invention, the objects as described above have been met, and the need in the art for a technique which can analyze previously unseen malicious executables, without regard to the type of file, has been satisfied.

BRIEF DESCRIPTION OF THE DRAWINGS

Further objects, features and advantages of the invention will become apparent from the following detailed description taken in conjunction with the accompanying figures showing illustrative embodiments of the invention, in which:

FIG. 1 is a flow chart illustrating an overview of a method of detection model generation in accordance with the present invention.

FIG. 2-4 illustrate a several approaches to binary profiling. FIG. 5 illustrates sample classification rules determined from the features represented in FIG. 3.

FIG. 6 illustrates sample classification rules found by a RIPPER algorithm.

FIG. 7 illustrates sample classification rules found by a Naive Bayes algorithm.

FIG. 8 is a flow chart illustrating a method of detecting malicious executables in accordance with the present invention.

FIG. 9 is a simplified diagram illustrating the architecture of the malicious email detector and model generator in accordance with the present invention.

FIG. 10 is a flow chart, similar to FIG. 8, illustrating another method of detecting malicious executables in accordance with the present invention.

FIG. 11 is a plot illustrating the interactive effect of false positive rate and detection rate on the performance of the detection model or classifier in accordance with the present invention.

Throughout the figures, the same reference numerals and characters, unless otherwise stated, are used to denote like features, elements, components or portions of the illustrated embodiments. Moreover, while the subject invention will now be described in detail with reference to the figures, it is done so in connection with the illustrative embodiments. It is intended that changes and modifications can be made to the described embodiments without departing from the true scope and spirit of the subject invention as defined by the appended claims.

DETAILED DESCRIPTION OF EXEMPLARY EMBODIMENTS

This invention will be further understood in view of the following detailed description.

An exemplary system and methods for detecting malicious email attachments was implemented in UNIX with respect to Sendmail (a message transfer agent (MTA) which ensures messages get from source message servers to destination message servers for recipients to access their email, as produced by Sendmail, Inc. or Emeryville, Calif.) using Proc-

US 7,979,907 B2

5

mail (a publicly available program that processes e-mail messages received by the server, as further described in Stephen R. van den Berg and Philip Guenther, "Procmail," online publication as viewed on <http://www.procmail.org>, 2001). This system and methods uses data mining methods in order to create the detection model. The data mining methods are used to create classifiers to detect the malicious executables. A classifier is a classification rule set, or detection model, generated by the data mining algorithm that was trained over, i.e., derived from a given set of training data.

In accordance with the exemplary embodiment, a data mining-based filter integrates with Procmail's pre-existent security filter to detect malicious executable attachments. The filter uses a scoring system based on a data mining classifier to determine whether or not an attachment may be malicious. If an attachment's score is above a certain threshold it is considered malicious. The data mining classifier provides the ability to detect both the set of known malicious executables and a set of previously unseen, but similar malicious executables.

A flowchart illustrating the process 10 of creating of the classification rule set is illustrated in FIG. 1. An early stage in the process is to assemble the dataset (step 12) which will be used for training, and for optionally testing the detection model. In the exemplary embodiment, this step included gathering a large set of executables 14 from public sources. In addition, each example program in the data set is a Windows or MS-DOS format executable, although the framework is applicable to other formats. In the exemplary embodiment, the programs were gathered either from FTP sites, or personal computers in the Data Mining Lab at Columbia University. A total of 4,031 programs were used.

In a subsequent stage, each data item, or executable, is labeled by class (step 16). The learning problem in the exemplary embodiment is defined with two classes, e.g., malicious and benign. As discussed above, a malicious executable is defined to be a program that performs a malicious function, such as compromising a system's security, damaging a system, or obtaining sensitive information without the user's permission. A benign program does not perform such malicious functions. Thus, the data set was divided into two groups: (1) malicious and (2) benign executables. In order to train the classification rule set, the classes of the executables must be known in advance. Of the 4,031 programs used in the data set, 3,301 were malicious executables and 1,000 were benign executables. The malicious executables consisted of viruses, Trojans, and cracker/network tools. There were no duplicate programs in the data set. To standardize the data-set, an updated McAfee's virus scanner, produced by McAfee.com Corporation of Sunnyvale, Calif., was used to label the programs as either malicious or benign executables. All labels were assumed to be correct for purposes of the analysis.

Another step, which may be performed concurrently with or subsequent to the above step, is to divide the dataset into two subsets which include a training set and a test set (step 18). The data mining algorithms use the training set to generate the classification rule sets. After training, a test set may be used to test the accuracy of the classifiers on a set of unseen examples. It is understood that testing the detection model is an optional step to determine the accuracy of the detection model, and, as such, may be omitted from the process.

The next step of the method is to extract features from each executable (Step 20). Features in a data mining framework are defined as properties extracted from each example program in the data set, e.g., byte sequences, that a classifier uses to generate detection models. (Signatures, as distinguished from features, typically refer to a specific feature value, while

6

a feature is a property or attribute of data (such as "byte sequence feature") which may take on a set of values. Signature based methods are methods that inspect and test data to determine whether a specific feature value is present in that data, and then classify or alarm accordingly.) In the present invention, the presence of specific feature values is used by the learning algorithms to calculate a probability or likelihood of classification of the data. The features which are extracted in the exemplary embodiment are static properties, which are properties that do not require executing the binary in order to be detected or extracted.

In the exemplary embodiment, hexdump was used in the feature extraction step. Hexdump, as is known in the art (Peter Miller, "Hexdump," on line publication 2000, <http://gd.tu-wien.ac.at/softeng/Aegis/hexdump.html> which is incorporated by reference in its entirety herein), is an open source tool that transforms binary files into hexadecimal files. The byte sequence feature is informative because it represents the machine code in an executable. After the "hexdumps" are created, features are produced in the form illustrated in FIG. 2 in which each line represents a short sequence of machine code instructions. In the analysis, a guiding assumption is made that similar instructions were present in malicious executables that differentiated them from benign programs, and the class of benign programs had similar byte code that differentiated them from the malicious executables. Each byte sequence in the program is used as a feature.

Many additional methods of feature extraction are also useful to carry out step 20, above, and are described herein. For example, octale encoding may be used rather than hexadecimal encoding. According to another approach to feature extraction is to extract resource information from the binary that provides insight to its behavior, which is also referred to herein as "binary profiling." According to this approach, a subset of the data may be examined which is in Portable Executable (PE) format (which is described in "Portable Executable Format," online publication as viewed on, <http://support.microsoft.com/support/kb/articles/Q121/4/60.asp>, 1999, which is incorporated by reference in its entirety herein.) For instance, an executable in a standard Windows user interface may normally call the User Interfaces Dynamically Linked Library (USER32.DLL). This approach assumes that if an executable being evaluated does not call USER32.DLL, then the program does not have the standard Windows user interface. To extract resource information from Windows executables, GNU's Bin-Utils may be used (as described in "GNU Binutils Cygwin, online publication as viewed on <http://sourceware.cygwin.com/cygwin>, 1999, which is incorporated by reference in its entirety herein). GNU's Bin-Utils suite of tools can analyze PE binaries within Windows. In PE, or Common Object File Format (COFF), program headers are composed of a COFF header, an Optional header, at MS-DOS stub, and a file signature. All of the information about the binary is obtained from the program header without executing the unknown program but by examining the static properties of the binary, using libBFD, which is a library within Bin-Utils, to extract information in object format. Object format for a PE binary gives the file size, the names of DLLs, and the names of function calls within those DLLs and Relocation Tables. From the object format, it is possible to extract a set of features to compose a feature vector, or string, for each binary representative of resources referenced by the binary.

Three types of features may be analyzed to determine how resources affect a binary's behavior:

1. The list of DLLs used by the binary
2. The list of DLL function calls made by the binary
3. The number of different function calls within each DLL

A first approach to binary profiling used the DLLs loaded by the binary as features. Data can be modeled by extracting a feature or a set of features, and each set of features may be represented as a vector of feature values. The feature vector comprised of a number of boolean values, e.g., 30, representing whether or not a binary used a DLL. Typically, not every DLL was used in all of the binaries, but a majority of the binaries called the same resource. For example, almost every binary called GDI32.DLL, which is the Windows NT Graphics Device Interface and is a core component of WinNT. The example vector given in FIG. 3 is composed of at least two unused resources: ADVAPI32.DLL, the Advanced Windows API, and WSOCK32.DLL, the Windows Sockets API. It also uses at least two resources: AVI-CAP32.DLL, the AVI capture API, and WINMM.DLL, the Windows Multimedia API.

A second approach to binary profiling uses DLLs and their function calls as features. This approach is similar to the first, described above, but with added function call information. The feature vector is composed of a greater number, e.g., 2,229, of boolean values. Because some of the DLL's had the same function names it was important to record which DLL the function came from. The example vector given in FIG. 4 is composed of at least four resources. Two functions were called in AD-VAP132.DLL: AdjustTokenPrivileges() and GetFileSecurityA(), and two functions were called in WSOCK32.DLL: recv() and send().

A third approach to binary profiling counts the number of different function calls used within each DLL, and uses such counts as features. The feature vector included several, e.g., 30, integer values. This profile provides an approximate measure of how heavily a DLL is used within a specific binary. This is a macro-resource usage model because the number of calls to each resource is counted instead of detailing referenced functions. For example, if a program only called the recv() and send() functions of WSOCK32.DLL, then the count would be 2. It should be noted that this third approach does not count the number of times those functions might have been called. The example vector given in FIG. 5 describes an example that calls two functions in ADVAPI32.DLL, ten functions in AVICAP32.DLL, eight functions in WINMM.DLL, and two functions from WSOCK32.DLL.

Another method useful for feature extraction (step 20) does not require PE format for the executables, and therefore is applicable to Non-PE executables. Headers in PE format are in plain text, which allows extraction of the same information from the PE executables by extracting the plain text headers. Non-PE executables also have strings encoded in them. This information is used to classify the entire data set, rather than being limited only to the subset of data including libBFD, described above. To extract features from the data set according to this third method, the GNU strings program was used. The strings program extracts consecutive printable characters from any file. Typically there are many printable strings in binary files. Some common strings found in the dataset are illustrated in Table 1.

TABLE 1

kernel	microsoft	windows	getmodulehandlea
getversion	getstartupinfoa	win	getmodulefilenamea
messageboxa	closehandle	null	dispatchmessagea
library	getprocaddress	advapi	getlasterror
loadlibrarya	exitprocess	heap	getcommandlinea
reloc	createfilea	writefile	setfilepointer
application	showwindow	time	regclosekey

Through testing it was observed that similar strings were present in malicious executables that distinguished them from benign programs, and similar strings in benign programs that distinguished them from malicious executables. According to this technique, each string in the binary was used as a feature for the classifier.

Once the features were extracted using hexdump, or any other feature extraction method, such as those described herein, a classifier was trained to label a program as malicious or benign (Step 22). The classifier computes the probability or likelihood that a program is a member of a certain classification given the features or byte strings that are contained in that program. (Throughout the description herein, the term "probability" will generally refer to probability or likelihood, except where specified.)

In the exemplary embodiment, the classifier was a Naive Bayes classifier that was incorporated into Procmail, as will be described in greater detail herein. A Naive Bayes classifier is one exemplary machine learning algorithm that computes a model of a set of labeled training data and subsequently may use that model to predict the classification of other data. Its output is a likelihood (based on mathematical probability theory) associated with each classification possible for the other data. The Naive Bayes algorithm computes the likelihood that a program is a member each classification, e.g., malicious and benign, given the features or byte strings that are contained in that program. For instance, if a program contained a significant number of malicious byte sequences and a few or no benign sequences, then it labels that binary as malicious. Likewise, a binary that was composed of many benign features and a smaller number of malicious features is labeled benign by the system. In accordance with the invention, the assumption was made that there were similar byte sequences in malicious executables that differentiated them from benign programs, and the class of benign programs had similar sequences that differentiated them from the malicious executables.

In particular, the Naive Bayes algorithm first computes (a) the probability that a given feature is malicious and (b) the probability that the feature is benign, by computing statistics on the set of training data. Then to predict whether a binary, or collection of hex strings, was malicious or benign, those probabilities were computed for each hex string in the binary, and then the Naive Bayes independence assumption was used. The independence assumption was applied in order to efficiently compute the probability that a binary was malicious and the probability that the binary was benign.

Specifically, the Naive Bayes algorithm computes the class C of a program, given that the program contains a set of features F. (The Naive Bayes algorithm is described, for example, in T. Mitchell, "Naive Bayes Classifier," *Machine Learning*, McGraw-Hill, 1997, pp. 177-180, which is incorporated by reference in its entirety herein.) The term C is defined as a random variable over the set of classes: benign and malicious executables. That is, the classifier computes P(C|F), the probability that a program is in a certain class C given the program contains the set of features F. According to the Bayes rule, the probability is expressed in equation (1):

$$P(C | F) = \frac{P(F | C) * P(C)}{P(F)} \tag{1}$$

To use the Naive Bayes rule, it is assumed that the features occur independently from one another. If a program F include the features F₁, F₂, F₃, . . . , F_n, then equation (1) may be

US 7,979,907 B2

9

re-written as equation (2). (In this description, subscripted features F_x refers to a set of code strings.)

$$P(C|F) = \frac{\prod_{i=1}^n P(F_i|C) * P(C)}{\prod_{j=1}^n P(F_j)} \quad (2)$$

Each $P(F_i|C)$ is the frequency that feature string F_i occurs in a program of class C . $P(C)$ is the proportion of the class C in the entire set of programs.

The output of the classifier is the highest probability class for a given set of strings. Since the denominator of equation (1) is the same for all classes, the maximum class is taken over all classes C of the probability of each class computed in (2) to get equation (3):

$$\text{Most Likely Class} = \underset{C}{\text{max}} \left(P(C) \prod_{i=1}^n P(F_i|C) \right) \quad (3)$$

In equation (3), the term max_C denotes the function that returns the class with the highest probability. "Most Likely Class" is the class in C with the highest probability and hence the most likely classification of the example with features F .

To train the classifier, a record was made for how many programs in each class contained each unique feature. This information was used to classify a new program into an appropriate class. Feature extraction, as described above, was used to determine the features contained in the program. Then, equation (3) was applied to compute the most likely class for the program.

The Naive Bayes method is a highly effective technique, but also requires significant amounts of main memory, such as RAM, e.g., greater than 1 gigabyte, to generate a detection model when the data or the set of features it analyzes is very large. To make the algorithm more efficient, the problem may be divided into smaller pieces that would fit in memory and generate a classifier for each of the subproblems. For example, the subproblem was to classify based on 16 subsets of the data organized according to the first letter of the hex string. This data mining algorithm is referred to as "Multi-Naive Bayes." This algorithm was essentially a collection of Naive Bayes algorithms that voted on an overall classification for an example. The Multi-Naive Bayes calculations are advantageously executed in a parallel and distributed computing system for increased speed.

According to this approach, several Naive Bayes classifiers may be trained so that all hex strings are trained on. For example, one classifier is trained on all hex strings starting with an "A", and another on all hex strings starting with an "0". This is done 16 times and then a voting algorithm is used to combine their outputs. Each Naive Bayes algorithm classified the examples in the test set as malicious or benign, and this counted as a vote. The votes are combined by the Multi-Naive Bayes algorithm to output a final classification for all the Naive Bayes.

According to the exemplary embodiment, the data may be divided evenly into several sets, e.g. six sets, by putting each i th line in the binary into the $(i \bmod n)$ th set where n is the number of sets. For each set, a Naive Bayes classifier is trained. The prediction for a binary is the product of the predictions of the n classifiers. In the exemplary embodiment, 6 classifiers ($n=6$) were used.

10

More formally, the Multi-Naive Bayes promotes a vote of confidence between all of the underlying Naive Bayes classifiers. Each classifier determines a probability of a class C given a set of bytes F which the Multi-Naive Bayes uses to generate a probability for class C given features F over all the classifiers.

The likelihood of a class C given feature F and the probabilities learned by each classifier NaiveBayes_i are determined. In equation (4) the likelihood $L_{NB}(C|F)$ of class C given a set of feature F was computed:

$$L_{NB}(C|F) = \prod_{i=1}^{NB} P_{NB_i}(C|F) / P_{NB_i}(C) \quad (4)$$

where NB_i is a Naive Bayes classifier and NB is the set of all combined Naive Bayes classifiers (in this case 6). $P_{NB_i}(C|F)$ (generated from equation (2)) is the probability for class C computed by the classifier NaiveBayes_i given F divided by the probability of class C computed by NaiveBayes_i . Each $P_{NB_i}(C|F)$ was divided by $P_{NB_i}(C)$ to remove the redundant probabilities. All the terms were multiplied together to compute $L_{NB}(C|F)$, the final likelihood of C given F . $|NB|$ is the size of the set NB such that $\prod_{i=1}^{|NB|} P_{NB_i}(C|F)$.

The output of the multi-classifier given a set of bytes F is the class of highest probability over the classes given $L_{NB}(C|F)$ and $P_{NB}(C)$ the prior probability of a given class, which is represented by equation (5), below.

$$\text{Most Likely Class} = \underset{C}{\text{max}} (P_{NB}(C) * L_{NB}(C|F)) \quad (5)$$

Most Likely Class is the class in C with the highest probability hence the most likely classification of the example with features F , and max_C returns the class with the highest likelihood.

Additional embodiments of classifiers are described herein, which are also useful to classify an executable as benign or malicious. Alternatively, inductive rule learners may be used as classifiers. Another algorithm, RIPPER, is an inductive rule learner (RIPPER is described in W. Cohen, "Learning Trees and Rules with Set-Valued Features," *American Association for Artificial Intelligence*, 1996, which is incorporated by reference in its entirety herein). This algorithm generates a detection model composed of resource rules that was built to detect future examples of malicious executables. RIPPER is a rule-based learner that builds a set of rules that identify the classes while minimizing the amount of error. The error is defined by the number of training examples misclassified by the rules. This algorithm may be used with libBFD information as features, which were described above.

As is known in the art, an inductive algorithm learns what a malicious executable is, given a set of training examples. Another useful algorithm for building a set of rules is Find-S. Find-S finds the most specific hypothesis that is consistent with the training examples. For a positive training example the algorithm replaces any feature in the hypothesis that is inconsistent with the training example with a more general feature. For example, four features seen in Table 2 are:

1. "Does it have a GUI?"
2. "Does it perform a malicious function?"

US 7,979,907 B2

11

3. "Does it compromise system security?"
 4. "Does it delete files?"
 and finally the classification label "Is it malicious?"

TABLE 2

	Has a GUI?	Malicious Function?	Compromise Security?	Deletes Files?	Is it malicious?
1	yes	yes	yes	no	yes
2	no	yes	yes	yes	yes
3	yes	no	no	yes	no
4	yes	yes	yes	yes	yes

The defining property of any inductive learner is that no a priori assumptions have been made regarding the final concept. The inductive learning algorithm makes as its primary assumption that the data trained over is similar in some way to the unseen data.

A hypothesis generated by an inductive learning algorithm for this learning problem has four features. Each feature will have one of these values:

1. T, truth, indicating any value is acceptable in this position,
2. a value, either yes, or no, is needed in this position, or
3. a \perp , falsity, indicating that no value is acceptable for this position.

For example, the hypothesis $\langle T, T, T, T \rangle$ and the hypothesis $\langle \text{yes}, \text{yes}, \text{yes}, \text{no} \rangle$ would make the first example in Table 2 true. $\langle T, T, T, T \rangle$ would make any feature set true and $\langle \text{yes}, \text{yes}, \text{yes}, \text{no} \rangle$ is the set of features for example one.

Of all the hypotheses, values 1 is more general than 2, and 2 is more general than 3. For a negative example, the algorithm does nothing. Positive examples in this problem are defined to be the malicious executables and negative examples are the benign programs.

The initial hypothesis that Find-S starts with is $\langle \perp, \perp, \perp, \perp \rangle$. This hypothesis is the most specific because it is true over the fewest possible examples, none. Examining the first positive example in Table 2, $\langle \text{yes}, \text{yes}, \text{yes}, \text{no} \rangle$, the algorithm chooses the next most specific hypothesis $\langle \text{yes}, \text{yes}, \text{yes}, \text{no} \rangle$. The next positive example, $\langle \text{no}, \text{no}, \text{no}, \text{yes} \rangle$, is inconsistent with the hypothesis in its first and fourth attribute ("Does it have a GUI?" and "Does it delete files?") and those attributes in the hypothesis get replaced with the next most general attribute, T.

The resulting hypothesis after two positive examples is $\langle T, \text{yes}, \text{yes}, T \rangle$. The algorithm skips the third example, a negative example, and finds that this hypothesis is consistent with the final example in Table 2. The final rule for the training data listed in Table 2 is $\langle T, \text{yes}, \text{yes}, T \rangle$. The rule states that the attributes of a malicious executable, based on training data, are that it has a malicious function and compromises system security. This is consistent with the definition of a malicious executable stated above. Thus, it does not matter in this example if a malicious executable deletes files, or if it has a GUI or not.

RIPPER looks at both positive and negative examples to generate a set of hypotheses that more closely approximate the target concept while Find-S generates one hypothesis that approximates the target concept.

Each of the data mining algorithms generated its own classification rule set **24** to evaluate new examples. The classifi-

12

cation rule sets are incorporated in the filter to detect malicious executables, as will be described below in the exemplary embodiment. For purposes herein, a classification rule set is considered to have the standard meaning in data mining terminology, i.e., a set of hypotheses that predict the classification, e.g., malicious or benign, of an example, i.e., an executable, given the existence of certain features. The Naive Bayes rules take the form of $P(F|C)$, the probability of an feature F given a class C. The probability for a string occurring in a class is the total number of times it occurred in that class's training set divided by the total number of times that the string occurred over the entire training set. An example of such hypotheses are illustrated in FIG. 7. Here, the string "windows" was predicted to more likely occur in a benign program and string "*.COM" was more than likely in a malicious executable program.

This approach compensates for those instances where a feature, e.g., a hex string, occurs in only one class in the training data. When this occurs, the probability is arbitrarily increased from $0/n$, where n is the number of occurrences, to $1/n$. For example, a string (e.g. "AAAA") may occur in only one set, e.g., in the malicious executables. The probability of "AAAA" occurring in any future benign example is predicted to be 0, but this is an incorrect assumption. If a program was written to print out "AAAA" it will always be tagged a malicious executable even if it has other strings in it that would have labeled it benign. In FIG. 6, the string "*.COM" does not occur in any benign programs so the probability of "*.COM" occurring in class benign is approximated to be $1/12$ instead of $0/11$. This approximates real world probability that any string could occur in both classes even if during training it was only seen in one class.

The rule sets generated by the Multi-Naive Bayes algorithm are the collection of the rules generated by each of the component Naive Bayes classifiers. For each classifier, there is a rule set such as the one in FIG. 6. The probabilities in the rules for the different classifiers may be different because the underlying data that each classifier is trained on is different. The prediction of the Multi-Naive Bayes algorithm is the product of the predictions of the underlying Naive Bayes classifiers.

RIPPER's rules were built to generalize over unseen examples so the rule set was more compact than the signature-based methods. For the data set that contained 3,301 malicious executables the RIPPER rule set contained the five rules in FIG. 6.

Here, a malicious executable was consistent with one of four hypotheses:

1. it did not call `user32.EndDialog()` but it did call `kernel32.EnumCalendarInfoA()`
2. it did not call `user32.LoadIconA()`, `kernel32.GetTempPathA()`, or any function in `advapi32.dll`
3. it called `shell32.ExtractAssociatedIconA()`,
4. it called any function in `msvbmm.dll`, the Microsoft Visual Basic Library

A binary is labeled benign if it is inconsistent with all of the malicious binary hypotheses in FIG. 6.

Each data mining algorithm generated its own rule set **24** to evaluate new examples. The detection models are stored for subsequent application to classify previously unseen examples, as will be described below. An optional next step is to test the classification rule set **24** against the test data (step **26**). This step is described in greater detail below.

The process **100** of detecting malicious emails in accordance with the invention is illustrated in FIG. 8. A first step is to receive the emails at the server (step **102**). In the exemplary

US 7,979,907 B2

13

embodiment, the mail server is Sendmail. Procmail is a publicly available program that processes e-mail messages received by the server and looks for particular information in the headers or body of the message, and takes action on what it finds.

Subsequently, the emails are filtered to extract attachments or other components from the email (step 104). The executable attachments may then be saved to a file. In the exemplary embodiment, `html_trap.procmail`, a commonly-available routine, has been modified to include a call to a novel routine, `parser3`, which performs the functions of filtering attachments. The routine `parser3` includes a call to the routine `extractAttachments`, for example, which extracts executable attachments and other items of the email and saves them to a file, e.g., `$files_full_ref`, and also provides a string containing a directory of where the executable attachments are saved, e.g., `$dir`, and a reference to an array containing a list of the file names, e.g., `$files_ref`.

Features in the executable attachment are extracted (step 106), and those features are subsequently analyzed and used to classify the executable attachment as malicious or benign. In the exemplary embodiment, the routine `parser3` also includes a call to the routine `scanAttachments`, which in turn calls the routine `hexScan`, which performs the feature extraction of step 106. In particular, `hexScan` includes a function call to `hexdump`, a commonly-available routine which transforms the binary files in the attachment into a byte sequence of hexadecimal characters, as described above and illustrated in FIG. 2. The resulting hexadecimal string is saved as `"/tmp/$$.hex"`. These strings of hexadecimal code are the "features" which are used in the classification, described below. This byte sequence is useful because it represents the machine code in an executable. In addition, this approach involves analyzing the entire binary, rather than portions such as headers, an approach which consequently provides a great deal of information about the executable. It is understood that the feature extraction step described herein is alternatively performed with a binary profiling method in another embodiment, as described above and illustrated in FIGS. 3-4, and with a GNU strings method, also described above and illustrated in Table 1. In these embodiments, the step of calling the routine `hexScan` in `scanAttachments` is replaced by calls to routines that perform the binary profiling or GNU strings analysis.

The features extracted from the attachment in step 106 are evaluated using the classification rule set as described above, and the attachment is classified as malicious or benign (step 108). In the exemplary embodiment, the routine `hexScan` subsequently calls the routine `senb`, which calculates "scores" associated with the attachments. (As will be described below, such scores are representative of whether a binary is malicious or benign.) The routine `senb` evaluates the features, e.g., the hexadecimal string `"/tmp/$$.hex"` produced by `hexdump` against the rules in the Classification Rule Set, e.g., `"/etc/procmail/senb/aids_model.txt"`, and returns with a first score associated with the probability that the string is malicious and a second score associated with the probability that the string is benign. In order to obtain these scores, the routine `senb` invokes the routine `check_file`, which performs the Naive Bayes analysis on the features as described above in equation (1)-(5), and calculates scores associated with the probability that the program is malicious and benign. Where the Multi-Naive Bayes algorithm is used, the data is partitioned into components which are processed in parallel to increase processing speed. The routine `hexScan` then determines which of the scores is greater, e.g., malicious or benign. In other embodiments, a different classification algorithm

14

may be implemented, such as a function call to the RIPPER algorithm which will evaluate the features extracted in step 106 to determine whether they are malicious or benign.

A further step may be to identify the programs as borderline (step 110). Borderline executables are defined herein as programs that have similar probabilities of being benign and malicious (e.g., 50% chance it is malicious, and 50% chance it is benign; 60% chance malicious and 40% chance benign; 45% chance malicious and 55% chance benign, etc.). Due to the similar probabilities, borderline executables are likely to be mislabeled as either malicious or benign. Since borderline cases could potentially lower the detection and accuracy rates by being misclassified, it is desirable to identify these borderline cases, properly classify them as malicious or benign, and update the classification rule set to provide increased accuracy to the detection of malicious executables. The larger the data set that is used to generate models, then the more accurate the detection models will be. To execute this process, the system identifies programs as borderline using the criteria described below, and archives the borderline cases. At periodic intervals, the system sends the collection of these borderline cases to a central server, by the system administrator. Once at a central repository, such as data repository 244, these binaries can then be analyzed by experts to determine whether they are malicious or not, and subsequently included in the future versions of the detection models. Preferably, any binary that is determined to be a borderline case will be forwarded to the repository and wrapped with a warning as though it were a malicious attachment.

An exemplary metric to identify borderline cases, which may be implemented in `hexScan` or a similar routine is to define a borderline case to be a case where the difference between the probability, or score, that the program is malicious and the probability, or score, it is benign is below a threshold. This threshold may be set based on the policies of the host. For example, in a secure setting, the threshold could be set relatively high, e.g., 20%. In this case, all binaries that have a 60/40 split are labeled as borderline. In other words, binaries with a 40-60% chance (according to the model) of being malicious and 40-60% chance of being benign would be labeled borderline. This setting can be determined by the system administrator. An exemplary default setting of 51.25/48.75 may be used with a threshold of 2.5%, which was derived from testing.

The routine `scanAttachments` receives the output of `hexScan` which is a determination of whether the program is malicious or benign, and assigns the string a boolean "0" or "1." (Where the probabilities of being malicious and of being benign are similar, it may be labeled borderline, as discussed above.) Subsequently, `scanAttachments` invokes the routine `md5log` to associate a unique identifier for each attachment in by using the MD5 algorithm, (as described in R. Rivest, "The MD5 Message Digest Algorithm," *Internet RFC1321*, Paril 1992, which is incorporated by reference in its entirety herein.) The input to MD5 is the hexadecimal representation of the binary. These identifiers are then kept in a log along with other information such as whether the attachment was malicious, benign, or borderline and with what certainty the system made those predictions (Step 112).

The results of this analysis are sent from `parser3` to `html_trap.procmail`, which inserts warnings that the file may be malicious and may quarantine the attachment. The routine `html_trap.procmail` reintegrates filtered email back into normal email traffic.

An exemplary system 200 in accordance with the invention is illustrated in FIG. 9. The system 200 includes a malicious email detector 220 and model generator 240. The system may

US 7,979,907 B2

15

resides on the server of a computer or on a host or client of the computer system to receive emails before they are forwarded to users of the system.

The malicious email detector may include an email filter 222, a feature extractor 224, a rule repository 226, a rule evaluator 230, and an email interface 232. In the exemplary embodiment, the email filter 222 may include the routine Parser3 which filters attachments from the emails as described above. Parser3 calls the routine extractAttachments, for example, which extracts attachments and other items of the email. The email filter 222 may also filter out updated classification rules sent by the model generator 240, and forward them to the rule repository 226.

The feature extractor 224 receives the executable attachments and extracts those byte sequence features which will be analyzed and used to classify the program as malicious or benign. In the exemplary embodiment, the routine scanAttachments calls the routine hexScan, which performs the function of the feature extractor 224. In particular, hexScan includes a function call to hexdump, which transforms the binary files into hexadecimal strings. The rule repository 226 may be a database which contains the classification rule set generated by a data mining model in a process such as that illustrated in FIG. 1. The rule evaluator 230 evaluates the byte sequence features extracted from the attachments by the feature extractor 224 using the classification rule set provided by the rule repository 226.

In the exemplary embodiment, the routine hexScan calls the routine senb, which performs the function of rule evaluator 230. The routine senb evaluates the byte sequence features, e.g., the hexadecimal string against the classification rule set in the rule repository 26, e.g., "/etc/procmail/senb/aids_model.txt," and returns with a score that the string is malicious and a score that the string is benign. The rule evaluator 230 may also provide an indication that the string is borderline.

The results of this analysis may be sent to the email interface 232, which reintegrates filtered email back into normal email traffic 300, and which may send the model generator 240 (described below) each attachment to be analyzed further. If the program was considered malicious by the rule evaluator 230, the email interface 232 may add warnings to the email or quarantine the email. In the exemplary embodiment, the routine html-trap.procmail performs this function.

The classification rule set may require updates periodically. For example, after a number of borderline cases have been identified by the rule evaluator 230, it may be desirable to generate a new detection model, and subsequently distribute the updated models. This embodiment of the system 300, which is illustrated in FIG. 10, is substantially identical to system 100, with the differences noted herein. For example, the email filter 222 may maintain a running counter of the number of borderline executables identified (step 150 of FIG. 9). When a predetermined threshold is exceeded (proportional to the overall traffic of email received) (step 152 of FIG. 9), a notification may be sent that the threshold has been exceeded (step 154 of FIG. 9). Subsequently, the model generator 240 may be invoked to generate an updated classification rule set.

A new classification rule set is generated at the model generator 240 by running the data mining algorithm on the new data set that contains the borderline cases along with their correct classification (as determined by expert analysis), and the existing training data set. As described herein, the data mining algorithm may be a Naive Bayes or a Multi-Naive Bayes algorithm, or any other appropriate algorithm for calculating the probability or likelihood that a feature is a mem-

16

ber of a class. When the Multi-Naive Bayes analysis is used herein, the data is partitioned into several components and all the components may be processed in parallel to increase speed. This updated model may then be distributed to the malicious email detector 220. The filter interface 242 may receive copies of all attachments from the email interface 232. In the exemplary embodiment, the routine senb may perform the function of the filter interface 242. The data repository 244 receives copies of attachments from the filter interface 42, and stores the attachments. In the exemplary embodiment, the attachments may be stored as a datafile.

The feature extractor 246 accesses attachments stored in the data repository 244, and then extracts features from the attachments. This function may be performed by invoking the hexdump routine, as described above. The learning algorithm executor 248 receives features from the feature extractor 246 and executes learning algorithms on the features extracted from the attachments to generate an updated classification rule set. In the exemplary embodiment, the routine senb calls the routine test_table, which in turn invokes test_class. Test_class invokes test_file, which performs the function of creating the updated classification rule set, including performing the Naive Bayes calculations, described above in equations (1)-(5).

In an exemplary embodiment, the filter interface 242 receives the classification model rule set from the learning algorithm executor 248 and transmits the classification model rule set to the malicious email detector 220, where it is used to update the classification rule set stored in the rule repository 226. According to the exemplary embodiment, portions of the classification model rule set that have changed may be distributed, rather than the entire classification model rule set, to improve efficiency. In order to avoid constantly sending a large model from the model generator 240 to the malicious email detector 220, the administrator is provided with the option of receiving this smaller file. Using the update algorithm, the older model can then be updated. The full model will also be available to provide additional options for the system administrator. Efficient update of the model is possible because the underlying representation of the models is probabilistic. Thus, the model is a count of the number of times that each byte string appears in a malicious program versus the number of times that it appears in a benign program. An update model can then be easily summed with the older model to create a new model. From these counts the algorithm computes the probability that an attachment is malicious in a method described above. In order to combine the models, the counts of the old model are summed with the new information.

As shown in Table 3, in model A, the old detection model, a byte string occurred 99 times in the malicious class, and one time in the benign class. In model B, the update model, the same byte string was found three times in the malicious class and four times in the benign class. The combination of models A and B would state that the byte string occurred 102 times in the malicious class and five times in the benign class. The combination of A and B would be the new detection model after the update.

TABLE 3

Model A (old)
The byte string occurred in 99 malicious executables
The byte string occurred in 1 benign executable

TABLE 3-continued

Model B (new)
The byte string occurred in 3 malicious executables The byte string occurred in 4 benign executables.
Model C (update)
The byte string occurred in 102 malicious executables The byte string occurred in 5 benign executables.

To compare the results of the methods and system described herein with traditional methods, a prior art signature-based method was implemented (step 26, of FIG. 1). First, the byte-sequences that were only found in the malicious executable class were calculated. These byte-sequences were then concatenated together to make a unique signature for each malicious executable example. Thus each malicious executable signature contained only byte-sequences found in the malicious executable class. To make the signature unique, the byte-sequences found in each example were concatenated together to form one signature. This was done because a byte-sequence that was only found in one class during training could possibly be found in the other class during testing, and lead to false positives when deployed.

The virus scanner that was used to label the data set (step 16, above) contained signatures for every malicious example in the data set, so it was necessary to implement a similar signature-based method. This was done to compare the two algorithms' accuracy in detecting new malicious executables. In the tests, the signature-based algorithm was only allowed to generate signatures for the same set of training data that the data mining method used. This allowed the two methods to be fairly compared. The comparison was made by testing the two methods on a set of binaries not contained in the training set.

To quantify the performance of the method described herein, statistics were computed on the performance of the data mining-based method, tables 4 and 5 are included herein which include counts for true positives, true negatives, false positives and false negatives. A true positive, TP, is a malicious example that is correctly classified as malicious, and a true negative, TN, is a benign example that is correctly classified as benign. A false positive, FP, is a benign program that has been mislabeled by an algorithm as malicious, while a false negative, FN, is a malicious executable that has been mis-classified as a benign program.

The overall accuracy of the algorithm is calculated as the number of programs the system classified correctly divided by the total number of binaries tested. The detection rate is the number of malicious binaries correctly classified divided by the total number of malicious programs tested.

The results were estimated over new executables by using 5-fold cross validation technique, as described in R. Kohavi, "A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection," *IJCAI*, 1995. Cross-validation, as is known in the art, is the standard method to estimate the performance of predictions over unseen data in Data Mining. For each set of binary profiles the data was partitioned into five equal size partitions. Four of the partitions were used for training a model and then evaluating that model on the remaining partition. Then the process was repeated five times leaving out a different partition for testing each time. This provided a measure of the method's accuracy on unseen data. The results of these five tests were averaged to obtain a measure of how the algorithm performs over the entire set.

To evaluate the algorithms over new executables, the algorithms generated their detection models over the set of train-

ing data and then tested their models over the set of test data. This was done five times in accordance with cross fold validation.

Tables 4 displays the results. The data mining algorithm had the highest detection rate, 97.76%, compared with the signature-based method's detection rate of 33.96%. Along with the higher detection rate the data mining method had a higher overall accuracy, 96.88% vs. 49.31%. The false positive rate of 6.01% though was higher than the signature-based method, 0%.

TABLE 4

Profile Type	Detection Rate	False Positive Rate	Overall Accuracy
Signature Method	33.96%	0%	49.31%
Data Mining Method	97.76%	6.01%	96.88%

FIG. 11 displays the plot of the detection rate vs. false positive rate using Receiver Operation Characteristic curves, as described in K. H. Zou et al., "Smooth Non-Parametric ROC Curves for Continuous Diagnostic Tests," *Statistics in Medicine*, 1997. Receiver Operating Characteristic (ROC) curves are a way of visualizing the trade-offs between detection and false positive rates. In this instance, the ROC curve shows how the data mining method (illustrated in dashed line) can be configured for different environments. For a false positive rate less than or equal to 1% the detection rate would be greater than 70%, and for a false positive rate greater than 8% the detection rate would be greater than 99%. Thus, more secure settings would select a threshold setting associated with a point on the data mining line towards the right (indicated by arrow), and applications needing fewer false alarms should choose a point towards the left (indicated by arrow).

The performance of the models in detecting known executables was also evaluated. For this task, the algorithms generated detection models for the entire set of data. Their performance was then evaluated by testing the models on the same training set.

As shown in Table 5, both methods detected over 99% of known executables. The data mining algorithm detected 99.87% of the malicious examples and misclassified 2% of the benign binaries as malicious. However, the signatures for the binaries that the data mining algorithm misclassified were identified, and the algorithm can include those signatures in the detection model without lowering accuracy of the algorithm in detecting unknown binaries. After the signatures for the executables that were misclassified during training had been generated and included in the detection model, the data mining model had a 100% accuracy rate when tested on known executables.

TABLE 5

Profile Type	Detection Rate	False Positive Rate	Overall Accuracy
Signature Method	100%	0%	100%
Data Mining Method.	99.87%	2%	99.44%

In order for the data mining algorithm to quickly generate the models, it is advantageous for all calculations to be done in memory. The algorithm consumed space in excess of a gigabyte of RAM. By splitting the data into smaller pieces, the algorithm was done in memory with no loss in accuracy. In addition, the calculations may be performed in parallel.

19

The training of a classifier took 2 hours 59 minutes and 49 seconds running on Pentium III 600 Linux machine with 1 GB of RAM. The classifier took on average 2 minutes and 28 seconds for each of the 4,301 binaries in the data set. The amount of system resources taken for using a model are equivalent to the requirements for training a model. So on a Pentium III 600 Linux box with 1 GB of RAM it would take on average 2 minutes 28 seconds per attachment. Another advantageous of splitting the data into smaller partitions (in connection with the Multi-Naive Bayes analysis) is that the Naive Bayes algorithm is executed on each partition on parallel hardware, which reduces the total training time from 2 hours and 59 minutes, to 2 minutes and 28 seconds if each piece is concurrently executed.

It will be understood that the foregoing is only illustrative of the principles of the invention, and that various modifications can be made by those skilled in the art without departing from the scope and spirit of the invention.

What is claimed is

1. A method for classifying an executable attachment in an email received at a computer system comprising:

- a) filtering said executable attachment from said email;
- b) extracting a byte sequence feature from said executable attachment; and
- c) classifying said executable attachment by comparing said byte sequence feature of said executable attachment with a classification rule set derived from byte sequence features of a set of executables having a predetermined class in a set of classes,

wherein said classifying comprises determining using a computer processor, with a Multi-Naive Bayes algorithm, a probability that said executable attachment is a member of each class in said set of classes based on said byte sequence feature and dividing said step of determining said probability into a plurality of processing steps and executing said processing steps in parallel.

2. The method as defined in claim 1, wherein the step of extracting said byte sequence feature from said executable attachment comprises extracting static properties of said executable attachment.

3. The method as defined in claim 1, wherein the step of extracting said byte sequence feature from said executable attachment comprises converting said executable attachment from binary format to hexadecimal format.

4. The method as defined in claim 1, wherein the step of extracting said byte sequence features from said executable attachment comprises creating a byte string representative of resources referenced by said executable attachment.

5. The method as defined in claim 1, wherein the step of classifying said executable attachment comprises determining a probability that said executable attachment is a member of each class in a set of classes consisting of malicious and benign.

6. The method as defined in claim 1, wherein the step of classifying said executable attachment comprises determining a probability that said executable attachment is a member of each class in a set of classes consisting of malicious, benign, and borderline.

7. The method as defined in claim 1, wherein the step of classifying the executable attachment comprises classifying said executable attachment as malicious if said probability that said executable attachment is malicious is greater than said probability that said executable attachment is benign.

8. The method as defined in claim 1, wherein the step of classifying the executable attachment comprises classifying said executable attachment as benign if said probability that said executable attachment is benign is greater than said probability that said executable attachment is malicious.

20

9. The method as defined in claim 1, wherein the step of classifying the executable attachment comprises classifying said executable attachment as borderline if a difference between said probability that said executable attachment is benign and said probability that said executable attachment is malicious is within a predetermined threshold.

10. A system for classifying an executable attachment in an email received at a computer system comprising:

one or more computer processors executing instructions which implement:

- a) an email filter configured to filter said executable attachment from said email;
- b) a feature extractor configured to extract a byte sequence feature from said executable attachment; and
- c) a rule evaluator configured to: classify said executable attachment by comparing said byte sequence feature of said executable attachment to a classification rule set derived from byte sequence features of a set of executables having a predetermined class in a set of classes,

determine a probability that said executable attachment is a member of a class of said set of classes based on said byte sequence feature, and

divide the determination of said probability into a plurality of processing steps and to execute said processing steps in parallel.

11. The system as defined in claim 10, wherein the feature extractor is configured to extract static properties of said executable attachment.

12. The system as defined in claim 10, wherein the feature extractor is configured to convert said executable attachment from binary format to hexadecimal format.

13. The system as defined in claim 10, wherein the feature extractor is configured to create a byte string representative of resources referenced by said executable attachment.

14. The system as defined in claim 10, wherein the rule evaluator is configured to predict the classification of said executable attachment as one class of a set of classes consisting of malicious and benign.

15. The system as defined in claim 10, wherein the rule evaluator is configured to predict the classification of said executable attachment as one class of a set of classes consisting of malicious, benign, and borderline.

16. The system as defined in claim 10, wherein the rule evaluator is configured to determine said probability that said executable attachment is a member of one class of said set of classes with a Naive Bayes algorithm.

17. The system as defined in claim 10, wherein the rule evaluator is configured to determine said probability that said executable attachment is a member of a class of said set of classes with a multi-Naive Bayes algorithm.

18. The system as defined in claim 10, wherein the rule evaluator is configured to classify said executable attachment as malicious if said probability that said executable attachment is malicious is greater than said probability that said executable attachment is benign.

19. The system as defined in claim 10, wherein the rule evaluator is configured to classify said executable attachment as benign if said probability that said executable attachment is benign is greater than said probability that said executable attachment is malicious.

20. The system as defined in claim 10, wherein the rule evaluator is configured to classify said executable attachment as borderline if a difference between said probability that said executable attachment is benign and said probability that said executable attachment is malicious is within a predetermined threshold.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE
CERTIFICATE OF CORRECTION

PATENT NO. : 7,979,907 B2
APPLICATION NO. : 12/338479
DATED : July 12, 2011
INVENTOR(S) : Matthew G. Schultz et al.

Page 1 of 1

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

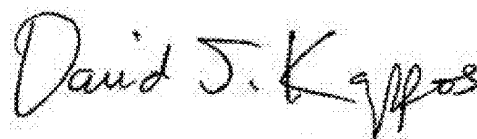
ON THE TITLE PAGE:

--Item (75) Inventors: Matthew G. Schultz, Ithaca, NY (US); Eleazar Eskin, Santa Monica, CA (US); Erez Zadok, Middle Island, NY (US); Manasi Bhattacharyya, Flushing, NY (US); Stolfo Salvatore, Ridgewood, NJ (US)

should read

--Item (75) Inventors: Matthew G. Schultz, Ithaca, NY (US); Eleazar Eskin, Santa Monica, CA (US); Erez Zadok, Middle Island, NY (US); Manasi Bhattacharyya, Flushing, NY (US); Salvatore J. Stolfo, Ridgewood, NJ (US)

Signed and Sealed this
Sixteenth Day of August, 2011



David J. Kappos
Director of the United States Patent and Trademark Office

7

(12) **United States Patent**
Apap et al.

(10) **Patent No.:** **US 7,448,084 B1**
 (45) **Date of Patent:** **Nov. 4, 2008**

(54) **SYSTEM AND METHODS FOR DETECTING INTRUSIONS IN A COMPUTER SYSTEM BY MONITORING OPERATING SYSTEM REGISTRY ACCESSES**

6,778,995 B1 8/2004 Gallivan
 6,820,081 B1 11/2004 Kawai et al.
 6,888,548 B1 5/2005 Gallivan
 6,978,274 B1 12/2005 Gallivan et al.
 7,035,876 B2 4/2006 Kawai et al.
 7,080,076 B1 7/2006 Williamson et al.
 2003/0070003 A1* 4/2003 Chong et al. 709/330
 2006/0174319 A1* 8/2006 Kraemer et al. 726/1

(75) Inventors: **Frank Apap**, Valley Stream, NY (US);
Andrew Honig, East Windsor, NJ (US);
Hershkop Shlomo, Brooklyn, NY (US);
Eleazar Eskin, Santa Monica, CA (US);
Salvatore J. Stolfo, Ridgewood, NJ (US)

OTHER PUBLICATIONS

Eskin, E., M. Miller, Z.D. Zhong, G. Yi, W.A. Lee, and S. J. Stolfo. Adaptive Model Generation for Intrusion Detection Systems. Workshop on Intrusion Detection and Prevention, 7th ACM Conference on Computer Security, Athens. Nov. 2000.*

(73) Assignee: **The Trustees of Columbia University in the City of New York**, New York, NY (US)

(Continued)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 422 days.

Primary Examiner—Christopher A Revak
 (74) *Attorney, Agent, or Firm*—Baker Botts LLP

(21) Appl. No.: **10/352,343**

(57) **ABSTRACT**

(22) Filed: **Jan. 27, 2003**

A method for detecting intrusions in the operation of a computer system is disclosed which comprises gathering features from records of normal processes that access the files system of the computer, such as the Windows registry, and generating a probabilistic model of normal computer system usage based on occurrences of said features. The features of a record of a process that accesses the Windows registry are analyzed to determine whether said access to the Windows registry is an anomaly. A system is disclosed, comprising a registry auditing module configured to gather records regarding processes that access the Windows registry; a model generator configured to generate a probabilistic model of normal computer system usage based on records of a plurality of processes that access the Windows registry and that are indicative of normal computer system usage; and a model comparator configured to determine whether the access of the Windows registry is an anomaly.

Related U.S. Application Data

(60) Provisional application No. 60/351,857, filed on Jan. 25, 2002.

(51) **Int. Cl.**
G06F 21/22 (2006.01)
G06F 11/30 (2006.01)

(52) **U.S. Cl.** **726/24; 726/22**
 (58) **Field of Classification Search** **726/23, 726/22, 4**

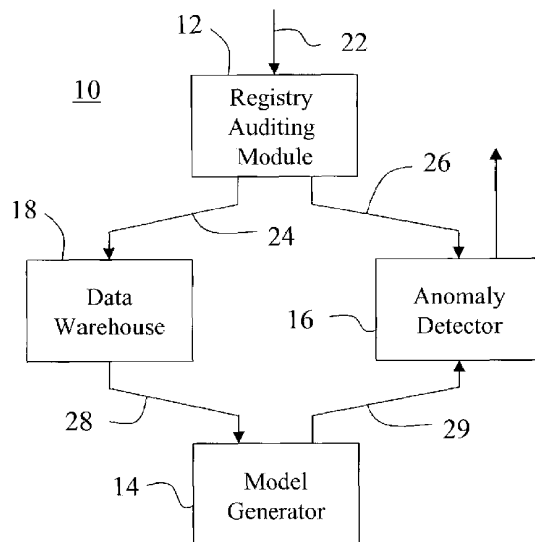
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

6,647,400 B1* 11/2003 Moran 707/205
 6,742,124 B1* 5/2004 Kilpatrick et al. 726/23

28 Claims, 3 Drawing Sheets



US 7,448,084 B1

Page 2

OTHER PUBLICATIONS

- Korba, Jonathan. Windows NT Attacks for the Evaluation of Intrusion Detection Systems. May 2000.*
- Lee et al., "A Framework for Constructing Features and Models for Intrusion Detection Systems", Nov. 2000, ACM Transactions on Information and System Security, vol. 3, No. 4, pp. 22.*
- U.S. Appl. No. 10/352,342, filed Jan. 27, 2003 claiming priority to U.S. Appl. No. 60/351,913, filed Jan. 25, 2002, entitled "Data Warehouse Architecture For Adaptive Model Generation Capability In Systems For Detecting Intrusion In Computer Systems," of Salvatore J. Stolfo, Eleazar Eskin, Matthew Miller, Juxin Zhang and Zhi-Da Zhong. (AP34982).
- U.S. Appl. No. 10/327,811, filed Dec. 19, 2002 claiming priority to U.S. Appl. No. 60/342,872, filed Dec. 20, 2001, entitled "System and Methods for Detecting A Denial-Of-Service Attack On A Computer System" of Salvatore J. Stolfo, Shlomo Hershkop, Rahul Bhan, Suhail Mohiuddin and Eleazar Eskin. (AP34898).
- U.S. Appl. No. 10/320,259, filed Dec. 16, 2002 claiming priority to U.S. Appl. No. 60/328,682, filed Oct. 11, 2001 and U.S. Appl. No. 60/352,894, filed Jan. 29, 2002, entitled "Methods of Unsupervised Anomaly Detection Using A Geometric Framework" of Eleazar Eskin, Salvatore J. Stolfo and Leonid Portnoy. (AP34888).
- U.S. Appl. No. 10/269,718, filed Oct. 11, 2002 claiming priority to U.S. Appl. No. 60/328,682, filed Oct. 11, 2001 and U.S. Appl. No. 60/340,198, filed Dec. 14, 2001, entitled "Methods For Cost-Sensitive Modeling For Intrusion Detection" of Salvatore J. Stolfo, Wenke Lee, Wei Fan and Matthew Miller. (AP34885).
- U.S. Appl. No. 10/269,694, filed Oct. 11, 2002 claiming priority to U.S. Appl. No. 60/328,682, filed Oct. 11, 2001 and U.S. Appl. No. 60/339,952, filed Dec. 13, 2001, entitled "System And Methods For Anomaly Detection And Adaptive Learning" of Wei Fan, Salvatore J. Stolfo. (AP34886).
- U.S. Appl. No. 10/222,632, filed Aug. 16, 2002 claiming priority to U.S. Appl. No. 60/312,703, filed Aug. 16, 2001 and U.S. Appl. No. 60/340,197, filed Dec. 14, 2001, entitled "System And Methods For Detecting Malicious Email Transmission" of Salvatore J. Stolfo, Eleazar Eskin, Manasi Bhattacharyya and Matthew G. Schultz. (AP34887).
- U.S. Appl. No. 10/208,432, filed Jul. 30, 2002 claiming priority to U.S. Appl. No. 60/308,622, filed Jul. 30, 2001 and U.S. Appl. No. 60/308,623, filed Jul. 30, 2001, entitled "System And Methods For Detection Of New Malicious Executables" of Matthew G. Schultz, Eleazar Eskin, Erez Zadok and Salvatore J. Stolfo. (AP34542).
- U.S. Appl. No. 10/208,402, filed Jul. 30, 2002 claiming priority to U.S. Appl. No. 60/308,621, filed Jul. 30, 2001, entitled "System And Methods For Intrusion Detection With Dynamic Windows Sizes" of Eleazar Eskin and Salvatore J. Stolfo. (AP34541).
- Honig A et al., (2002) "Adaptive model generation: An Architecture for the deployment of data mining-based intrusion detection systems." In *Data Mining for Security Applications*. Kluwer.
- Friedman N. et al., (1999) "Efficient bayesian parameter estimation in large discrete domains."
- Javits HS et al., Mar. 7, 1994, "The nides statistical component: Description and justification." *Technical report, SRI International*.
- D. E. Denning, An Intrusion Detection Model, *IEEE Transactions on Software Engineering*, SE-13:222-232, 1987.
- Wenke Lee, Sal Stolfo, and Kui Mok. "Mining in a Data-flow Environment: Experience in Network Intrusion Detection" In *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '99)*, San Diego, CA, Aug. 1999.
- Stephanie Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A Sense of Self for UNIX Processes," *IEEE Computer Society*, pp. 120-128, 1996.
- Christina Warrender, Stephanie Forrest, and Barak Pearlmutter, "Detecting Intrusions Using System Calls: Alternative Data Models," *IEEE Computer Society*, pp. 133-145, 1999.
- S. A. Hofmeyr, Stephanie Forrest, and A. Somayaji, "Intrusion Detect Using Sequences of System Calls," *Journal of Computer Society*, 6:151-180, 1998.
- W. Lee, S. J. Stolfo, and P. K. Chan, "Learning Patterns from UNIX Processes Execution Traces for Intrusion Detection," AAAI Press, pp. 50-56, 1997.
- Eleazar Eskin, "Anomaly Detection Over Noisy Data Using Learned Probability Distributions," *Proceedings of the Seventeenth International Conference on Machine Learning (ICML-2000)*, 2000.
- N. Friedman and Y. Singer, "Efficient Bayesian Parameter Estimation in Large Discrete Domains," *Advances in Neural Information Processing Systems 11*, MIT Press, 1999.
- M. Mahoney and P. Chan, "Detecting Novel Attacks by Identifying Anomalous Network Packet Headers," *Technical Report CS-2001-2*, Florida Institute of Technology, Melbourne, FL, 2001.
- H. Debar et al., "Intrusion Detection Exchange Format Data Model," Internet Engineering Task Force, Jun. 15, 2000.

* cited by examiner

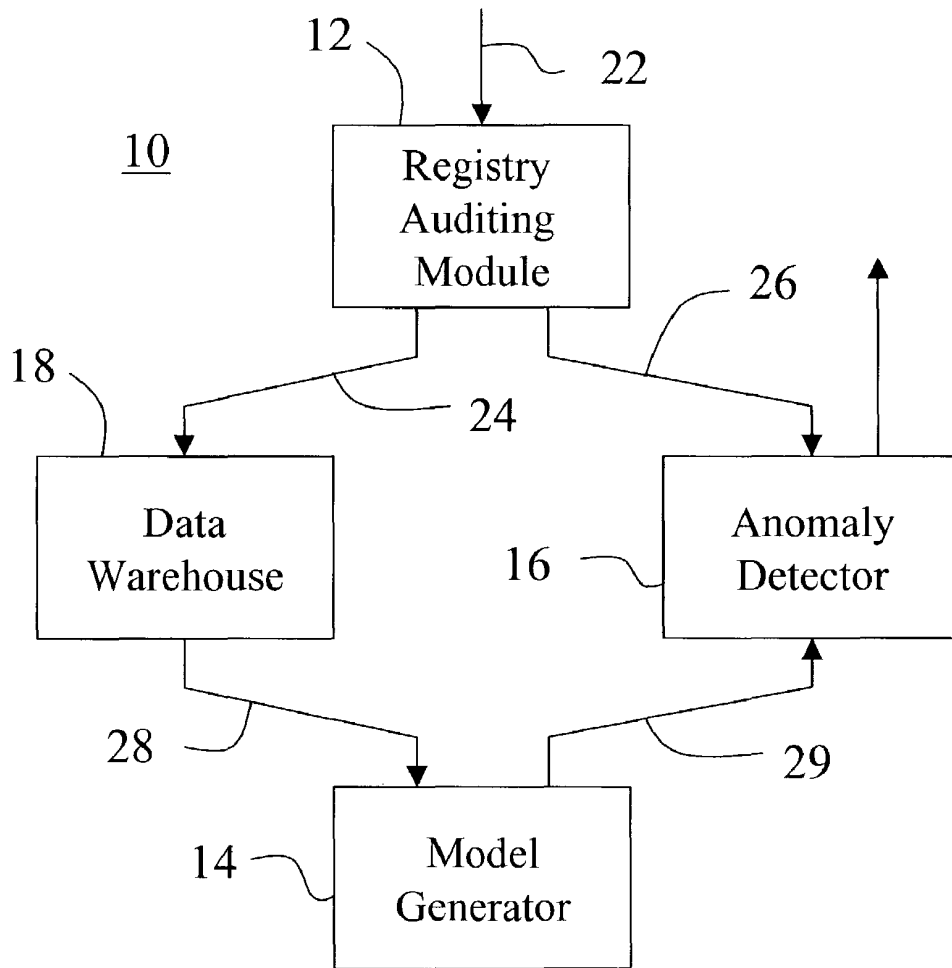


FIG. 1

30

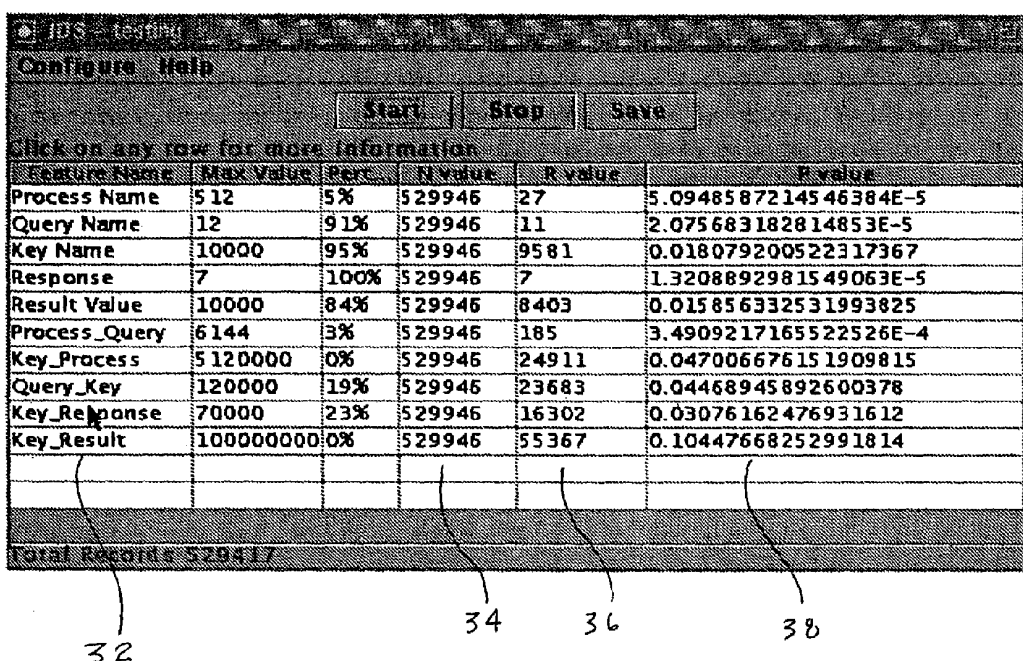


FIG. 2

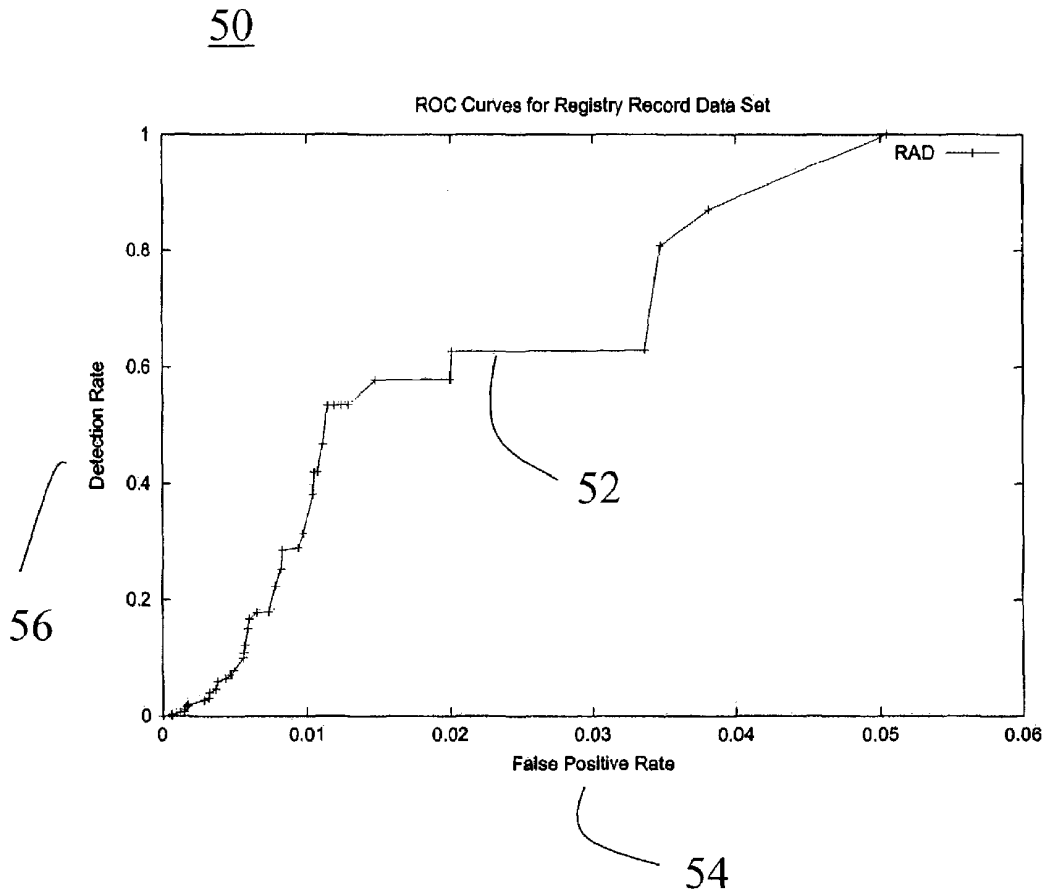


FIG. 3

US 7,448,084 B1

1

SYSTEM AND METHODS FOR DETECTING INTRUSIONS IN A COMPUTER SYSTEM BY MONITORING OPERATING SYSTEM REGISTRY ACCESSES

CLAIM FOR PRIORITY TO RELATED APPLICATIONS

This application claims the benefit of U.S. Provisional Patent Application Ser. No. 60/351,857, filed on Jan. 25, 2002, entitled "Behavior Based Anomaly Detection for Host-Based Systems for Detection of Intrusion in Computer Systems," which is hereby incorporated by reference in its entirety herein.

STATEMENT OF GOVERNMENT RIGHT

The present invention was made in part with support from United States Defense Advanced Research Projects Agency (DARPA), grant nos. FAS-526617, SRTSC-CU019-7950-1, and F30602-00-1-0603. Accordingly, the United States Government may have certain rights to this invention.

COMPUTER PROGRAM LISTING

A computer program listing is submitted in duplicate on CD. Each CD contains a routines listed in the Appendix, which CD was created on Jan. 27, 2002, and which is 22 MB in size. The files on this CD are incorporated by reference in their entirety herein.

COPYRIGHT NOTICE

A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by any one of the patent disclosure, as it appears in the Patent and Trademark Office patent files or records, but otherwise reserves all copyright rights whatsoever.

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates to systems and methods for detecting anomalies in a computer system, and more particularly to the use of probabilistic and statistical models to model the behavior of processes which access the file system of the computer, such as the Windows™ registry.

2. Background

Windows™ is currently one of the most widely used operating systems, and consequently computer systems running the Windows™ operating system are frequently subject to attacks. Malicious software is often used to perpetrate these attacks. Two conventional approaches to respond to malicious software include virus scanners, which attempt to detect the malicious software, and security patches that are created to repair the security "hole" in the operating system that the malicious software has been found to exploit. Both of these methods for protecting hosts against malicious software suffer from drawbacks. While they may be effective against known attacks, they are unable to detect and prevent new and previously unseen types of malicious software.

Many virus scanners are signature-based, which generally means that they use byte sequences or embedded strings in software to identify certain programs as malicious. If a virus scanner's signature database does not contain a signature for a malicious program, the virus scanner is unable to detect or

2

protect against that malicious program. In general, virus scanners require frequent updating of signature databases, otherwise the scanners become useless to detect new attacks. Similarly, security patches protect systems only when they have been written, distributed and applied to host systems in response to known attacks. Until then, systems remain vulnerable and attacks are potentially able to spread widely.

Frequent updates of virus scanner signature databases and security patches are necessary to protect computer systems using these approaches to defend against attacks. If these updates do not occur on a timely basis, these systems remain vulnerable to very damaging attacks caused by malicious software. Even in environments where updates are frequent and timely, the systems are inherently vulnerable from the time new malicious software is created until the software is discovered, new signatures and patches are created, and ultimately distributed to the vulnerable systems. Since malicious software may be propagated through email, the malicious software may reach the vulnerable systems long before the updates are in place.

Another approach is the use of intrusion detection systems (IDS). Host-based IDS systems monitor a host system and attempt to detect an intrusion. In an ideal case, an IDS can detect the effects or behavior of malicious software rather than distinct signatures of that software. In practice, many of the commercial IDS systems that are in widespread use are signature-based algorithms, having the drawbacks discussed above. Typically, these algorithms match host activity to a database of signatures which correspond to known attacks. This approach, like virus detection algorithms, requires previous knowledge of an attack and is rarely effective on new attacks. However, recently there has been growing interest in the use of data mining techniques, such as anomaly detection, in IDS systems. Anomaly detection algorithms may build models of normal behavior in order to detect behavior that deviates from normal behavior and which may correspond to an attack. One important advantage of anomaly detection is that it may detect new attacks, and consequently may be an effective defense against new malicious software. Anomaly detection algorithms have been applied to network intrusion detection (see, e.g., D. E. Denning, "An Intrusion Detection Model," *IEEE Transactions on Software Engineering*, SE-13: 222-232, 1987; H. S. Javitz and A. Valdes, "The NIDES Statistical Component: Description and Justification," *Technical report, SRI International*, 1993; and W. Lee, S. J. Stolfo, and K. Mok, "Data Mining in Work Flow Environments: Experiences in Intrusion Detection," *Proceedings of the 1999 Conference on Knowledge Discovery and Data Mining (KDD-99)*, 1999) and also to the analysis of system calls for host based intrusion detection (see, e.g., Stephanie Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A Sense of Self for UNIX Processes," *IEEE Computer Society*, pp. 120-128, 1996; Christina Warrender, Stephanie Forrest, and Barak Pearlmutter, "Detecting Intrusions Using System Calls: Alternative Data Models," *IEEE Computer Society*, pp. 133-145, 1999; S. A. Hofmeyr, Stephanie Forrest, and A. Somayaji, "Intrusion Detect Using Sequences of System Calls," *Journal of Computer Security*, 6:151-180, 1998; W. Lee, S. J. Stolfo, and P. K. Chan, "Learning Patterns from UNIX Processes Execution Traces for Intrusion Detection," *AAAI Press*, pp. 50-56, 1997; and Eleazar Eskin, "Anomaly Detection Over Noisy Data Using Learned Probability Distributions," *Proceedings of the Seventeenth International Conference on Machine Learning (ICML-2000)*, 2000).

There are drawbacks to the prior art approaches. For example, the system call approach to host-based intrusion detection has several disadvantages which inhibit its use in

US 7,448,084 B1

3

actual deployments. A first is that the computational overhead of monitoring all system calls is potentially very high, which may degrade the performance of a system. A second is that system calls themselves are typically irregular by nature. Consequently, it is difficult to differentiate between normal and malicious behavior, and such difficulty to differentiate behavior may result in a high false positive rate.

Accordingly, there is a need in the art for an intrusion detection system which overcomes these limitations of the prior art.

SUMMARY OF THE INVENTION

It is an object of the invention to provide techniques which are effective in detecting attacks while maintaining a low rate of false alarms.

It is another object of the invention to generate a model of the normal access to the Windows registry, and to detect anomalous accesses to the registry that are indicative of attacks.

These and other aspects of the invention are realized by a method for detecting intrusions in the operation of a computer system comprising the steps of gathering features from records of normal processes that access the Windows registry. Another step comprises generating a probabilistic model of normal computer system usage, e.g., free of attacks, based on occurrences of said features. The features of a record of a process that accesses the Windows registry are analyzed to determine whether said access to the Windows registry is an anomaly.

According to an exemplary embodiment, the step of gathering features from the records of normal processes that access the Windows registry may comprise gathering a feature corresponding to a name of a process that accesses the registry. Another feature may correspond to the type of query being sent to the registry. A further feature may correspond to an outcome of a query being sent to the registry. A still further feature may correspond to a name of a key being accessed in the registry. Yet another feature may correspond to a value of said key being accessed. In addition, any other features may be combined to detect anomalous behavior that may not be identified by analyzing a single feature.

The step of generating a probabilistic model of normal computer system usage may comprise determining a likelihood of observing a feature in the records of processes that access the Windows registry. This may comprise performing consistency checks. For example, a first order consistency check may comprise determining the probability of observation of a given feature. The step of determining a likelihood of observing a feature may comprise a second order consistency check, e.g., determining a conditional probability of observing a first feature in said records of processes that access the Windows registry given an occurrence of a second feature is said records.

In the preferred embodiment, the step of analyzing a record of a process that accesses the Windows registry may comprise, for each feature, determining if a value of the feature has been previously observed for the feature. If the value of the feature has not been observed, then a score may be determined based on a probability of observing said value of the feature. If the score is greater than a predetermined threshold, the method comprises labeling the access to the Windows registry as anomalous and labeling the process that accessed the Windows registry as malicious.

A system for detecting intrusions in the operation of a computer system has an architecture which may comprise a registry auditing module, e.g., a sensor, configured to gather

4

records regarding processes that access the Windows registry. A model generator is also provided which is configured to generate a probabilistic model of normal computer system usage based on records of a plurality of processes that access the Windows registry and that are indicative of normal computer system usage, e.g., free of attacks. A model comparator, e.g., anomaly detector, is configured to receive said probabilistic model of normal computer system usage and to receive records regarding processes that access the Windows registry and to determine whether the access of the Windows registry is an anomaly.

The system may also comprise a database configured to receive records regarding processes that access the Windows registry from said registry auditing module. The model generator may be configured to receive the records regarding processes that access the Windows registry from the database.

The model generator may be configured to determine a conditional probability of observing a first feature in records regarding processes that access the Windows registry given an occurrence of a second feature is said record. The model comparator may be configured to determine a score based on the likelihood of observing a feature in a record regarding a process that accesses the Windows registry. The model comparator may also be configured to determine whether an access to the Windows registry is anomalous based on whether the score exceeds a predetermined threshold.

In accordance with the invention, the objects as described above have been met, and the need in the art for detecting intrusions based on registry accesses has been satisfied.

BRIEF DESCRIPTION OF THE DRAWINGS

Further objects, features, and advantages of the invention will become apparent from the following detailed description taken in conjunction with the accompanying figures showing illustrative embodiments of the invention, in which:

FIG. 1 is a block diagram illustrating the architecture of the system in accordance with the present invention.

FIG. 2 is an exemplary user interface in accordance with the present invention.

FIG. 3 is a plot illustrating the result of an embodiment of the present invention.

Throughout the figures, the same reference numerals and characters, unless otherwise stated, are used to denote like features, elements, components, or portions of the illustrated embodiments. Moreover, while the subject invention will now be described in detail with reference to the figures, it is done so in connection with the illustrative embodiments without departing from the true scope and spirit of the invention as defined by the appended claims.

DETAILED DESCRIPTION OF THE EXEMPLARY EMBODIMENTS

A novel intrusion detection system **10** is disclosed herein and illustrated in FIG. 1. System **10** monitors a program's access of the file system of the computer, e.g., Microsoft™ Windows™ registry (hereinafter referred to as the "Windows™ registry" or the "registry") and determines whether the program is malicious. The system and methods described herein incorporate a novel technique referred to herein as "RAD" (Registry Anomaly Detection), which monitors the accesses to the registry, preferably in real time, and detects the actions of malicious software.

As is known in the art, the registry is an important component of the Windows™ operating system and is implemented very widely. Accordingly, a substantial amount of data

US 7,448,084 B1

5

regarding activity associated with the registry is available. The novel technique includes building a sensor, e.g., registry auditing module, on the registry and applying the information gathered by this sensor to an anomaly detector. Consequently, registry activity that corresponds to malicious software may be detected. Several advantages of monitoring the registry include the fact that registry activity is regular by nature, that the registry can be monitored with low computational overhead, and that almost all system activities query the registry.

An exemplary embodiment of the novel system, e.g., system 10, comprises several components: a registry auditing module 12, a model generator 14, and an anomaly detector 16. Generally, the sensor 12 serves to output data for each registry activity to a database 18 where it is stored for training. The model generator 14 reads data from the database 18, and creates a model of normal behavior. The model is then used by the anomaly detector 16 to decide whether each new registry access should be considered anomalous. The above components can reside on the host system itself, or on a remote network connected to the host.

A description of the Windows™ registry is provided herein. As is known in the art, the registry is a database of information about a computer's configuration. The registry contains information that is continually referenced by many different programs during the operation of the computer system. The registry may store information concerning the hardware installed on the system, the ports that are being used, profiles for each user, configuration settings for programs, and many other parameters of the system. The registry is the main storage location for all configuration information for almost all programs. The registry is also the storage location for all security information such as security policies, user names, and passwords. The registry also stores much of the important configuration information that are needed by programs in order to run.

The registry is organized hierarchically as a tree. Each entry in the registry is called a "key" and has an associated value. One example of a registry key is:

```
HKCU\Software\America Online\AOL Instant Messenger™\CurrentVersion\Users\aimuser\Login\Password
```

This example represents a key used by the AOL™ Instant Messenger™ program. This key stores an encrypted version of the password for the user name "aimuser." Upon execution, the AOL™ Instant Messenger™ program access the registry. In particular, the program queries this key in the registry in order to retrieve the stored password for the local user. Information is accessed from the registry by individual registry accesses or queries. The information associated with a registry query may include the key, the type of query, the result, the process that generated the query, and whether the query was successful. One example of a query is a read for the key, shown above. For example, the record of the query is:

```
Process: aim.exe
```

```
Query: QueryValue
```

```
Key: HKCU\Software\America Online\AOL Instant Messenger™\CurrentVersion\Users\aimuser\Login\Password
```

```
Response: SUCCESS
```

```
ResultValue: "BCOFHIIHBAHF"
```

The registry serves as an effective data source to monitor for attacks because it has been found that many attacks are manifested as anomalous registry behavior. For example, certain attacks have been found to take advantage of Windows™ reliance on the registry. Indeed, many attacks themselves rely on the registry in order to function properly. Many programs store important information in the registry, despite

6

the fact that any other program can likewise access data anywhere inside the registry. Consequently, some attacks target the registry to take advantage of this access. In order to address this concern, security permissions are included in some versions of Windows™. However, such permissions are not included in all versions of Windows™, and even when they are included, many common applications do not make use of this security feature.

"Normal" computer usage with respect to the registry is described herein. Most typical Windows™ programs access a certain set of keys during execution. Furthermore, each user typically uses a certain set of programs routinely while running their machine. This set of programs may be a set of all programs installed on the machine, or a small subset of these programs.

Another important characteristic of normal registry activity is that it has been found to be substantially regular over time. Most programs may either (1) access the registry only on startup and shutdown, or (2) access the registry at specific intervals. Since this access to the registry appears to be substantially regular, monitoring the registry for anomalous activity provides useful results because a program which substantially deviates from this normal activity may be easily detected as anomalous.

Other normal registry activity occurs only when the operating system is installed by the manufacturer. Some attacks involve launching programs that have not been launched before and/or changing keys that have not been changed since the operating system was first installed by the manufacturer.

If a model of the normal registry behavior is trained over clean data, then these kinds of registry operations will not appear in the model, and can be detected when they occur. Furthermore, malicious programs may need to query parts of the registry to get information about vulnerabilities. A malicious program can also introduce new keys that will help create vulnerabilities in the machine.

Some examples of malicious programs and how they produce anomalous registry activity are as follows:

Setup Trojan: This program, when launched, adds full read/write sharing access on the file system of the host machine. It makes use of the registry by creating a registry structure in the networking section of the Windows™ keys. The structure stems from HKLM\Software\Microsoft\Windows\CurrentVersion\Network\LanMan. It then makes several, e.g., eight, new keys for its use. It also accesses HKLM\Security\Provider in order to find information about the security of the machine to help determine vulnerabilities. This key is not accessed by any normal programs during training or testing in our experiments, and therefore its use is clearly suspicious in nature.

Back Oriffice 2000: This program opens a vulnerability on a host machine, which may grant anyone with the back oriffice client program complete control over the host machine. This program makes extensive use of the registry. In doing so, it does access a key that is very rarely accessed on the Windows™ system. This key, HKLM\Software\Microsoft\VBA\Monitors, was not accessed by any normal programs in either the training or test data. Accordingly, the detection algorithm was able to determine it as anomalous. This program also launches many other programs (e.g., LoadWC.exe, Patch.exe, runonce.exe, bo2k_i_o_intl.e) as part of the attack, in which all of these programs made anomalous accesses to the registry.

Aimrecover: This program obtains passwords from AOL™ users without authorization. It is a simple program that reads the keys from the registry where the AOL™ Instant Messenger™ program stores the user names and passwords. These

US 7,448,084 B1

7

accesses are considered anomalous because Aimrecover is accessing a key that usually is accessed and was created by a different program.

Disable Norton: This is a simple exploitation of the registry that disables Norton™ Antivirus. This attack toggles one record in the registry, in particular, the key `HKLM\SOFTWARE\INTELLANDesk\VirusProtect6\CurrentVersion\Storages\Files\System\RealTimeScan\OnOff`. If this value is set to 0, then Norton™ Antivirus real-time system monitoring is turned off. Again, this is considered anomalous because of its access to a key that was created by a different program.

L0phtCrack: This program is a widely used password cracking program for Windows™ machines. It obtains the hashed SAM file containing the passwords for all users, and then uses either a dictionary or a brute force approach to find the passwords. This program also uses flaws in the Windows™ encryption scheme in order to try to find some of the characters in a password in order to obtain a password faster. This program uses the registry by creating its own section in the registry. This new section may include many create key and set value queries, all of which will be on keys that did not exist previously on the host machine and therefore have not been seen before.

An additional aspect of normal computer usage of the registry is described herein. During testing (as will be described below) all of the programs observed in the data set cause Explorer™ to access a key specifically for that application. This key has the following format:

```
HKLM\Software\Microsoft\WindowsNT
  \CurrentVersion\Image File Execution Options\[processName]
```

where “processName” is the name of the process being run. (It is believed that all programs in general have this behavior.) This key is accessed by Explorer™ each time an application is run. Given this information, a detection system may be able to determine when new applications are run, which will be a starting point to determine malicious activity. In addition, many programs add themselves in the auto-run section of the registry under the following key:

```
HKLM\Software\Microsoft\Windows\CurrentVersion\Run.
```

While this activity is not malicious in nature, it is nevertheless an uncommon event that may suggest that a system is being attacked. Trojans such as Back Orifice utilize this part of the registry to auto load themselves on each boot.

Anomaly detectors, such as anomaly detector 16, do not operate by looking for malicious activity directly. Rather, they look for deviations from normal activity. Consequently, such deviations, which represent normal operation, may nevertheless be declared an attack by the system. For example, the installation of a new program on a system may be viewed as anomalous activity by the anomaly detector, in which new sections of the registry and many new keys may be created. This activity may interpreted as malicious (since this activity was not in the training data) and a false alarm may be triggered, much like the process of adding a new machine to a network may cause an alarm on an anomaly detector that analyzes network traffic.

There are a few possible solutions to avoid this problem. Malicious programs often install quietly so that the user does not know the program is being installed. This is not the case with most installations. For example, in an exemplary embodiment, the algorithm is programmed to ignore alarms while the install shield program is running, because the user would be aware that a new program is being installed (as opposed to malicious activity occurring without the users

8

knowledge and permission). In another embodiment, the user is prompted when a detection occurs, which provides the user with the option of informing the algorithm that the detected program is not malicious and therefore grants permission for such program to be added to the training set of data to update the anomaly detector for permissible software.

In order to detect anomalous registry accesses, model generator 14 of system 10 generates a model of normal registry activity. A set of five basic features are extracted from each registry access. (An additional five features may be added, which are combinations of the five basic features, as described below.) Statistics of the values of these features over normal data are used to create the probabilistic model of normal registry behavior. This model of normal registry behavior may include a set of consistency checks applied to the features, as will be described below. When detecting anomalies, the model of normal behavior is used to determine whether the values of the features of the new registry accesses are consistent with the normal data. If such values are not consistent, the algorithm labels the registry access as anomalous, and the processes that accessed the registry as malicious.

In the exemplary embodiment, the data model consists of five basic features gathered by the registry auditing module 12 from an audit stream. (It is contemplated that additional features may also provide significant results.) In the exemplary embodiment, the features are as follows:

Process: The name of the process accessing the registry. This allows the tracking of new processes that did not appear in the training data.

Query: The type of query being sent to the registry, for example, QueryValue, CreateKey, and SetValue are valid query types. This allows the identification of query types that have not been seen before. There are many query types but only a few are used under normal circumstances.

Key: The actual key being accessed. Including this feature allows the algorithm to locate keys that are never accessed in the training data. Many keys are used only once for special situations like system installation. Some of these keys can be used by attacks to create vulnerabilities.

Response: A feature which describes the outcome of the query, for example, success, not found, no more, buffer overflow, and access denied.

Result Value: The value of the key being accessed. Including this feature allows the algorithm to detect abnormal values being used to create abnormal behavior in the system.

Composite features may also be used, which are a combination of two basic features, such as those discussed above. These composite features are useful in detecting anomalies since they allow the system to give more detail to its normal usage model where the existence of the basic features in isolation would not necessary be detected as anomalous. The following is a list of exemplary composite fields that may be used by system 10:

Process/Query: This key is a combination of the process and query fields, and may provide information to determine if a particular process is executing queries that are not typically executed by this process.

Key/Process: This key is a combination of the key and process fields, and may allow the algorithm to detect whether or not a process accesses portions of the registry it typically doesn't access. Also it detects when a key is being accessed by a process that normally doesn't access it. This analysis is very useful because many processes access keys belonging to other programs, and this field would allow the detection of such an event.

Query/Key: This key is a combination of the query and key fields, and may determine if a key is being accessed in a

US 7,448,084 B1

9

different way than usual. For example, there are many key values in the registry that are written once at the time of their creation, and then are subsequently read from the registry without change. Some of these keys store crucial information for the execution of certain programs. If a malicious process were to write to one of these keys, after the time of their creation, this field would enable the algorithm to detect such an event, and the record would appear anomalous.

Response/Key: This key is a combination of the key and response fields. Many keys are found to be used in the same manner each time they are used. Consequently, they will always return the same response. Since a different response to a key may be indicative of abnormal or malicious system activity, the algorithm which utilizes this combination key would be able to detect such abnormal or malicious activity.

Result Value/Key: This key is a combination of the key and result value. During operation of the computer system, many keys will always contain a value from a certain set or range of values. When the value is outside that range, that could be an indicator of a malicious program taking advantage of a vulnerability.

As an illustration, an exemplary registry access is displayed in Table 1. The second column is normal access by the process aim.exe (which is used with AOL™ Instant Messenger™) to access the key where passwords are stored. The third column of Table 1 is a malicious access by a process aimrecover.exe to the same key. The final column of Table 1 shows which fields register by the anomaly detector as anomalous. As seen in the table, all of the basic features, e.g., process, query, key, response, and result value, do not appear anomalous for the normal process aim.exe, when compared with the malicious process aimrecover.exe. However, the composite keys are useful for detecting the anomalous behavior of aimrecover.exe. For example, the fact that the process aimrecover.exe is accessing a key that is usually associated with another process, i.e., aim.exe, is detected as an anomaly. This conclusion is made because under normal circumstances only aim.exe accesses the key that stores the AOL™ Instant Messenger™ password. The occurrence of another process accessing this key is considered suspicious. By examining the combination of two basic features, the algorithm can detect this anomaly.

TABLE 1

Feature	aim.exe	aimrecover.exe	Anomalous
Process	aim.exe	aimrecover.exe	no
Query	QueryValue	QueryValue	no
Key	HKCU\Software\America Online\AOL Instant Messenger (TM)\CurrentVersion\Users\aimuser>Login\Password	HKCU\Software\America Online\AOL Instant Messenger (TM)\CurrentVersion\Users\aimuser>Login\Password	no
Response	SUCCESS	SUCCESS	no
Result Value	“BCOFHIHBBAHF”	“BCOFHIHBBAHF”	no
Process/Query	aim.exe:QueryValue	aimrecover:QueryValue	no
Query/Key	QueryValue:HKCU\Software\America Online\AOL Instant Messenger (TM)\CurrentVersion\Users\aimuser>Login\Password	QueryValue:HKCU\Software\America Online\AOL Instant Messenger (TM)\CurrentVersion\Users\aimuser>Login\Password	no
Response/Key	SUCCESS:HKCU\Software\America Online\AOL Instant Messenger (TM)\CurrentVersion\Users\aimuser>Login\Password	SUCCESS:HKCU\Software\America Online\AOL Instant Messenger (TM)\CurrentVersion\Users\aimuser>Login\Password	no
Process/Key	aim.exe:HKCU\Software\America Online\AOL Instant Messenger (TM)\CurrentVersion\Users\aimuser>Login\Password	aimrecover.exe:HKCU\Software\America Online\AOL Instant Messenger (TM)\CurrentVersion\Users\aimuser>Login\Password	yes

10

Exemplary embodiments of intrusion detection algorithms which may be used by the system 10 will now be described, although it is understood that other anomaly detection algorithms may also be used in connection with the present invention. Since a significant amount of data is monitored in real time, the algorithm that is selected must be very efficient. According to a first exemplary embodiment, the features that were monitored from each registry access are used to train a model over features extracted from normal data. That model allows for the classification of registry accesses as either normal or malicious, as will be described herein.

In general, a principled probabilistic approach to anomaly detection can be reduced to density estimation. If a density function p(x) can be estimated over the normal data, anomalies are defined as data elements that occur with low probability. In practice, estimating densities is a very complex, non-trivial problem. In detecting intrusions into the registry, a complication is that each of the features have many possible values. For example, the key feature, defined above, may have over 30,000 values in the training set. Since there are so many possible feature values, it is relatively rare that the same exact record occurs more than once in the data. Data sets of this type are referred to as “sparse.”

Since probability density estimation is a very complex problem over sparse data, the method of the present invention defines a set of consistency checks over the normal data for determining which records from a sparse data set are anomalous. Each consistency check is applied to an observed record by the anomaly detector. If the record fails any consistency check, the record is labeled as anomalous.

In the exemplary embodiment, two kinds of consistency checks are applied. The first consistency check evaluates whether or not a feature value is consistent with observed values of that feature in the normal data set. This type of consistency check is referred to as a first order consistency check, e.g., each registry record may be viewed as the outcome of five random variables, one for each feature, X₁, X₂, X₃, X₄, X₅. The consistency checks compute the likelihood of an observation of a given feature denoted as P(X_i).

US 7,448,084 B1

11

The second consistency check handles pairs of features, as discussed in the example in Table 1. For each pair of features, the conditional probability of a feature value given another feature value is considered. These consistency checks are referred to as second order consistency checks. These likelihoods are denoted as $P(X_i|X_j)$. For each value of X_j , there may be a different probability distribution over X_i .

In the exemplary embodiment, since there are five basic feature values, for each record, there are five first order consistency checks and 20 second order consistency checks of which five examples are given above. If the likelihood of any of the consistency checks is below a threshold, the record is labeled as anomalous. The determination of the threshold is described in greater detail below.

The manner in which the likelihoods for the first order ($P(X_i)$) consistency checks and the second order ($P(X_i|X_j)$) consistency checks are computed is described herein. From the normal data, there is a set of observed counts from a discrete alphabet, e.g., a finite number of distinct symbols or feature values, for each of the consistency checks. Computing the above likelihoods reduces to estimating a multinomial expression. In theory, the maximum likelihood estimate may be used, which computes the ratio of the counts of a particular element to the total counts. However, the maximum likelihood estimate has been found to be biased when relatively small amounts of data, e.g., "sparse data," are available. The distribution may be smoothed by adding a virtual count to each possible element. For anomaly detection, it is often desirable to take into account how likely it is to observe a previously unobserved element. Thus, if many different elements have been seen in the training data, it is therefore more likely to see additional, unobserved elements, as opposed to the case where very few elements have been seen, in which additional, unobserved elements would be unlikely. (The term "element" here refers to feature values, or a vector of feature values.)

To estimate the likelihoods, an estimator is used, which gives the following prediction for element i :

$$P(X = i) = \frac{\alpha + N_i}{k^0 \alpha + N} C \quad (1)$$

if element i was observed in the training data. If element i was not previously observed, then the following prediction is used:

$$P(X = i) = \frac{1}{L - k^0} (1 - C) \quad (2)$$

In these equations, the term α is a prior count for each element. The term N_i is the number of times element i was observed; N is the total number of observations, k^0 is the number of different elements observed, and L is the total number of possible elements or the alphabet size. The parameters are either observed or computed (e.g., N and k^0 are determined by storing values and computing frequency counts) while L is defined by the particular system being modeled, i.e., the type of variables (e.g., 32 bit integers) defines a range of possible values. Thus, L is predefined by the implementation details of the system. Here, the system is modeling the Windows™ registry, which has a predefined range of possible values (as defined by the programmers of the registry.) The scaling factor C takes into account how

12

likely it is to observe a previously observed element versus an unobserved element. C is computed by the following equation:

$$C = \left(\sum_{k=k^0}^L \frac{k^0 \alpha + N}{k \alpha + N} m_k \right) \left(\sum_{k \geq k^0} m_k \right)^{-1} \quad (3)$$

where

$$m_k = P(S = k) \frac{k!}{(k - k^0)!} \frac{\Gamma(k \alpha)}{\Gamma(k \alpha + N)}$$

and $P(S=k)$ is a prior probability associated with the size of the subset of elements in the alphabet that have non-zero probability. Although the computation of C is expensive, it only needs to be done once for each consistency check at the end of training. Second order consistency checks are done in like fashion, except the particular values being measured are not distinct features values, but pairs of feature values, considering these pairs as a distinct element.

The prediction of the probability estimator is derived using a mixture of Dirichlet estimators, as are known in the art, see, e.g., the estimator presented in N. Friedman and Y. Singer, "Efficient Bayesian Parameter Estimation in Large Discrete Domains," *Advances in Neural Information Processing Systems* 11, MIT Press, which is incorporated by reference in its entirety herein.) The scores computed in the attached software code correspond to the estimates provided by the consistency checks in equations (1) and (2) above.

This exemplary embodiment of the algorithm labels every registry access as either normal or anomalous. Programs can have anywhere from just a few registry accesses to several thousand. This means that many attacks will be represented by large numbers of records where many of those records will be considered anomalous.

A second exemplary embodiment of the algorithm is described herein. Using the features that are monitored from each registry access, a score is computed to classify each access as either normal or malicious. A set of normal registry accesses are analyzed as a model of normal usage of the registry. Then using this model, new registry records are analyzed to determine whether or not they are malicious.

As the data is being collected, several important statistics are collected about each feature and the values that occur for each feature. For each feature, which values occurred for that feature and how many distinct values occurred for the feature, r , are recorded. Accordingly, r is a measure of how likely it is to see a new value for the feature. If many distinct values for a feature have been previously observed, i.e., a high value for r , and subsequently a never-observed value is encountered, such new value would be expected and considered normal. In contrast, if only a few distinct values have been observed, i.e., a low value for r , the observation of a new value is unlikely and possibly anomalous. The total number of registry different elements, e.g., training records, that are observed during training, n , is also recorded.

During training, for each of the features, all of the distinct observed values of the feature are stored, as well as the number of distinct observed values r . The total number of training records n is computed. For each feature, the algorithm computes $p=r/n$, which is an approximation of the probability of

US 7,448,084 B1

13

observing an unseen value for that feature in the normal data. To minimize storage requirements, instead of storing all distinct values, the algorithm hashes all of the values and stores the hashes of the values in a bit vector. More details on this technique is described in the implementation of PHAD (Packet Header Anomaly Detection), an anomaly detection algorithm known in the art, which was developed to detect anomalies in packet headers (see, e.g., M. Mahoney and P. Chan, "Detecting Novel Attacks by Identifying Anomalous Network Packet Headers," *Technical Report CS-2001-2*, Florida Institute of Technology, Melbourne, Fla., 2001).

Once the model has been trained, new registry accesses can be evaluated and a score computed to determine whether or not the registry accesses are abnormal. For a new registry access, we first extract the features for the registry access. For each of these features, a check is performed to see if the value of the feature has been observed for the feature. If the value has not been observed, a heuristic score is computed which determines the level of anomaly for that feature. The score is determined as $1/p$ for each feature. Intuitively this score will be higher for features where fewer distinct values have been observed. The final score for a registry access is the sum of the scores for each feature that observed a previously unobserved value. If this value is greater than a threshold, we label the registry access anomalous and declare the process that generated it as malicious. The results from this experiment are described below.

The basic architecture of the system **10** will now be discussed in greater detail herein. With continued reference to FIG. **1**, the registry auditing module **12** monitors accesses to the registry. In the exemplary embodiment, the registry auditing module **12** is a "Basic Auditing Module" (BAM). In general, BAMs are known in the art, and implement an architecture and interface which provide a consistent data representation for a sensor. As indicated by arrow **22**, they include a "hook" into the audit stream (in this case the registry) and various communication and data-buffering components. BAMs use an XML data representation substantially identical to the IETF standard for IDS systems (See, e.g., Internet Engineering Task Force. Intrusion detection exchange format. On-line publication, <http://www.ietf.org/html.charters/idwg-charter.html>, 2000.), minor syntactical differences. The registry auditing module **12** runs in the background on a Windows™ machine, where it gathers information on registry reads and writes, e.g., the 5 features discussed above. Registry auditing module **12** uses Win32 hooks to tap into the registry and log all reads and writes to the registry. The software uses an architecture substantially identical to SysInternal's Regmon (See, e.g., SysInternals. Regmon for Windows™ NT/9x. *Online publication*, 2000. <http://www.sysinternals.com/ntw2k/source/regmon.shtml>), and extracts a subset of data available to Regmon. After gathering the registry data, registry auditing module **12** can be configured for two distinct uses. One use is to act as the data source for model generation. When registry auditing module **12** is used as the data source for model generation, its output is sent to a database **18** (as indicated by arrow **24**) where it is stored and later used by the model generator **16** described herein. The second use of registry auditing module **12** is to act as the data source for the real-time anomaly detector **14** described herein. While in this mode, the output of registry auditing module **12** is sent directly to the anomaly detector **14** (indicated by arrow **26**) where it is processed in real time.

The model generation infrastructure consists of two components. A database **18** is used to store all of the collected registry accesses from the training data. A model generator **14** then uses this collected data to create a model of normal

14

usage. The model generator **14** uses one of the two exemplary algorithms discussed above (or other similar algorithm) to build a model that will represent normal usage. It utilizes the data stored in the database **18** which was generated by registry auditing module **12**. The database **18** is described in greater detail is concurrently filed U.S. application Ser. No. 10/352,342, entitled "System and Methods for Adaptive Model Generation for Detecting Intrusion in Computer Systems," to Andrew Honig, et al., which is incorporated by reference in its entirety herein. (Arrow **28** indicates the flow of data from the data warehouse **18** to the model generator **14**.) The model itself is comprised of serialized Java objects. This allows for a single model to be generated and to easily be distributed to additional machines. Having the model easily deployed to new machines is a desirable feature since in a typical network, many Windows™ machines have similar usage patterns which allow for the same model to be used for multiple machines. The GUI **30** for the model generator using the second embodiment of the algorithm is shown in FIG. **2**. Column **32** indicates the feature name, column **34** indicates the n-value, column **36** indicates the r-value, and column **38** indicates the p-value. Additional details for generating a model are described in U.S. application Ser. No. 10/208,432 filed Jul. 30, 2002 entitled "System and Methods for Detection of New Malicious Executables," to Matthew G. Schulz et al., which is incorporated by reference in its entirety herein.

The anomaly detector **16** will load the normal usage model created by the model generator **14** (as indicated by arrow **29**) and begin reading each record from the output data stream of registry auditing module **12** (arrow **26**). One of the algorithms, as discussed above, is then applied against each record of registry activity. The score generated by the anomaly detection algorithm is then compared by a user configurable threshold to determine if the record should be considered anomalous. A list of anomalous registry accesses are stored and displayed as part of the detector.

The system described herein is a statistical model of expected registry queries and results. If an attacker wanted to usurp a host-based detector, they can a) turn off the detector at the host (and hope no alarms go off elsewhere) or b) they can attack the host based detector by changing its rules or changing its statistical model so it won't alarm.

Accordingly, in order to protect the statistical model of the system, from being attacked, it is put in the registry. The registry is essentially a data base, and the statistical model comprises query values and probabilities. The evaluation of the model first accesses values and probability estimates. This information can be stored in the registry. Hence, any process that attempts to touch the model (for example, to change some values in the model) will be abnormal registry accesses and set off the alarm. Consequently, the system would be protected from having its own model being attacked since it will notice when it is under attack.

In order to evaluate the system, data was gathered by running a registry auditing module **12** on a host machine. During training, several programs were run in order to generate normal background traffic. In order to generate normal data for building an accurate and complete training model, it was important to run various applications in various ways. By examining registry traffic, it was discovered that it is not just which programs that are run, but also how they are run that affect registry activity. For example, running ping.exe from the command prompt does not generate registry activity. However, running ping.exe directly from the run dialog box does generate registry activity: By understanding such details of the registry, a more complete training model was built. Beyond the normal execution of standard programs, such as

US 7,448,084 B1

15

Microsoft™ Word, Internet Explorer, and Winzip, the training also included performing tasks such as emptying the Recycling Bin and using Control Panel.

The training data collected for the experiment was collected on Windows™ NT 4.0 over two days of normal usage. “Normal” usage is defined to mean what is believed to be typical use of a Windows™ platform in a home setting. For example, it was assumed that users would log in, check some internet sites, read some mail, use word processing, then log off. This type of session was taken to be relatively typical of computer users. Normal programs are those which are bundled with the operating systems, or are in use by most Windows™ users.

The simulated home use of Windows™ generated a clean (attack-free) dataset of approximately 500,000 records. The system was tested on a full day of test data with embedded attacks executed. This data was comprised of approximately 300,000 records, most of which were normal program executions interspersed with attacks among normal process executions. The normal programs run between attacks were intended to simulate an ordinary Windows™ session. The programs used were, for example, Microsoft™ Word, Outlook Express™, Internet Explorer™, Netscape™, AOL™ Instant Messenger™.

The attacks run include publicly available attacks such as aimrecover, browslist, bok2ss (back orifice), install.exe xtcp and exe (both for backdoor.XTCP), 10phtcrack, runtack, whackmole, and setuptrojan. Attacks were only run during the one day of testing throughout the day. Among the twelve attacks that were run, four instances were repetitions of the same attack. Since some attacks generated multiple processes there are a total of seventeen distinct processes for each attack. All of the processes (either attack or normal) as well as the number of registry access records in the test data is shown in Table 3 and described in greater detail herein.

The training and testing environments were set up to replicate a simple yet realistic model of usage of Windows™ systems. The system load and the applications that were run were meant to resemble what one may deem typical in normal private settings.

The first exemplary anomaly detection algorithm discussed above in equations (1)-(3) were trained over the normal data. Each record in the testing set was evaluated against this training data. The results were evaluated by computing two statistics: the detection rate and the false positive rate. The performance of the system was evaluated by measuring detection performance over processes labeled as either normal or malicious.

The detection rate reported below is the percentage of records generated by the malicious programs which are labeled correctly as anomalous by the model. The false positive rate is the percentage of normal records which are mislabeled anomalous. Each attack or normal process has many records associated with it. Therefore, it is possible that some records generated by a malicious program will be mislabeled even when some of the records generated by the attack are accurately detected. This will occur in the event that some of the records associated with one attack are labeled normal. Each record is given an anomaly score, S, that is compared to a user defined threshold. If the score is greater than the threshold, then that particular record is considered malicious. FIG. 3 shows how varying the threshold affects the output of detector. The actual recorded scores plotted in the figure are displayed in Table 2.

16

TABLE 2

	Threshold Score	False Positive Rate	Detection Rate
5	6.847393	0.001192	0.005870
	6.165698	0.002826	0.027215
	5.971925	0.003159	0.030416
	5.432488	0.004294	0.064034
	4.828566	0.005613	0.099253
	4.565011	0.006506	0.177161
10	3.812506	0.009343	0.288687
	3.774119	0.009738	0.314301
	3.502904	0.011392	0.533084
	3.231236	0.012790	0.535219
	3.158004	0.014740	0.577908
	2.915094	0.019998	0.578442
15	2.899837	0.020087	0.627001
	2.753176	0.033658	0.629136
	2.584921	0.034744	0.808431
	2.531572	0.038042	0.869797
	2.384402	0.050454	1.000000

Table 3 is sorted in order to show the results for classifying processes. Information about all processes in testing data including the number of registry accesses and the maximum and minimum score for each record as well as the classification. The top part of the table shows this information for all of the attack processes and the bottom part of the table shows this information for the normal processes. The reference number (by the attack processes) give the source for the attack. Processes that have the same reference number are part of the same attack. [1] AIMCrack. [2] Back Orifice. [3] Backdoor.xtcp. [4] Browse List. [5] Happy 99. [6] IPCrack. [7] LOpht Crack. [8] Setup Trojan.

TABLE 3

Program Name	Number of Records	Maximum Record Value	Minimum Record Value	Classification
LOADWC.EXE[2]	1	8.497072	8.497072	ATTACK
ipccrack.exe[6]	1	8.497072	8.497072	ATTACK
mstinit.exe[2]	11	7.253687	6.705313	ATTACK
bo2kss.exe[2]	12	7.253687	6.62527	ATTACK
runonce.exe[2]	8	7.253384	6.992995	ATTACK
browslist.exe[4]	32	6.807137	5.693712	ATTACK
install.exe[3]	18	6.519455	6.24578	ATTACK
SetupTrojan.exe[8]	30	6.444089	5.756232	ATTACK
AimRecover.exe[1]	61	6.444089	5.063085	ATTACK
happy99.exe[5]	29	5.918383	5.789022	ATTACK
bo2k.1..0.intl.e[2]	78	5.432488	4.820771	ATTACK
..INS0432...MP[2]	443	5.284697	3.094395	ATTACK
xtcp.exe[3]	240	5.265434	3.705422	ATTACK
bo2kefg.exe[2]	289	4.879232	3.520338	ATTACK
10phtcrack.exe[7]	100	4.688737	4.575099	ATTACK
Patch.exe[2]	174	4.661701	4.025433	ATTACK
bo2k.exe[2]	883	4.386504	2.405762	ATTACK
sstray.exe	17	7.253687	6.299848	NORMAL
CSRSS.EXE	63	7.253687	5.031336	NORMAL
SPOOLSS.EXE	72	7.070537	5.133161	NORMAL
ttash.exe	12	6.62527	6.62527	NORMAL
winmine.exe	21	6.56054	6.099177	NORMAL
em...exec.exe	29	6.337396	5.789022	NORMAL
winampa.exe	547	6.11399	2.883944	NORMAL
PINBALL.EXE	240	5.898464	3.705422	NORMAL
LSASS.EXE	2299	5.432488	1.449555	NORMAL
PING.EXE	50	5.345477	5.258394	NORMAL
EXCEL.EXE	1782	5.284697	1.704167	NORMAL
WINLOGON.EXE	399	5.191326	3.198755	NORMAL
rundl132.exe	142	5.057795	4.227375	NORMAL
explore.exe	108	4.960194	4.498871	NORMAL
netscape.exe	11252	4.828566	-0.138171	NORMAL
java.exe	42	4.828566	3.774119	NORMAL
aim.exe	1702	4.828566	1.750073	NORMAL
findfast.exe	176	4.679733	4.01407	NORMAL

US 7,448,084 B1

17

TABLE 3-continued

Program Name	Number of Records	Maximum Record Value	Minimum Record Value	Classification
TASKMGR.EXE	99	4.650997	4.585049	NORMAL
MSACCESS.EXE	2825	4.629494	1.243602	NORMAL
IEXPLORE.EXE	194274	4.628190	-3.419214	NORMAL
NTVDM.EXE	271	4.59155	3.584417	NORMAL
CMD.EXE	116	4.579538	4.428045	NORMAL
WINWORD.EXE	1541	4.457119	1.7081	NORMAL
EXPLORER.EXE	53894	4.31774	-1.704574	NORMAL
mmsgs.exe	7016	4.177509	0.334128	NORMAL
OSA9.EXE	705	4.163361	2.584921	NORMAL
MYCOME 1.EXE	1193	4.035649	2.105155	NORMAL
wscript.exe	527	3.883216	2.921123	NORMAL
WINZIP32.EXE	3043	3.883216	0.593845	NORMAL
notepad.exe	2673	3.883216	1.264339	NORMAL
POWERPNT.EXE	617	3.501078	-0.145078	NORMAL
AcroRd32.exe	1598	3.412895	0.393729	NORMAL
MDM.EXE	1825	3.231236	1.680336	NORMAL
ttermpro.exe	1639	2.899837	1.787768	NORMAL
SERVICES.EXE	1070	2.576196	2.213871	NORMAL
REGMON.EXE	259	2.556836	1.205416	NORMAL
RPCSS.EXE	4349	2.250997	0.812288	NORMAL

The process of setting the threshold is described herein. If the threshold is set at 8.497072, the processes LOADWC.EXE and ipccrack.exe are labeled as malicious and would detect the Back Orifice and IPCrack attacks. Since none of the normal processes have scores that high, we would have no false positives. If we lower the threshold to 6.444089, we would have detected several more processes from Back Orifice and the BrowseL.ist, BackDoor.xtcp, SetupTrojan and AimRecover attacks. However, at this level of threshold, the following processes would be labeled as false positives: systray.exe, CSRSS.EXE, SPOOLSS.EXE, ttssh.exe, and winmine.exe. [

By varying the threshold for the inconsistency scores on records, we were able to demonstrate the variability of the detection rate and false positive rate. The false positive rate versus the detection rate was plotted in an ROC (Receiver Operator Characteristic) curve 52 shown in Table 2 and the plot 50 in FIG. 3, in which the false positive rate 54 is plotted against the detection rate 56.

Another exemplary embodiment is described herein. The systems and method described above, uses Windows™ registry accesses, which is an example of a general means of detecting malicious uses of a host computer. However, other systems, such as Linux/Unix, do not use a registry. In those cases, a file system sensor would be used. Accesses to the file system provides an audit source of data (i.e., whenever an application is run, any and all things accessed are files, e.g., the main executable files are accessed, and the project files are accessed). This audit source can be observed, and a “normal baseline model” built, and then used for detecting abnormal file system accesses.

It will be understood that the foregoing is only illustrative of the principles of the invention, and that various modifications can be made by those skilled in the art without departing from the scope and spirit of the invention.

APPENDIX

The software listed herein is provided in an attached CD-Rom. The contents of the CD-Rom are incorporated by reference in their entirety herein.

A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copy-

18

right owner has no objection to the facsimile reproduction by any one of the patent disclosure, as it appears in the Patent and Trademark Office patent files or records, but otherwise reserves all copyright rights whatsoever.

5 PAD (Probabilistic Anomaly Detection) is a package that detects anomalies in fixed length records of discrete values.

The basic idea behind PAD is that it trains over a data set and then checks to see if new observed records are “consistent” with the data set. There are two types of consistency checks. First order consistency checks evaluate whether or not a single feature is consistent with other features in the dataset. Second order consistency checks evaluate if a pair of features is consistent with the data set. All consistency checks are evaluated by computing a predictive probability. This is done by estimating a multinomial using counts observed from the data set and then estimating the probability of seeing the observation.

Let us assume records of length n are being observed. That is, each record can be written as a set of random variables (X_1, X_2, \dots, X_n) . Each first order consistency check can be denoted as computing $P(X_i)$. This probability is trained over a data set, and then used to predict elements in another dataset. These two datasets can be in fact the same. To train the probability distribution, the counts of the observed symbols in the data set are collected. The second order consistency checks are $P(X_i, X_j)$. In this case, the counts of X_i are collected when X_j is observed. Note that there is a separate set of counts for each X_j . Accordingly, second order consistency checks take a significant amount of memory relative to first order consistency checks.

All of these probability estimates are obtained using the multinomial estimator presented in Friedman, Singer 1999 (incorporated by reference, above). The basic idea of the estimator is that it explicitly takes into account the probability of seeing an unobserved symbol.

The term c is the probability of observing an already observed symbol. Thus $(1-c)$ is the probability of observing an unobserved symbol. For the observed symbols, a Dirichlet is used to estimate the probabilities for the counts. If N is the total number of observations, and N_i is the observations of symbol i , if α is the “pseudo count” which is added to the count of each observed symbol, and k^0 is the number of observed symbols and L is the total number of symbols, the probability is as follows:

For an observed symbol i , the probability is: $C \cdot ((N_i + \alpha) / (k^0 \cdot \alpha + N))$. For an unobserved symbol, the probability is: $(1-C) \cdot (1 / (L - k^0))$. See equations (1) and (2) above, and the routine “updateC” in Classifier.c. In the second case, the probability is spread among possible unobserved symbols.

Since each feature in the record may have a different set of possible outcomes, if P is the probability estimated from the consistency check, the following term is reported: $\log(P / (1/L))$. This normalizes the consistency check to take into account the number of possible outcomes L . In general, c should be set as described in Friedman, Singer 1999.

However, this causes some overflow/underflow problems in the general case. Instead, the current version of the algorithm uses a heuristic to set c . This is done after observing the counts. This is called SIMPLE_MODE in the implementation. In SIMPLE_MODE, C is set to $C = N / (N + L - k^0)$. In addition, in SIMPLE_MODE, there is a variable OBSERVED_BIAS which adjusts the value of c toward observed or unobserved symbols. When OBSERVED_BIAS=0.0, there is no bias. Positive values increase the probability mass of observed symbols while negative values decrease the probability mass of unobserved symbols. For

US 7,448,084 B1

19

non-zero OBSERVED_BIAS, the value of c is adjusted so that the new value of c, c* is given as follows $C^*=C/(C+(1-C)*\exp(-OBSERVED_BIAS))$.

Package Installation and Quick Start: To install the package, the following steps should be performed:

1. Unpack the files.
2. cd into the src/ subdirectory.
3. type make
To test the package, the following steps should be performed:
4. cd into data/ subdirectory
5. type ./src/pad -e -g globalsFile.txt -p sampleInput.txt sampleInput.txt

Subsequently, the following output should be provided:

```
a aa aaa zzzz: 0.031253 1.386294 0.490206 0.693147
0.000000 0.031253 0.000000 0.000000 0.000000 1.386294
0.980829 0.980829 0.000000 0.490206 0.693147 0.693147
0.000000 0.693147 0.875469 0.875469: test
```

```
b aa fff dddd: 0.436718 1.386294 0.202524 0.000000
0.619039 0.436718 0.000000 0.000000 0.632523 1.386294
0.632523 0.000000 0.000000 0.202524 0.470004 0.000000
-0.000000 0.000000 -0.000000 0.000000: test1
```

```
c aa fffttt: 0.031253 1.386294 0.202524 0.000000 0.000000
0.031253 0.000000 0.000000 0.000000 1.386294 0.632523
0.000000 0.000000 0.202524 0.470004 0.000000 0.000000
0.000000-0.000000 0.000000: test2
```

```
g aa aaa zzzz: 0.031253 1.386294 0.490206 0.693147
0.000000 0.031253 0.000000 0.000000 0.000000 1.386294
0.980829 0.980829 0.000000 0.490206 0.693147 0.693147
0.000000 0.693147 0.875469 0.875469: test3
```

```
b aa aaa zzzz: 0.436718 1.386294 0.490206 0.693147
0.619039 0.436718 0.000000 0.000000 0.632523 1.386294
0.980829 0.980829 0.000000 0.490206 0.693147 0.693147
-0.000000 0.693147 0.875469 0.875469: test4
```

The next steps are then performed:

6. Type ./src/pad -e -g globalsFile.txt -w temp.cla sampleInput.txt

This should create a file called temp.cla which is the trained model (classifier) from the sample input.

7. Type ./src/pad -r -p sampleInput.txt temp.cla

This should provide the same output as above.

Usage Instructions: The executable has two modes: “examples” mode (using the -e option) which reads in examples from a file and trains the model using that data, and “read” mode (using the -r option) which reads in a model from a file. The executable requires one argument, which is either the file or examples. The globals file (specified with the -g option) defines all of the global variables. These include the number of columns, the file names containing the column symbol definitions. Note that when reading in a model, the column symbol files must be in the current directory.

Options: Command line options: In addition to -r and -e which set the mode, the following are options that can be used from the command line:

-g FILE	set globals file
-v	toggle verbose output

20

-continued

-s	toggle use of second order predictors
-w FILE	write classifier to file.
-p FILE	predict files

Globals File Options: Below is the globals file included in the distribution. All lines starting with “#” are comments.

```
10 # Globals Definition
#The input Symbols.
NUM_COLUMNS 4
15 #Set Simple Mode
SIMPLE_MODE TRUE
#Set Use Second Order Consistency Checks
USE_SECOND_ORDER TRUE
20 #Set Verbose mode
VERBOSE FALSE
#Allow unknown symbols in testing
25 ALLOW_UNKNOWN_SYMBOLS TRUE
#Set Initial Count for Predictors. This is the virtual count
#that is added to all observed symbols.
30 INITIAL_PREDICTION_COUNT 1
#Set the bias to observed symbols
OBSERVED_BIAS 0.0
#Set the Column Symbol Files
35 COLUMN_SYMBOL_FILENAME 1:C1.txt
COLUMN_SYMBOL_FILENAME 2:C2.txt
COLUMN_SYMBOL_FILENAME 3:C3.txt
40 COLUMN_SYMBOL_FILENAME 4:C4.txt
#Sets the number of symbols in a column
COLUMN_NUM_SYMBOLS 1:40
#Sets the classifier to ignore a column
45 IGNORE_COLUMN 2
#Sets the symbol that represents an ignored symbol
IGNORE_SYMBOL **
50 #Sets the symbol to represent an unknown symbol
UNKNOWN_SYMBOL UKS
```

Input file description: Each line of the input file corresponds to a record. Each record consists of the features in a record separated by a space. This is followed by a tab after which there is an optional comment. This comment is preserved in prediction and can be used in experiments to keep track of the type of a record and where it came from. Below is the sample input:

a	aa	aaa	zzzz	test
b	aa	fff	dddd	test1
c	aa	fff	tttt	test2
g	aa	aaa	zzzz	test3
b	aa	aaa	zzzz	test4

US 7,448,084 B1

21

A symbol file defines all of the possible symbols. Each symbol is on a separate line in the file. A sample symbol file is below:

aaa
ccc
fff
ggg

There are several options related to symbol files. The IGNORE_COLUMN can set the classifier to ignore a column completely. In this case, each element of the column gets mapped to a special symbol which is ignored in the model. In a single record, some of the fields can be set to a special symbol (by default "***") which tells the classifier to ignore its value. A special symbol (by default "UKS") denotes an unknown symbol. In training, unknown symbols are not allowed and will cause the program to exit. In testing, the unknown symbols are treated as if it is a symbol that has observed count of 0. The option ALLOW_UNKNOWN_SYMBOLS toggles the automatic mapping of unseen symbols to the special unknown symbol during testing. This makes running experiments easier because it is not necessary to have the symbol files contain the data in the test set.

Package Description: The software package contains the following:

/src/	directory consisting of all of the source files
/data/	directory consisting of a small sample input
/registry/	directory consisting of data and scripts to do the registry experiments.
/papers/	directory containing relevant papers to pad.

In the /src/ directory there are the following files:

Classifier.c	File that defines the Classifier (model)
Classifier.h	Header file for Classifier.c
Column.c	File that defines a single consistency check
Column.h	Header file for Column.c
Globals.c	Defines the global variables
Globals.h	Header file for Globals.c
HashTable.c	Hashtable implementation
HashTable.h	Header file for HashTable.c
includes.h	Include file for all files
Input.c	Implementation of I/O
Input.h	Header file for Input.c
Makefile	Makefile
memwatch.c	Package to detect memory leaks
memwatch.h	Package to detect memory leaks
pad.c	Main executable file
pad.h	Header file for pad.c
SymbolTable.c	Symbol Table implementation for mapping Symbols to Integers
SymbolTable.h	Header file for SymbolTable.c

Registry Experiments: The following steps should be performed:

1. cd registry/
 2. type ./src/pad -e -g regGlobs.txt -w model.cla registry.txt
- This creates a file called model.cla which is the model that is trained on 800k registry records. It should take about 190 MB of memory to train the model.

22

3. type ./src/pad -r -p registry.txt model.cla >predictions.txt
This reads in the model and evaluates all of the records. The file predictions.txt contains the values for all of the consistency checks.

4. Type ./computeResults.pl predictions.txt >final-predictions.txt

This determines the minimum value for a consistency check for each record and puts on each line this value and the comment.

5. Typesort -n final-predictions.txt >sorted-final-predictions.txt

This sorts the records in order of least consistent. This is the final output of the experiments.

6. Type ./computeROC.pl sorted-final-predictions.txt >roc.txt

This computes ROC points for the experiments.

We claim:

1. A method for detecting intrusions in the operation of a computer system comprising:

- (a) gathering features from records of normal processes that access the operating system registry;
- (b) generating a probabilistic model of normal computer system usage based on the features and determining the likelihood of observing an event that was not observed during the gathering of features from the records of normal processes; and
- (c) analyzing features from a record of a process that accesses the operating system registry to detect deviations from normal computer system usage to determine whether the access to the operating system registry is an anomaly.

2. The method according to claim 1, further comprising storing the probabilistic model of normal computer usage on the operating system registry.

3. The method according to claim 1, wherein gathering features from records of normal processes that access the operating system comprises gathering a feature corresponding to a name of a process accessing the operating system registry.

4. The method according to claim 1, wherein gathering features from records of normal processes that access the operating system registry comprises gathering a feature corresponding to a type of query being sent to the operating system registry.

5. The method according to claim 4, wherein gathering features from records of normal processes that access the operating system registry comprises gathering a feature corresponding to an outcome of a query being sent to the operating system registry.

6. The method according to claim 1, wherein gathering features from records of normal processes that access the operating system registry comprises gathering a feature corresponding to a name of a key being accessed in the operating system registry.

7. The method according to claim 6, wherein gathering features from records of normal processes that access the operating system registry comprises gathering a feature corresponding to a value of the key being accessed.

8. The method according to claim 1, wherein gathering features from records of normal processes that access the operating system registry comprises gathering two features selected from the group of features consisting of a name of a process accessing the operating system registry, a type of query being sent to the operating system registry, an outcome

US 7,448,084 B1

23

of a query being sent to the operating system registry, a name of a key being accessed in the operating system registry, and a value of the key being accessed.

9. The method according to claim 1, wherein generating a probabilistic model of normal computer system usage comprises determining a likelihood of observing a feature in the records of processes that access the operating system registry.

10. The method according to claim 9, wherein determining a likelihood of observing a feature comprises determining a conditional probability of observing a first feature in the records of processes that access the operating system registry given an occurrence of a second feature in the records.

11. The method according to claim 1, wherein analyzing a record of a process that accesses the operating system registry comprises, for each feature, performing a check to determine if a value of the feature has been previously observed for the feature.

12. The method according to claim 11, further comprising, if the value of the feature has not been observed, computing a score based on a probability of observing the value of the feature.

13. The method according to claim 12, further comprising, if the score is greater than a predetermined threshold, labeling the access to the operating system registry as anomalous and labeling the process that accessed the operating system registry as malicious.

14. A system for detecting intrusions in the operation of a computer system comprising:

- (a) an operating system registry;
- (b) a registry auditing module configured to gather records regarding processes that access the operating system registry;
- (c) a model generator configured to generate a probabilistic model of normal computer system usage based on records of a plurality of processes that access the operating system registry and that are indicative of normal computer system usage and to determine the likelihood of observing a process that was not observed in the records of the plurality of processes that access the operating system registry and that are indicative of normal computer usage; and
- (d) a model comparator configured to receive the probabilistic model of normal computer system usage and to receive records regarding processes that access the operating system registry and to detect deviations from normal computer system usage to determine whether the access of the operating system registry is an anomaly.

15. The system according to claim 14, wherein the probabilistic model of normal computer usage is stored in the operating system registry.

24

16. The system according to claim 14, further comprising a database configured to receive records regarding processes that access the operating system registry from the registry auditing module.

17. The system according to claim 14, wherein the model generator is configured to receive the records regarding processes that access the operating system registry from the database.

18. The system according to claim 14, wherein the records regarding processes that access the operating system registry comprise a feature of the access to the operating system registry.

19. The system according to claim 18, wherein the feature corresponds to a name of a process accessing the operating system registry.

20. The system according to claim 18, wherein the feature corresponds to a type of query being sent to the operating system registry.

21. The system according to claim 18, wherein the feature corresponds to an outcome of a query being sent to the operating system registry.

22. The system according to claim 18, wherein the feature corresponds to a name of a key being accessed in the operating system registry.

23. The system according to claim 18, wherein the feature corresponds to a value of the key being accessed.

24. The system according to claim 18, wherein the features corresponds to a combination of two features selected from the group of features consisting of a name of a process accessing the operating system registry, a type of query being sent to the operating system registry, an outcome of a query being sent to the operating system registry, a name of a key being accessed in the operating system registry, and a value of the key being accessed.

25. The system according to claim 14, wherein the model generator is configured to determine a likelihood of observing a feature in the records regarding processes that access the operating system registry.

26. The system according to claim 25, wherein the model generator is configured to determine a conditional probability of observing a first feature in records regarding processes that access the operating system registry given an occurrence of a second feature is the record.

27. The system according to claim 25, wherein the model comparator determines a score based on the likelihood of observing a feature in a record regarding a process that accesses the operating system registry.

28. The system according to claim 25, wherein the model comparator is configured to determine an access to the operating system registry is anomalous based on whether the score exceeds a predetermined threshold.

* * * * *

8

(12) **United States Patent**
Apap et al.

(10) **Patent No.:** **US 7,913,306 B2**
 (45) **Date of Patent:** ***Mar. 22, 2011**

(54) **SYSTEM AND METHODS FOR DETECTING INTRUSIONS IN A COMPUTER SYSTEM BY MONITORING OPERATING SYSTEM REGISTRY ACCESSES**

(75) Inventors: **Frank Apap**, Valley Stream, NY (US);
Andrew Honig, East Windsor, NJ (US);
Hershkop Shlomo, Brooklyn, NY (US);
Eleazar Eskin, Santa Monica, CA (US);
Salvatore J. Stolfo, Ridgewood, NJ (US)

(73) Assignee: **The Trustees of Columbia University in the City of New York**, New York, NY (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 214 days.

This patent is subject to a terminal disclaimer.

(21) Appl. No.: **12/154,405**

(22) Filed: **May 21, 2008**

(65) **Prior Publication Data**
 US 2009/0083855 A1 Mar. 26, 2009

Related U.S. Application Data

(63) Continuation of application No. 10/352,343, filed on Jan. 27, 2003, now Pat. No. 7,448,084.

(60) Provisional application No. 60/351,857, filed on Jan. 25, 2002.

(51) **Int. Cl.**
G06F 21/22 (2006.01)
G06F 11/30 (2006.01)

(52) **U.S. Cl.** **726/24; 726/22**

(58) **Field of Classification Search** None
 See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

- 6,647,400 B1 11/2003 Moran
- 6,742,124 B1 5/2004 Kilpatrick et al.
- 6,907,430 B2 6/2005 Chong et al.
- 7,162,741 B2 1/2007 Eskin et al.
- 7,225,343 B1 5/2007 Honig et al.
- 7,509,679 B2* 3/2009 Alagna et al. 726/24

(Continued)

OTHER PUBLICATIONS

U.S. Appl. No. 10/327,811, filed Dec. 19, 2002 claiming priority to U.S. Appl. No. 60/342,872, filed Dec. 20, 2001, entitled "System And Methods for Detecting a Denial-Of-Service Attack On A Computer System" of Salvatore J. Stolfo, Shlomo Hershkop, Rahul Bhan, Suhail Mohiuddin and Eleazar Eskin. (AP34898).

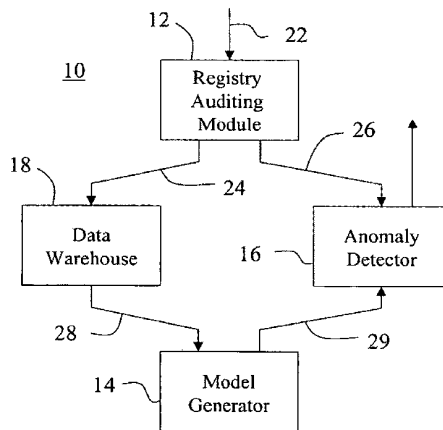
(Continued)

Primary Examiner — Christopher A Revak
 (74) *Attorney, Agent, or Firm* — Baker Botts LLP

(57) **ABSTRACT**

A method for detecting intrusions in the operation of a computer system is disclosed which comprises gathering features from records of normal processes that access the files system of the computer, such as the Windows registry, and generating a probabilistic model of normal computer system usage based on occurrences of said features. The features of a record of a process that accesses the Windows registry are analyzed to determine whether said access to the Windows registry is an anomaly. A system is disclosed, comprising a registry auditing module configured to gather records regarding processes that access the Windows registry; a model generator configured to generate a probabilistic model of normal computer system usage based on records of a plurality of processes that access the Windows registry and that are indicative of normal computer system usage; and a model comparator configured to determine whether the access of the Windows registry is an anomaly.

11 Claims, 3 Drawing Sheets



US 7,913,306 B2

Page 2

U.S. PATENT DOCUMENTS

2003/0065926	A1	4/2003	Schultz et al.	
2003/0167402	A1	9/2003	Stolfo et al.	
2004/0098607	A1*	5/2004	Alagna et al.	713/200
2004/0187023	A1*	9/2004	Alagna et al.	713/200
2006/0075492	A1*	4/2006	Golan et al.	726/22
2006/0174319	A1	8/2006	Kraemer et al.	
2009/0089040	A1*	4/2009	Monastyrsky et al.	703/26
2009/0288161	A1*	11/2009	Wei et al.	726/22
2009/0313699	A1*	12/2009	Jang et al.	726/23
2010/0095379	A1*	4/2010	Obrecht et al.	726/23

OTHER PUBLICATIONS

U.S. Appl. No. 10/320,259, filed Dec. 16, 2002 claiming priority to U.S. Appl. No. 60/328,682, filed Oct. 11, 2001 and U.S. Appl. No. 60/352,894, filed Jan. 29, 2002, entitled "Methods of Unsupervised Anomaly Detection Using a Geometric Framework" of Eleazar Eskin, Salvatore J. Stolfo and Leonid Portnoy. (AP34888).

U.S. Appl. No. 10/269,718, filed Oct. 11, 2002 claiming priority to U.S. Appl. No. 60/328,682, filed Oct. 11, 2001 and U.S. Appl. No. 60/340,198, filed Dec. 14, 2001, entitled "Methods for Cost-Sensitive Modeling for Intrusion Detection" of Salvatore J. Stolfo, Wenke Lee, Wei Fan and Matthew Miller. (AP34885).

U.S. Appl. No. 10/269,694, filed Oct. 11, 2002 claiming priority to U.S. Appl. No. 60/328,682, filed Oct. 11, 2001 and U.S. Appl. No. 60/339,952, filed Dec. 13, 2001, entitled "System and Methods for Anomaly Detection and Adaptive Learning" of Wei Fan, Salvatore J. Stolfo. (AP34886).

Lee et al., "A Framework for Constructing Features and Models for Intrusion Detection Systems," Nov. 2000, ACM Transactions on Information and System Security, vol. 3, No. 4, p. 22.

Eskin et al., "Adaptive Model Generation for Intrusion Detection Systems," Workshop on Intrusion Detection and Prevention, 7th ACM Conference on Computer Security, Athens, Nov. 2000.

Korba, "Windows NT Attacks for the Evaluation of Intrusion Detection Systems," May 2000.

Honig A et al., (2002) "Adaptive model generation: An Architecture for the deployment of data mining-based intrusion detection systems." In *Data Mining for Security Applications*. Kluwer.

Friedman N et al., (1999) "Efficient bayesian parameter estimation in large discrete domains." *Advances in Neural Information Processing Systems 11*.

Javits HS et al., Mar. 7, 1994, "The nides statistical component: Description and justification." *Technical report, SRI International*.

D. E. Denning, An Intrusion Detection Model, *IEEE Transactions on Software Engineering*, SE-13:222-232, 1987.

Wenke Lee, Sal Stolfo, and Kui Mok. "Mining in a Data-flow Environment: Experience in Network Intrusion Detection" In *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '99)*, San Diego, CA, Aug. 1999.

Stephanie Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A Sense of Self for UNIX Processes," *IEEE Computer Society*, pp. 120-128, 1996.

Christina Warrender, Stephanie Forrest, and Barak Pearlmutter, "Detecting Intrusions Using System Calls: Alternative Data Models," *IEEE Computer Society*, pp. 133-145, 1999.

S. A. Hofmeyr, Stephanie Forrest, and A. Somayaji, "Intrusion Detect Using Sequences of System Calls," *Journal of Computer Security*, 6:151-180, 1998.

W. Lee, S. J. Stolfo, and P. K. Chan, "Learning Patterns from UNIX Processes Execution Traces for Intrusion Detection," *AAAI Press*, pp. 50-56, 1997.

Eleazar Eskin, "Anomaly Detection Over Noisy Data Using Learned Probability Distributions," *Proceedings of the Seventeenth International Conference on Machine Learning (ICML-2000)*, 2000.

N. Friedman and Y. Singer, "Efficient Bayesian Parameter Estimation in Large Discrete Domains," *Advances in Neural Information Processing Systems 11*, MIT Press, 1999.

M. Mahoney and P. Chan, "Detecting Novel Attacks by Identifying Anomalous Network Packet Headers," *Technical Report CS-2001-2*, Florida Institute of Technology, Melbourne, FL, 2001.

H. Debar et al., "Intrusion Detection Exchange Format Data Model," Internet Engineering Task Force, Jun. 15, 2000.

* cited by examiner

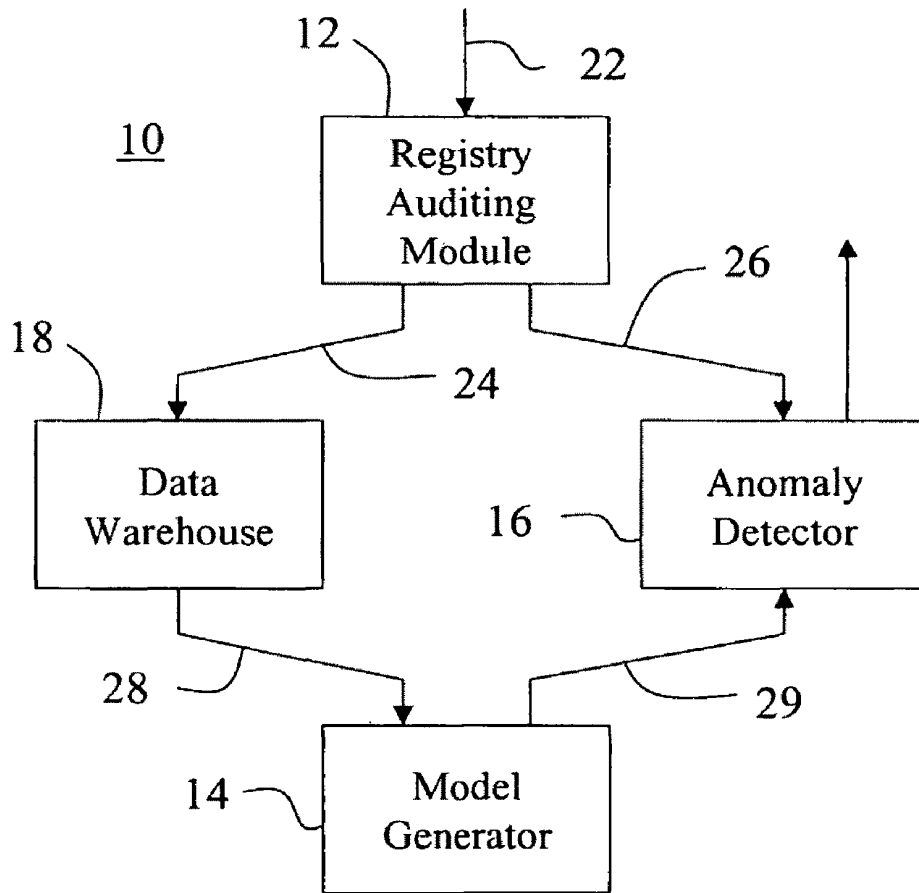


FIG. 1

30

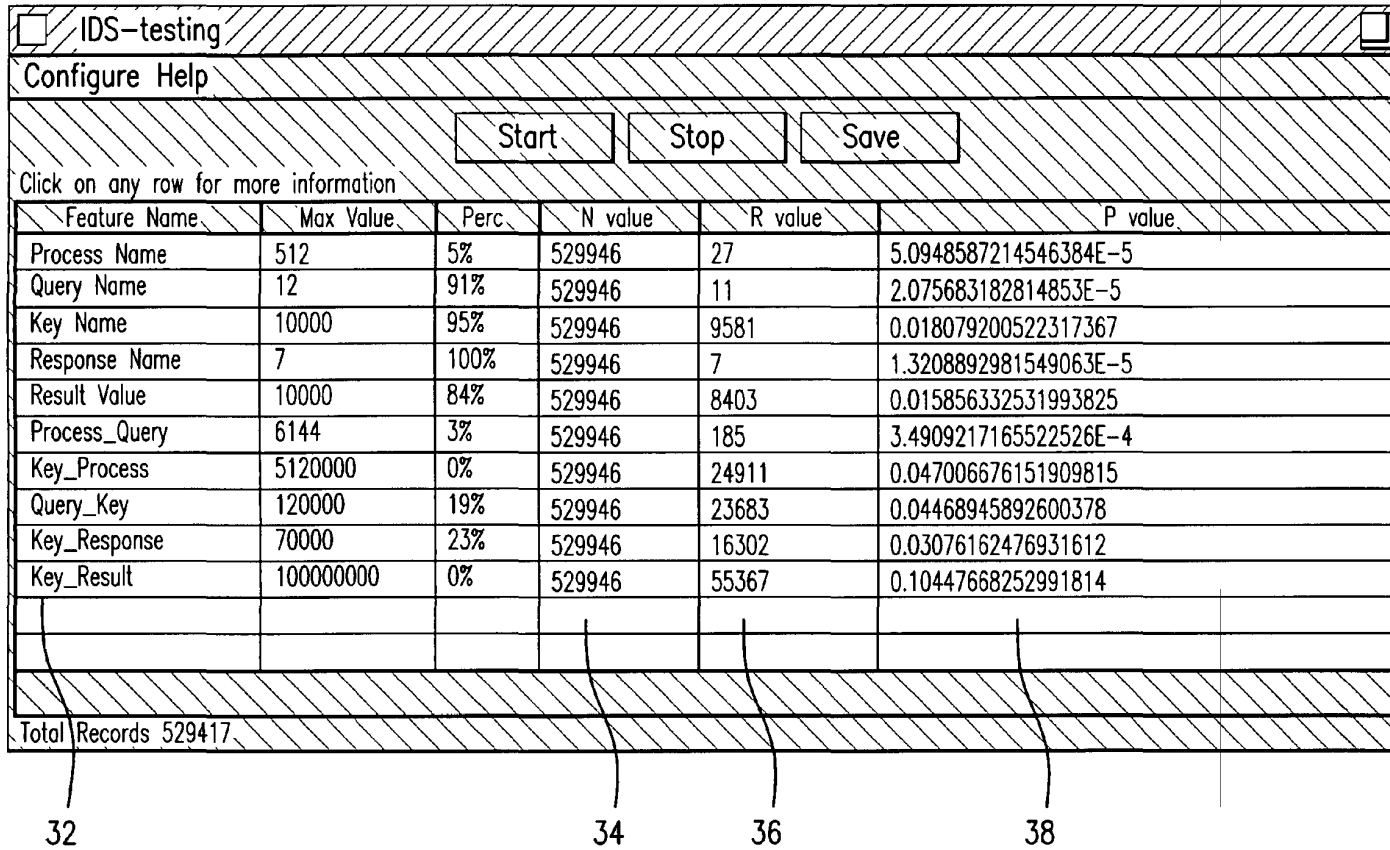


FIG. 2

COL00000041

U.S. Patent

Mar. 22, 2011

Sheet 2 of 3

US 7,913,306 B2

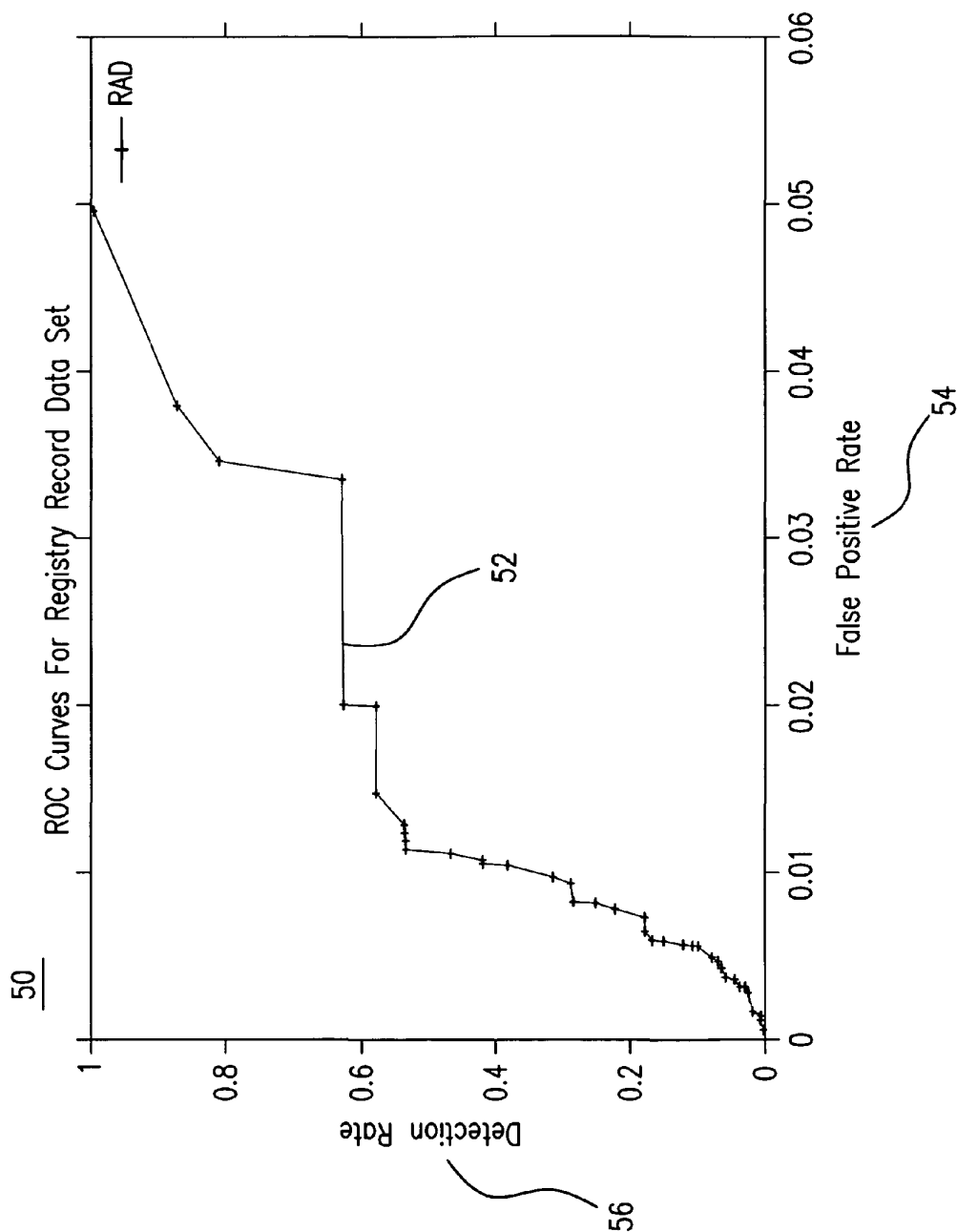


FIG. 3

US 7,913,306 B2

1

SYSTEM AND METHODS FOR DETECTING INTRUSIONS IN A COMPUTER SYSTEM BY MONITORING OPERATING SYSTEM REGISTRY ACCESSES

CROSS-REFERENCE TO RELATED APPLICATIONS

This application is a continuation of U.S. patent application Ser. No. 10/352,343, filed on Jan. 27, 2003, which is now U.S. Pat. No. 7,448,084, and which claims the benefit of U.S. Provisional Patent Application Ser. No. 60/351,857, filed on Jan. 25, 2002, each of which is incorporated by reference in its entirety herein.

STATEMENT REGARDING FEDERALLY SPONSORED RESEARCH OR DEVELOPMENT

This invention was made with government support under grant nos. FAS-526617, SRTSC-CU019-7950-1, and F30602-00-1-0603 awarded by the United States Defense Advanced Research Projects Agency (DARPA). The government has certain rights in the invention.

COMPUTER PROGRAM LISTING

A computer program listing is submitted in duplicate on CD. Each CD contains a routines listed in the Appendix, which CD was created on May 20, 2008, and which is 22 MB in size. The files on this CD are incorporated by reference in their entirety herein.

COPYRIGHT NOTICE

A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by any one of the patent disclosure, as it appears in the Patent and Trademark Office patent files or records, but otherwise reserves all copyright rights whatsoever.

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates to systems and methods for detecting anomalies in a computer system, and more particularly to the use of probabilistic and statistical models to model the behavior of processes which access the file system of the computer, such as the Windows™ registry.

2. Background

Windows™ is currently one of the most widely used operating systems, and consequently computer systems running the Windows™ operating system are frequently subject to attacks. Malicious software is often used to perpetrate these attacks. Two conventional approaches to respond to malicious software include virus scanners, which attempt to detect the malicious software, and security patches that are created to repair the security “hole” in the operating system that the malicious software has been found to exploit. Both of these methods for protecting hosts against malicious software suffer from drawbacks. While they may be effective against known attacks, they are unable to detect and prevent new and previously unseen types of malicious software.

Many virus scanners are signature-based, which generally means that they use byte sequences or embedded strings in software to identify certain programs as malicious. If a virus scanner’s signature database does not contain a signature for

2

a malicious program, the virus scanner is unable to detect or protect against that malicious program. In general, virus scanners require frequent updating of signature databases, otherwise the scanners become useless to detect new attacks. Similarly, security patches protect systems only when they have been written, distributed and applied to host systems in response to known attacks. Until then, systems remain vulnerable and attacks are potentially able to spread widely.

Frequent updates of virus scanner signature databases and security patches are necessary to protect computer systems using these approaches to defend against attacks. If these updates do not occur on a timely basis, these systems remain vulnerable to very damaging attacks caused by malicious software. Even in environments where updates are frequent and timely, the systems are inherently vulnerable from the time new malicious software is created until the software is discovered, new signatures and patches are created, and ultimately distributed to the vulnerable systems. Since malicious software may be propagated through email, the malicious software may reach the vulnerable systems long before the updates are in place.

Another approach is the use of intrusion detection systems (IDS). Host-based IDS systems monitor a host system and attempt to detect an intrusion. In an ideal case, an IDS can detect the effects or behavior of malicious software rather than distinct signatures of that software. In practice, many of the commercial IDS systems that are in widespread use are signature-based algorithms, having the drawbacks discussed above. Typically, these algorithms match host activity to a database of signatures which correspond to known attacks. This approach, like virus detection algorithms, requires previous knowledge of an attack and is rarely effective on new attacks. However, recently there has been growing interest in the use of data mining techniques, such as anomaly detection, in IDS systems. Anomaly detection algorithms may build models of normal behavior in order to detect behavior that deviates from normal behavior and which may correspond to an attack. One important advantage of anomaly detection is that it may detect new attacks, and consequently may be an effective defense against new malicious software. Anomaly detection algorithms have been applied to network intrusion detection (see, e.g., D. E. Denning, “An Intrusion Detection Model, *IEEE Transactions on Software Engineering*, SE-13: 222-232, 1987; H. S. Javitz and A. Valdes, “The NIDES Statistical Component: Description and Justification, *Technical report, SRI International*, 1993; and W. Lee, S. J. Stolfo, and K. Mok, “Data Mining in Work Flow Environments: Experiences in Intrusion Detection,” *Proceedings of the 1999 Conference on Knowledge Discovery and Data Mining (KDD-99)*, 1999) and also to the analysis of system calls for host based intrusion detection (see, e.g., Stephanie Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, “A Sense of Self for UNIX Processes,” *IEEE Computer Society*, pp. 120-128, 1996; Christina Warrender, Stephanie Forrest, and Barak Pearlmutter, “Detecting Intrusions Using System Calls: Alternative Data Models,” *IEEE Computer Society*, pp. 133-145, 1999; S. A. Hofmeyr, Stephanie Forrest, and A. Somayaji, “Intrusion Detect Using Sequences of System Calls,” *Journal of Computer Security*, 6:151-180, 1998; W. Lee, S. J. Stolfo, and P. K. Chan, “Learning Patterns from UNIX Processes Execution Traces for Intrusion Detection,” AAAI Press, pp. 50-56, 1997; and Eleazar Eskin, “Anomaly Detection Over Noisy Data Using Learned Probability Distributions,” *Proceedings of the Seventeenth International Conference on Machine Learning (ICML-2000)*, 2000).

There are drawbacks to the prior art approaches. For example, the system call approach to host-based intrusion

US 7,913,306 B2

3

detection has several disadvantages which inhibit its use in actual deployments. A first is that the computational overhead of monitoring all system calls is potentially very high, which may degrade the performance of a system. A second is that system calls themselves are typically irregular by nature. Consequently, it is difficult to differentiate between normal and malicious behavior, and such difficulty to differentiate behavior may result in a high false positive rate.

Accordingly, there is a need in the art for an intrusion detection system which overcomes these limitations of the prior art.

SUMMARY OF THE INVENTION

It is an object of the invention to provide techniques which are effective in detecting attacks while maintaining a low rate of false alarms.

It is another object of the invention to generate a model of the normal access to the Windows registry, and to detect anomalous accesses to the registry that are indicative of attacks.

These and other aspects of the invention are realized by a method for detecting intrusions in the operation of a computer system comprising the steps of gathering features from records of normal processes that access the Windows registry. Another step comprises generating a probabilistic model of normal computer system usage, e.g., free of attacks, based on occurrences of said features. The features of a record of a process that accesses the Windows registry are analyzed to determine whether said access to the Windows registry is an anomaly.

According to an exemplary embodiment, the step of gathering features from the records of normal processes that access the Windows registry may comprise gathering a feature corresponding to a name of a process that accesses the registry. Another feature may correspond to the type of query being sent to the registry. A further feature may correspond to an outcome of a query being sent to the registry. A still further feature may correspond to a name of a key being accessed in the registry. Yet another feature may correspond to a value of said key being accessed. In addition, any other features may be combined to detect anomalous behavior that may not be identified by analyzing a single feature.

The step of generating a probabilistic model of normal computer system usage may comprise determining a likelihood of observing a feature in the records of processes that access the Windows registry. This may comprise performing consistency checks. For example, a first order consistency check may comprise determining the probability of observation of a given feature. The step of determining a likelihood of observing a feature may comprise a second order consistency check, e.g., determining a conditional probability of observing a first feature in said records of processes that access the Windows registry given an occurrence of a second feature in said records.

In the preferred embodiment, the step of analyzing a record of a process that accesses the Windows registry may comprise, for each feature, determining if a value of the feature has been previously observed for the feature. If the value of the feature has not been observed, then a score may be determined based on a probability of observing said value of the feature. If the score is greater than a predetermined threshold, the method comprises labeling the access to the Windows registry as anomalous and labeling the process that accessed the Windows registry as malicious.

A system for detecting intrusions in the operation of a computer system has an architecture which may comprise a

4

registry auditing module, e.g., a sensor, configured to gather records regarding processes that access the Windows registry. A model generator is also provided which is configured to generate a probabilistic model of normal computer system usage based on records of a plurality of processes that access the Windows registry and that are indicative of normal computer system usage, e.g., free of attacks. A model comparator, e.g., anomaly detector, is configured to receive said probabilistic model of normal computer system usage and to receive records regarding processes that access the Windows registry and to determine whether the access of the Windows registry is an anomaly.

The system may also comprise a database configured to receive records regarding processes that access the Windows registry from said registry auditing module. The model generator may be configured to receive the records regarding processes that access the Windows registry from the database.

The model generator may be configured to determine a conditional probability of observing a first feature in records regarding processes that access the Windows registry given an occurrence of a second feature in said record. The model comparator may be configured to determine a score based on the likelihood of observing a feature in a record regarding a process that accesses the Windows registry. The model comparator may also be configured to determine whether an access to the Windows registry is anomalous based on whether the score exceeds a predetermined threshold.

In accordance with the invention, the objects as described above have been met, and the need in the art for detecting intrusions based on registry accesses has been satisfied.

BRIEF DESCRIPTION OF THE DRAWINGS

Further objects, features, and advantages of the invention will become apparent from the following detailed description taken in conjunction with the accompanying figures showing illustrative embodiments of the invention, in which:

FIG. 1 is a block diagram illustrating the architecture of the system in accordance with the present invention.

FIG. 2 is an exemplary user interface in accordance with the present invention.

FIG. 3 is a plot illustrating the result of an embodiment of the present invention.

Throughout the figures, the same reference numerals and characters, unless otherwise stated, are used to denote like features, elements, components, or portions of the illustrated embodiments. Moreover, while the subject invention will now be described in detail with reference to the figures, it is done so in connection with the illustrative embodiments without departing from the true scope and spirit of the invention as defined by the appended claims.

DETAILED DESCRIPTION OF THE EXEMPLARY EMBODIMENTS

A novel intrusion detection system **10** is disclosed herein and illustrated in FIG. 1. System **10** monitors a program's access of the file system of the computer, e.g., Microsoft™ Windows™ registry (hereinafter referred to as the "Windows™ registry" or the "registry") and determines whether the program is malicious. The system and methods described herein incorporate a novel technique referred to herein as "RAD" (Registry Anomaly Detection), which monitors the accesses to the registry, preferably in real time, and detects the actions of malicious software.

As is known in the art, the registry is an important component of the Windows™ operating system and is implemented

US 7,913,306 B2

5

very widely. Accordingly, a substantial amount of data regarding activity associated with the registry is available. The novel technique includes building a sensor, e.g., registry auditing module, on the registry and applying the information gathered by this sensor to an anomaly detector. Consequently, registry activity that corresponds to malicious software may be detected. Several advantages of monitoring the registry include the fact that registry activity is regular by nature, that the registry can be monitored with low computational overhead, and that almost all system activities query the registry.

An exemplary embodiment of the novel system, e.g., system 10, comprises several components: a registry auditing module 12, a model generator 14, and an anomaly detector 16. Generally, the sensor 12 serves to output data for each registry activity to a database 18 where it is stored for training. The model generator 14 reads data from the database 18, and creates a model of normal behavior. The model is then used by the anomaly detector 16 to decide whether each new registry access should be considered anomalous. The above components can reside on the host system itself, or on a remote network connected to the host.

A description of the Windows™ registry is provided herein. As is known in the art, the registry is a database of information about a computer's configuration. The registry contains information that is continually referenced by many different programs during the operation of the computer system. The registry may store information concerning the hardware installed on the system, the ports that are being used, profiles for each user, configuration settings for programs, and many other parameters of the system. The registry is the main storage location for all configuration information for almost all programs. The registry is also the storage location for all security information such as security policies, user names, and passwords. The registry also stores much of the important configuration information that are needed by programs in order to run.

The registry is organized hierarchically as a tree. Each entry in the registry is called a "key" and has an associated value. One example of a registry key is:

```
HKCU\Software\America Online\AOL Instant Messenger (TM)\
CurrentVersion\Users\aimuser\Login\Password
```

This example represents a key used by the AOL™ Instant Messenger™ program. This key stores an encrypted version of the password for the user name "aimuser." Upon execution, the AOL™ Instant Messenger™ program access the registry. In particular, the program queries this key in the registry in order to retrieve the stored password for the local user. Information is accessed from the registry by individual registry accesses or queries. The information associated with a registry query may include the key, the type of query, the result, the process that generated the query, and whether the query was successful. One example of a query is a read for the key, shown above. For example, the record of the query is:

```
Process: aim.exe
Query: Queryvalue
Key: HKCU\Software\America Online\AOL Instant Mes-
senger(TM)\CurrentVersion\Users\aimuser\Login\Pass-
word
Response: SUCCESS
ResultValue: "BCOFHIHBBAHF"
```

The registry serves as an effective data source to monitor for attacks because it has been found that many attacks are manifested as anomalous registry behavior. For example, cer-

6

tain attacks have been found to take advantage of Windows™ reliance on the registry. Indeed, many attacks themselves rely on the registry in order to function properly. Many programs store important information in the registry, despite the fact that any other program can likewise access data anywhere inside the registry. Consequently, some attacks target the registry to take advantage of this access. In order to address this concern, security permissions are included in some versions of Windows™. However, such permissions are not included in all versions of Windows™, and even when they are included, many common applications do not make use of this security feature.

"Normal" computer usage with respect to the registry is described herein. Most typical Windows™ programs access a certain set of keys during execution. Furthermore, each user typically uses a certain set of programs routinely while running their machine. This set of programs may be a set of all programs installed on the machine, or a small subset of these programs.

Another important characteristic of normal registry activity is that it has been found to be substantially regular over time. Most programs may either (1) access the registry only on startup and shutdown, or (2) access the registry at specific intervals. Since this access to the registry appears to be substantially regular, monitoring the registry for anomalous activity provides useful results because a program which substantially deviates from this normal activity may be easily detected as anomalous.

Other normal registry activity occurs only when the operating system is installed by the manufacturer. Some attacks involve launching programs that have not been launched before and/or changing keys that have not been changed since the operating system was first installed by the manufacturer.

If a model of the normal registry behavior is trained over clean data, then these kinds of registry operations will not appear in the model, and can be detected when they occur. Furthermore, malicious programs may need to query parts of the registry to get information about vulnerabilities. A malicious program can also introduce new keys that will help create vulnerabilities in the machine.

Some examples of malicious programs and how they produce anomalous registry activity are as follows:

Setup Trojan: This program, when launched, adds full read/write sharing access on the file system of the host machine. It makes use of the registry by creating a registry structure in the networking section of the Windows™ keys. The structure stems from HKI.M\Software\Microsoft\Windows\CurrentVersion\Network\LanMan. It then makes several, e.g., eight, new keys for its use. It also accesses HKLM\Security\Provider in order to find information about the security of the machine to help determine vulnerabilities. This key is not accessed by any normal programs during training or testing in our experiments, and therefore its use is clearly suspicious in nature.

Back Orifice 2000: This program opens a vulnerability on a host machine, which may grant anyone with the back orifice client program complete control over the host machine. This program makes extensive use of the registry. In doing so, it does access a key that is very rarely accessed on the Windows™ system. This key, HKLM\Software\Microsoft\VBA\Monitors, was not accessed by any normal programs in either the training or test data. Accordingly, the detection algorithm was able to determine it as anomalous. This program also launches many other programs (e.g., LoadWC.exe, Patch.exe, runonce.exe, bo2k_i_o_intl.e) as part of the attack, in which all of these programs made anomalous accesses to the registry.

US 7,913,306 B2

7

Aimrecover: This program obtains passwords from AOL™ users without authorization. It is a simple program that reads the keys from the registry where the AOL™ Instant Messenger™ program stores the user names and passwords. These accesses are considered anomalous because Aimrecover is accessing a key that usually is accessed and was created by a different program.

Disable Norton: This is a simple exploitation of the registry that disables Norton™ Antivirus. This attack toggles one record in the registry, in particular, the key HKLM\SOFTWARE\INTELLANDesk\Virusprotect6\CurrentVersion\Storages\Files\System\RealTimeScan\OnOff. If this value is set to 0, then Norton™ Antivirus real-time system monitoring is turned off. Again, this is considered anomalous because of its access to a key that was created by a different program.

L0phtCrack: This program is a widely used password cracking program for Windows™ machines. It obtains the hashed SAM file containing the passwords for all users, and then uses either a dictionary or a brute force approach to find the passwords. This program also uses flaws in the Windows™ encryption scheme in order to try to find some of the characters in a password in order to obtain a password faster. This program uses the registry by creating its own section in the registry. This new section may include many create key and set value queries, all of which will be on keys that did not exist previously on the host machine and therefore have not been seen before.

An additional aspect of normal computer usage of the registry is described herein. During testing (as will be described below) all of the programs observed in the data set cause Explorer™ to access a key specifically for that application. This key has the following format:

```
HKLM\Software\Microsoft\WindowsNT\CurrentVersion\Image File
Execution Options\[processName]
```

where “processName” is the name of the process being run. (It is believed that all programs in general have this behavior.) This key is accessed by Explorer™ each time an application is run. Given this information, a detection system may be able to determine when new applications are run, which will be a starting point to determine malicious activity. In addition, many programs add themselves in the auto-run section of the registry under the following key:

```
HKLM\Software\Microsoft\Windows\CurrentVersion\Run.
```

While this activity is not malicious in nature, it is nevertheless an uncommon event that may suggest that a system is being attacked. Trojans such as Back Orifice utilize this part of the registry to auto load themselves on each boot.

Anomaly detectors, such as anomaly detector 16, do not operate by looking for malicious activity directly. Rather, they look for deviations from normal activity. Consequently, such deviations, which represent normal operation, may nevertheless be declared an attack by the system. For example, the installation of a new program on a system may be viewed as anomalous activity by the anomaly detector, in which new sections of the registry and many new keys may be created. This activity may interpreted as malicious (since this activity was not in the training data) and a false alarm may be triggered, much like the process of adding a new machine to a network may cause an alarm on an anomaly detector that analyzes network traffic.

There are a few possible solutions to avoid this problem. Malicious programs often install quietly so that the user does

8

not know the program is being installed. This is not the case with most installations. For example, in an exemplary embodiment, the algorithm is programmed to ignore alarms while the install shield program is running, because the user would be aware that a new program is being installed (as opposed to malicious activity occurring without the users knowledge and permission). In another embodiment, the user is prompted when a detection occurs, which provides the user with the option of informing the algorithm that the detected program is not malicious and therefore grants permission for such program to be added to the training set of data to update the anomaly detector for permissible software.

In order to detect anomalous registry accesses, model generator 14 of system 10 generates a model of normal registry activity. A set of five basic features are extracted from each registry access. (An additional five features may be added, which are combinations of the five basic features, as described below.) Statistics of the values of these features over normal data are used to create the probabilistic model of normal registry behavior. This model of normal registry behavior may include a set of consistency checks applied to the features, as will be described below. When detecting anomalies, the model of normal behavior is used to determine whether the values of the features of the new registry accesses are consistent with the normal data. If such values are not consistent, the algorithm labels the registry access as anomalous, and the processes that accessed the registry as malicious.

In the exemplary embodiment, the data model consists of five basic features gathered by the registry auditing module 12 from an audit stream. (It is contemplated that additional features may also provide significant results.) In the exemplary embodiment, the features are as follows:

Process: The name of the process accessing the registry. This allows the tracking of new processes that did not appear in the training data.

Query: The type of query being sent to the registry, for example, QueryValue, CreateKey, and SetValue are valid query types. This allows the identification of query types that have not been seen before. There are many query types but only a few are used under normal circumstances.

Key: The actual key being accessed. Including this feature allows the algorithm to locate keys that are never accessed in the training data. Many keys are used only once for special situations like system installation. Some of these keys can be used by attacks to create vulnerabilities.

Response: A feature which describes the outcome of the query, for example, success, not found, no more, buffer overflow, and access denied.

Result Value: The value of the key being accessed. Including this feature allows the algorithm to detect abnormal values being used to create abnormal behavior in the system.

Composite features may also be used, which are a combination of two basic features, such as those discussed above. These composite features are useful in detecting anomalies since they allow the system to give more detail to its normal usage model where the existence of the basic features in isolation would not necessarily be detected as anomalous. The following is a list of exemplary composite fields that may be used by system 10:

Process/Query: This key is a combination of the process and query fields, and may provide information to determine if a particular process is executing queries that are not typically executed by this process.

Key/Process: This key is a combination of the key and process fields, and may allow the algorithm to detect whether or not a process accesses portions of the registry it typically doesn't access. Also it detects when a key is being accessed by

a process that normally doesn't access it. This analysis is very useful because many processes access keys belonging to other programs, and this field would allow the detection of such an event.

Query/Key: This key is a combination of the query and key fields, and may determine if a key is being accessed in a different way than usual. For example, there are many key values in the registry that are written once at the time of their creation, and then are subsequently read from the registry without change. Some of these keys store crucial information for the execution of certain programs. If a malicious process were to write to one of these keys, after the time of their creation, this field would enable the algorithm to detect such an event, and the record would appear anomalous.

Response/Key: This key is a combination of the key and response fields. Many keys are found to be used in the same manner each time they are used. Consequently, they will always return the same response. Since a different response to a key may be indicative of abnormal or malicious system activity, the algorithm which utilizes this combination key would be able to detect such abnormal or malicious activity.

Result Value/Key: This key is a combination of the key and result value. During operation of the computer system, many keys will always contain a value from a certain set or range of values. When the value is outside that range, that could be an indicator of a malicious program taking advantage of a vulnerability.

As an illustration, an exemplary registry access is displayed in Table 1. The second column is normal access by the process aim.exe (which is used with AOL™ Instant Messenger™) to access the key where passwords are stored. The third column of Table 1 is a malicious access by a process aimrecover.exe to the same key. The final column of Table 1 shows which fields register by the anomaly detector as anomalous. As seen in the table, all of the basic features, e.g., process, query, key, response, and result value, do not appear anomalous for the normal process aim.exe, when compared with the malicious process aimrecover.exe. However, the composite keys are useful for detecting the anomalous behavior of aimrecover.exe. For example, the fact that the process aimrecover.exe is accessing a key that is usually associated with another process, i.e., aim.exe, is detected as an anomaly. This conclusion is made because under normal circumstances only aim.exe accesses the key that stores the AOL™ Instant Messenger™ password. The occurrence of another process accessing this key is considered suspicious. By examining the combination of two basic features, the algorithm can detect this anomaly.

Exemplary embodiments of intrusion detection algorithms which may be used by the system 10 will now be described, although it is understood that other anomaly detection algorithms may also be used in connection with the present invention. Since a significant amount of data is monitored in real time, the algorithm that is selected must be very efficient. According to a first exemplary embodiment, the features that were monitored from each registry access are used to train a model over features extracted from normal data. That model allows for the classification of registry accesses as either normal or malicious, as will be described herein.

In general, a principled probabilistic approach to anomaly detection can be reduced to density estimation. If a density function $p(x)$ can be estimated over the normal data, anomalies are defined as data elements that occur with low probability. In practice, estimating densities is a very complex, non-trivial problem. In detecting intrusions into the registry, a complication is that each of the features have many possible values. For example, the key feature, defined above, may have over 30,000 values in the training set. Since there are so many possible feature values, it is relatively rare that the same exact record occurs more than once in the data. Data sets of this type are referred to as "sparse."

Since probability density estimation is a very complex problem over sparse data, the method of the present invention defines a set of consistency checks over the normal data for determining which records from a sparse data set are anomalous. Each consistency check is applied to an observed record by the anomaly detector. If the record fails any consistency check, the record is labeled as anomalous.

In the exemplary embodiment, two kinds of consistency checks are applied. The first consistency check evaluates whether or not a feature value is consistent with observed values of that feature in the normal data set. This type of consistency check is referred to as a first order consistency check, e.g., each registry record may be viewed as the outcome of five random variables, one for each feature, X_1, X_2, X_3, X_4, X_5 . The consistency checks compute the likelihood of an observation of a given feature denoted as $P(X_i)$.

The second consistency check handles pairs of features, as discussed in the example in Table 1. For each pair of features, the conditional probability of a feature value given another feature value is considered. These consistency checks are referred to as second order consistency checks. These likelihoods are denoted as $P(X_i|X_j)$. For each value of X_j , there may be a different probability distribution over X_i .

TABLE 1

Feature	aim.exe	aimrecover.exe	Anomalous
Process	aim.exe	aimrecover.exe	no
Query	QueryValue	QueryValue	no
Key	HKCU\Software\America Online\AOL Instant Messenger (TM)\CurrentVersion\Users\aimuser\Login\Password	HKCU\Software\America Online \AOL Instant Messenger (TM) \CurrentVersion\Users\aimuser\Login\Password	no
Response	SUCCESS	SUCCESS	no
Result Value	"BCOFHIIHBAHF"	"BCOFHIIHBAHF"	no
Process/Query	aim.exe:QueryValue	aimrecover:QueryValue	no
Query/Key	QueryValue:HKCU\Software\America Online\AOL Instant Messenger (TM)\CurrentVersion\Users\aimuser\Login\Password	QueryValue:HKCU\Software\America Online \AOL Instant Messenger (TM)\CurrentVersion\Users\aimuser\Login\Password	no
Response/Key	SUCCESS:HKCU\Software\America Online\AOL Instant Messenger (TM)\CurrentVersion\Users\aimuser\Login\Password	SUCCESS:HKCU\Software\America Online\AOL Instant Messenger (TM)\CurrentVersion\Users\aimuser\Login\Password	no
Process/Key	aim.exe:HKCU\Software\America Online\AOL Instant Messenger (TM)\CurrentVersion\Users\aimuser\Login\Password	aimrecover.exe:HKCU\Software\America Online\AOL Instant Messenger (TM)\CurrentVersion\Users\aimuser\Login\Password	yes

US 7,913,306 B2

11

In the exemplary embodiment, since there are five basic feature values, for each record, there are five first order consistency checks and 20 second order consistency checks of which five examples are given above. If the likelihood of any of the consistency checks is below a threshold, the record is labeled as anomalous. The determination of the threshold is described in greater detail below.

The manner in which the likelihoods for the first order ($P(X_i)$) consistency checks and the second order ($P(X_i|X_j)$) consistency checks are computed is described herein. From the normal data, there is a set of observed counts from a discrete alphabet, e.g., a finite number of distinct symbols or feature values, for each of the consistency checks. Computing the above likelihoods reduces to estimating a multinomial expression. In theory, the maximum likelihood estimate may be used, which computes the ratio of the counts of a particular element to the total counts. However, the maximum likelihood estimate has been found to be biased when relatively small amounts of data, e.g., "sparse data," are available. The distribution may be smoothed by adding a virtual count to each possible element. For anomaly detection, it is often desirable to take into account how likely it is to observe a previously unobserved element. Thus, if many different elements have been seen in the training data, it is therefore more likely to see additional, unobserved elements, as opposed to the case where very few elements have been seen, in which additional, unobserved elements would be unlikely. (The term "element" here refers to feature values, or a vector of feature values.)

To estimate the likelihoods, an estimator is used, which gives the following prediction for element i :

$$P(X = i) = \frac{\alpha + N_i}{k^0 \alpha + N} \quad (1)$$

if element i was observed in the training data. If element i was not previously observed, then the following prediction is used:

$$P(X = i) = \frac{1}{L - k^0} (1 - C) \quad (2)$$

In these equations, the term α is a prior count for each element. The term N_i is the number of times element i was observed; N is the total number of observations, k^0 is the number of different elements observed, and L is the total number of possible elements or the alphabet size. The parameters are either observed or computed (e.g., N and k^0 are determined by storing values and computing frequency counts) while L is defined by the particular system being modeled, i.e., the type of variables (e.g., 32 bit integers) defines a range of possible values. Thus, L is predefined by the implementation details of the system. Here, the system is modeling the Windows™ registry, which has a predefined range of possible values (as defined by the programmers of the registry.) The scaling factor C takes into account how likely it is to observe a previously observed element versus an unobserved element. C is computed by the following equation:

$$C = \left(\sum_{k=k^0}^L \frac{k^0 \alpha + N}{k \alpha + N} m_k \right) \left(\sum_{k=k^0} m_k \right)^{-1} \quad (3)$$

12

where

$$m_k = P(S = k) \frac{k!}{(k - k^0)!} \frac{\Gamma(k \alpha)}{\Gamma(k \alpha + N)}$$

and $P(S=k)$ is a prior probability associated with the size of the subset of elements in the alphabet that have non-zero probability. Although the computation of C is expensive, it only needs to be done once for each consistency check at the end of training. Second order consistency checks are done in like fashion, except the particular values being measured are not distinct features values, but pairs of feature values, considering these pairs as a distinct element.

The prediction of the probability estimator is derived using a mixture of Dirichlet estimators, as are known in the art, see, e.g., the estimator presented in N. Friedman and Y. Singer, "Efficient Bayesian Parameter Estimation in Large Discrete Domains," *Advances in Neural Information Processing Systems* 11, MIT Press, which is incorporated by reference in its entirety herein.) The scores computed in the attached software code correspond to the estimates provided by the consistency checks in equations (1) and (2) above.

This exemplary embodiment of the algorithm labels every registry access as either normal or anomalous. Programs can have anywhere from just a few registry accesses to several thousand. This means that many attacks will be represented by large numbers of records where many of those records will be considered anomalous.

A second exemplary embodiment of the algorithm is described herein. Using the features that are monitored from each registry access, a score is computed to classify each access as either normal or malicious. A set of normal registry accesses are analyzed as a model of normal usage of the registry. Then using this model, new registry records are analyzed to determine whether or not they are malicious.

As the data is being collected, several important statistics are collected about each feature and the values that occur for each feature. For each feature, which values occurred for that feature and how many distinct values occurred for the feature, r , are recorded. Accordingly, r is a measure of how likely it is to see a new value for the feature. If many distinct values for a feature have been previously observed, i.e., a high value for r , and subsequently a never-observed value is encountered, such new value would be expected and considered normal. In contrast, if only a few distinct values have been observed, i.e., a low value for r , the observation of a new value is unlikely and possibly anomalous. The total number of registry different elements, e.g., training records, that are observed during training, n , is also recorded.

During training, for each of the features, all of the distinct observed values of the feature are stored, as well as the number of distinct observed values r . The total number of training records n is computed. For each feature, the algorithm computes $p=r/n$, which is an approximation of the probability of observing an unseen value for that feature in the normal data. To minimize storage requirements, instead of storing all distinct values, the algorithm hashes all of the values and stores the hashes of the values in a bit vector. More details on this technique is described in the implementation of PHAD (Packet Header Anomaly Detection), an anomaly detection algorithm known in the art, which was developed to detect anomalies in packet headers (see, e.g., M. Mahoney and P. Chan, "Detecting Novel Attacks by Identifying Anomalous Network Packet Headers," *Technical Report CS-2001-2*, Florida Institute of Technology, Melbourne, Fla., 2001).

US 7,913,306 B2

13

Once the model has been trained, new registry accesses can be evaluated and a score computed to determine whether or not the registry accesses are abnormal. For a new registry access, we first extract the features for the registry access. For each of these features, a check is performed to see if the value of the feature has been observed for the feature. If the value has not been observed, a heuristic score is computed which determines the level of anomaly for that feature. The score is determined as $1/p$ for each feature. Intuitively this score will be higher for features where fewer distinct values have been observed. The final score for a registry access is the sum of the scores for each feature that observed a previously unobserved value. If this value is greater than a threshold, we label the registry access anomalous and declare the process that generated it as malicious. The results from this experiment are described below.

The basic architecture of the system **10** will now be discussed in greater detail herein. With continued reference to FIG. **1**, the registry auditing module **12** monitors accesses to the registry. In the exemplary embodiment, the registry auditing module **12** is a "Basic Auditing Module" (BAM). In general, BAMs are known in the art, and implement an architecture and interface which provide a consistent data representation for a sensor. As indicated by arrow **22**, they include a "hook" into the audit stream (in this case the registry) and various communication and data-buffering components. BAMs use an XML data representation substantially identical to the IETF standard for IDS systems (See, e.g., Internet Engineering Task Force. Intrusion detection exchange format. On-line publication, <http://www.ietf.org/html.charters/idwg-charter.html>, 2000.), minor syntactical differences. The registry auditing module **12** runs in the background on a Windows™ machine, where it gathers information on registry reads and writes, e.g., the 5 features discussed above. Registry auditing module **12** uses Win32 hooks to tap into the registry and log all reads and writes to the registry. The software uses an architecture substantially identical to SysInternal's Regmon (See, e.g., SysInternals. Regmon for Windows™ NT/9x. Online publication, 2000. <http://www.sysinternals.com/ntw2k/source/regmon.shtml>), and extracts a subset of data available to Regmon. After gathering the registry data, registry auditing module **12** can be configured for two distinct uses. One use is to act as the data source for model generation. When registry auditing module **12** is used as the data source for model generation, its output is sent to a database **18** (as indicated by arrow **24**) where it is stored and later used by the model generator **16** described herein. The second use of registry auditing module **12** is to act as the data source for the real-time anomaly detector **14** described herein. While in this mode, the output of registry auditing module **12** is sent directly to the anomaly detector **14** (indicated by arrow **26**) where it is processed in real time.

The model generation infrastructure consists of two components. A database **18** is used to store all of the collected registry accesses from the training data. A model generator **14** then uses this collected data to create a model of normal usage. The model generator **14** uses one of the two exemplary algorithms discussed above (or other similar algorithm) to build a model that will represent normal usage. It utilizes the data stored in the database **18** which was generated by registry auditing module **12**. The database **18** is described in greater detail is concurrently filed U.S. Application serial No. [not yet known], entitled "System and Methods for Adaptive Model Generation for Detecting Intrusion in Computer Systems," to Andrew Honig, et al., which is incorporated by reference in its entirety herein. (Arrow **28** indicates the flow of data from the data warehouse **18** to the model generator **14**.) The model

14

itself is comprised of serialized Java objects. This allows for a single model to be generated and to easily be distributed to additional machines. Having the model easily deployed to new machines is a desirable feature since in a typical network, many Windows™ machines have similar usage patterns which allow for the same model to be used for multiple machines. The GUI **30** for the model generator using the second embodiment of the algorithm is shown in FIG. **2**. Column **32** indicates the feature name, column **34** indicates the n-value, column **36** indicates the r-value, and column **38** indicates the p-value. Additional details for generating a model are described in U.S. application Ser. No. 10/208,432 filed Jul. 30, 2002 entitled "System and Methods for Detection of New Malicious Executables," to Matthew G. Schulz et al., which is incorporated by reference in its entirety herein.

The anomaly detector **16** will load the normal usage model created by the model generator **14** (as indicated by arrow **29**) and begin reading each record from the output data stream of registry auditing module **12** (arrow **26**). One of the algorithms, as discussed above, is then applied against each record of registry activity. The score generated by the anomaly detection algorithm is then compared by a user configurable threshold to determine if the record should be considered anomalous. A list of anomalous registry accesses are stored and displayed as part of the detector.

The system described herein is a statistical model of expected registry queries and results. If an attacker wanted to usurp a host-based detector, they can a) turn off the detector at the host (and hope no alarms go off elsewhere) or b) they can attack the host based detector by changing its rules or changing its statistical model so it won't alarm.

Accordingly, in order to protect the statistical model of the system, from being attacked, it is put in the registry. The registry is essentially a data base, and the statistical model comprises query values and probabilities. The evaluation of the model first accesses values and probability estimates. This information can be stored in the registry. Hence, any process that attempts to touch the model (for example, to change some values in the model) will be abnormal registry accesses and set off the alarm. Consequently, the system would be protected from having its own model being attacked since it will notice when it is under attack

In order to evaluate the system, data was gathered by running a registry auditing module **12** on a host machine. During training, several programs were run in order to generate normal background traffic. In order to generate normal data for building an accurate and complete training model, it was important to run various applications in various ways. By examining registry traffic, it was discovered that it is not just which programs that are run, but also how they are run that affect registry activity. For example, running ping.exe from the command prompt does not generate registry activity. However, running ping.exe directly from the run dialog box does generate registry activity: By understanding such details of the registry, a more complete training model was built. Beyond the normal execution of standard programs, such as Microsoft™ Word, Internet Explorer, and Winzip, the training also included performing tasks such as emptying the Recycling Bin and using Control Panel.

The training data collected for the experiment was collected on Windows™ NT 4.0 over two days of normal usage. "Normal" usage is defined to mean what is believed to be typical use of a Windows™ platform in a home setting. For example, it was assumed that users would log in, check some internet sites, read some mail, use word processing, then log off. This type of session was taken to be relatively typical of

US 7,913,306 B2

15

computer users. Normal programs are those which are bundled with the operating systems, or are in use by most Windows™ users.

The simulated home use of Windows™ generated a clean (attack-free) dataset of approximately 500,000 records. The system was tested on a full day of test data with embedded attacks executed. This data was comprised of approximately 300,000 records, most of which were normal program executions interspersed with attacks among normal process executions. The normal programs run between attacks were intended to simulate an ordinary Windows™ session. The programs used were, for example, Microsoft™ Word™, Outlook Express™, Internet Explorer™, Netscape™, AOL™ Instant Messenger™.

The attacks run include publicly available attacks such as aimrecover, browslist, bok2ss (back orifice), install.exe xtcp and exe (both for backdoor.XTCP), 10phtcrack, runtack, whackmole, and setuptrojan. Attacks were only run during the one day of testing throughout the day. Among the twelve attacks that were run, four instances were repetitions of the same attack. Since some attacks generated multiple processes there are a total of seventeen distinct processes for each attack. All of the processes (either attack or normal) as well as the number of registry access records in the test data is shown in Table 3 and described in greater detail herein.

The training and testing environments were set up to replicate a simple yet realistic model of usage of Windows™ systems. The system load and the applications that were run were meant to resemble what one may deem typical in normal private settings.

The first exemplary anomaly detection algorithm discussed above in equations (1)-(3) were trained over the normal data. Each record in the testing set was evaluated against this training data. The results were evaluated by computing two statistics: the detection rate and the false positive rate. The performance of the system was evaluated by measuring detection performance over processes labeled as either normal or malicious.

The detection rate reported below is the percentage of records generated by the malicious programs which are labeled correctly as anomalous by the model. The false positive rate is the percentage of normal records which are mislabeled anomalous. Each attack or normal process has many records associated with it. Therefore, it is possible that some

16

records generated by a malicious program will be mislabeled even when some of the records generated by the attack are accurately detected. This will occur in the event that some of the records associated with one attack are labeled normal. Each record is given an anomaly score, S, that is compared to a user defined threshold. If the score is greater than the threshold, then that particular record is considered malicious. FIG. 3 shows how varying the threshold affects the output of detector. The actual recorded scores plotted in the figure are displayed in Table 2.

TABLE 2

Threshold Score	False Positive Rate	Detection Rate
6.847393	0.001192	0.005870
6.165698	0.002826	0.027215
5.971925	0.003159	0.030416
5.432488	0.004294	0.064034
4.828566	0.005613	0.099253
4.565011	0.006506	0.177161
3.812506	0.009343	0.288687
3.774119	0.009738	0.314301
3.502904	0.011392	0.533084
3.231236	0.012790	0.535219
3.158004	0.014740	0.577908
2.915094	0.019998	0.578442
2.899837	0.020087	0.627001
2.753176	0.033658	0.629136
2.584921	0.034744	0.808431
2.531572	0.038042	0.869797
2.384402	0.050454	1.000000

Table 3 is sorted in order to show the results for classifying processes. Information about all processes in testing data including the number of registry accesses and the maximum and minimum score for each record as well as the classification. The top part of the table shows this information for all of the attack processes and the bottom part of the table shows this information for the normal processes. The reference number (by the attack processes) give the source for the attack. Processes that have the same reference number are part of the same attack. [1] AIMCrack. [2] Back Orifice. [3] Backdoor.xtcp. [4] Browse List. [5] Happy 99. [6] IPCrack. [7] LOpht Crack. [8] Setup Trojan.

TABLE 3

Program Name	Number of Records	Maximum Record Value	Minimum Record Value	Classification
LOADWC.EXE[2]	1	8.497072	8.497072	ATTACK
ipccrack.exe[6]	1	8.497072	8.497072	ATTACK
mstinit.exe[2]	11	7.253687	6.705313	ATTACK
bo2kss.exe[2]	12	7.253687	6.62527	ATTACK
runonce.exe[2]	8	7.253384	6.992995	ATTACK
browselist.exe[4]	32	6.807137	5.693712	ATTACK
install.exe[3]	18	6.519455	6.24578	ATTACK
SetupTrojan.exe[8]	30	6.444089	5.756232	ATTACK
AimRecover.exe[1]	61	6.444089	5.063085	ATTACK
happy99.exe[5]	29	5.918383	5.789022	ATTACK
bo2k_1_0_intl.e[2]	78	5.432488	4.820771	ATTACK
_INS0432._MP[2]	443	5.284697	3.094395	ATTACK
xtcp.exe[3]	240	5.265434	3.705422	ATTACK
bo2kcfg.exe[2]	289	4.879232	3.520338	ATTACK
l0phtcrack.exe[7]	100	4.688737	4.575099	ATTACK
Patch.exe[2]	174	4.661701	4.025433	ATTACK
bo2k.exe[2]	883	4.386504	2.405762	ATTACK
systray.exe	17	7.253687	6.299848	NORMAL
CSRSS.EXE	63	7.253687	5.031336	NORMAL
SPOOLSS.EXE	72	7.070537	5.133161	NORMAL
ttssh.exe	12	6.62527	6.62527	NORMAL
winmine.exe	21	6.56054	6.099177	NORMAL

US 7,913,306 B2

17

18

TABLE 3-continued

Program Name	Number of Records	Maximum Record Value	Minimum Record Value	Classification
em_exec.exe	29	6.337396	5.789022	NORMAL
winampa.exe	547	6.11399	2.883944	NORMAL
PINBALL.EXE	240	5.898464	3.705422	NORMAL
LSASS.EXE	2299	5.432488	1.449555	NORMAL
PING.EXE	50	5.345477	5.258394	NORMAL
EXCEL.EXE	1782	5.284697	1.704167	NORMAL
WINLOGON.EXE	399	5.191326	3.198755	NORMAL
rundll32.exe	142	5.057795	4.227375	NORMAL
explore.exe	108	4.960194	4.498871	NORMAL
netscape.exe	11252	4.828566	-0.138171	NORMAL
java.exe	42	4.828566	3.774119	NORMAL
aim.exe	1702	4.828566	1.750073	NORMAL
findfast.exe	176	4.679733	4.01407	NORMAL
TASKMGR.EXE	99	4.650997	4.585049	NORMAL
MSACCESS.EXE	2825	4.629494	1.243602	NORMAL
IEXPLORE.EXE	194274	4.628190	-3.419214	NORMAL
NTVDM.EXE	271	4.59155	3.584417	NORMAL
CMD.EXE	116	4.579538	4.428045	NORMAL
WINWORD.EXE	1541	4.457119	1.7081	NORMAL
EXPLORER.EXE	53894	4.31774	-1.704574	NORMAL
msmsgs.exe	7016	4.177509	0.334128	NORMAL
OSA9.EXE	705	4.163361	2.584921	NORMAL
MYCOME 1.EXE	1193	4.035649	2.105155	NORMAL
wscript.exe	527	3.883216	2.921123	NORMAL
WINZIP32.EXE	3043	3.883216	0.593845	NORMAL
notepad.exe	2673	3.883216	1.264339	NORMAL
POWERPNT.EXE	617	3.501078	-0.145078	NORMAL
AcroRd32.exe	1598	3.412895	0.393729	NORMAL
MDM.EXE	1825	3.231236	1.680336	NORMAL
ttermpro.exe	1639	2.899837	1.787768	NORMAL
SERVICES.EXE	1070	2.576196	2.213871	NORMAL
REGMON.EXE	259	2.556836	1.205416	NORMAL
RPCSS.EXE	4349	2.250997	0.812288	NORMAL

The process of setting the threshold is described herein. If the threshold is set at 8.497072, the processes LOAD-WC.EXE and ipccrack.exe are labeled as malicious and would detect the Back Orifice and IPCrack attacks. Since none of the normal processes have scores that high, we would have no false positives. If we lower the threshold to 6.444089, we would have detected several more processes from Back Orifice and the BrowseList, BackDoor.xtcp, SetupTrojan and AimRecover attacks. However, at this level of threshold, the following processes would be labeled as false positives: systray.exe, CSRSS.EXE, SPOOLSS.EXE, ttssh.exe, and winmine.exe.

By varying the threshold for the inconsistency scores on records, we were able to demonstrate the variability of the detection rate and false positive rate. The false positive rate versus the detection rate was plotted in an ROC (Receiver Operator Characteristic) curve 52 shown in Table 2 and the plot 50 in FIG. 3, in which the false positive rate 54 is plotted against the detection rate 56.

Another exemplary embodiment is described herein. The systems and method described above, uses Windows™ registry accesses, which is an example of a general means of detecting malicious uses of a host computer. However, other systems, such as Linux/Unix, do not use a registry. In those cases, a file system sensor would be used. Accesses to the file system provides an audit source of data (i.e., whenever an application is run, any and all things accessed are files, e.g., the main executable files are accessed, and the project files are accessed). This audit source can be observed, and a “normal baseline model” built, and then used for detecting abnormal file system accesses.

It will be understood that the foregoing is only illustrative of the principles of the invention, and that various modifications can be made by those skilled in the art without departing from the scope and spirit of the invention.

APPENDIX

The software listed herein is provided in an attached CD-Rom. The contents of the CD-Rom are incorporated by reference in their entirety herein.

A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by any one of the patent disclosure, as it appears in the Patent and Trademark Office patent files or records, but otherwise reserves all copyright rights whatsoever.

PAD (Probabilistic Anomaly Detection) is a package that detects anomalies in fixed length records of discrete values.

The basic idea behind PAD is that it trains over a data set and then checks to see if new observed records are “consistent” with the data set. There are two types of consistency checks. First order consistency checks evaluate whether or not a single feature is consistent with other features in the dataset. Second order consistency checks evaluate if a pair of features is consistent with the data set. All consistency checks are evaluated by computing a predictive probability. This is done by estimating a multinomial using counts observed from the data set and then estimating the probability of seeing the observation.

Let us assume records of length n are being observed. That is, each record can be written as a set of random variables (X_1, X_2, \dots, X_n) . Each first order consistency check can be denoted as computing $P(X_i)$. This probability is trained over a data set, and then used to predict elements in another dataset. These two datasets can be in fact the same. To train the probability distribution, the counts of the observed symbols in the data set are collected. The second order consistency checks are $P(X_i, X_j)$. In this case, the counts of X_i are collected when X_j is observed. Note that there is a separate set of counts for each X_j . Accordingly, second order

US 7,913,306 B2

19

consistency checks take a significant amount of memory relative to first order consistency checks.

All of these probability estimates are obtained using the multinomial estimator presented in Friedman, Singer 1999 (incorporated by reference, above). The basic idea of the estimator is that it explicitly takes into account the probability of seeing an unobserved symbol.

The term C is the probability of observing an already observed symbol. Thus $(1-C)$ is the probability of observing an unobserved symbol. For the observed symbols, a Dirichlet is used to estimate the probabilities for the counts. If N is the total number of observations, and N_i is the observations of symbol i , if α is the "pseudo count" which is added to the count of each observed symbol, and k^0 is the number of observed symbols and L is the total number of symbols, the probability is as follows:

For an observed symbol i , the probability is: $C(N_i + \alpha) / (k^0 \alpha + N)$. For an unobserved symbol, the probability is: $(1-C) * (1 / (L - k^0))$. See equations (1) and (2) above, and the routine "updateC" in Classifier.c. In the second case, the probability is spread among possible unobserved symbols.

Since each feature in the record may have a different set of possible outcomes, if P is the probability estimated from the consistency check, the following term is reported: $\log(P / (1/L))$. This normalizes the consistency check to take into account the number of possible outcomes L . In general, C should be set as described in Friedman, Singer 1999.

However, this causes some overflow/underflow problems in the general case. Instead, the current version of the algorithm uses a heuristic to set C . This is done after observing the counts. This is called SIMPLE_MODE in the implementation. In SIMPLE_MODE, C is set to $C = N / (N + L - k^0)$. In addition, in SIMPLE_MODE, there is a variable OBSERVED_BIAS which adjusts the value of C toward observed or unobserved symbols. When OBSERVED_BIAS = 0.0, there is no bias. Positive values increase the probability mass of observed symbols while negative values decrease the probability mass of unobserved symbols. For non-zero OBSERVED_BIAS, the value of C is adjusted so that the new value of C , C^* is given as follows $C^* = C / (C + ((1-C) * \exp(-OBSERVED_BIAS)))$.

Package Installation and Quick Start: To install the package, the following steps should be performed:

1. Unpack the files.
2. cd into the src/subdirectory.
3. type make

To test the package, the following steps should be performed:

4. cd into data/subdirectory
5. type `./src/pad -e -g globalsFile.txt -p sampleInput.txt sampleInput.txt`

Subsequently, the following output should be provided:

```

a aa aaa zzzz: 0.031253 1.386294 0.490206 0.693147 0.000000 0.031253
0.000000 0.000000 0.000000 1.386294 0.980829 0.980829 0.000000 0.490206
0.693147 0.693147 0.000000 0.693147 0.875469 0.875469: test
b aa fff dddd: 0.436718 1.386294 0.202524 0.000000 0.619039 0.436718
0.000000 0.000000 0.632523 1.386294 0.632523 0.000000 0.000000 0.202524
0.470004 0.000000 -0.000000 0.000000 -0.000000 0.000000: test1
c aa fff tttt: 0.031253 1.386294 0.202524 0.000000 0.000000 0.031253
0.000000 0.000000 0.000000 1.386294 0.632523 0.000000 0.000000 0.202524
0.470004 0.000000 0.000000 0.000000 -0.000000 0.000000: test2
g aa aaa zzzz: 0.031253 1.386294 0.490206 0.693147 0.000000 0.031253
0.000000 0.000000 0.000000 1.386294 0.980829 0.980829 0.000000 0.490206
0.693147 0.693147 0.000000 0.693147 0.875469 0.875469: test3
b aa aaa zzzz: 0.436718 1.386294 0.490206 0.693147 0.619039 0.436718
0.000000 0.000000 0.632523 1.386294 0.980829 0.980829 0.000000 0.490206
0.693147 0.693147 -0.000000 0.693147 0.875469 0.875469: test4

```

20

The next steps are then performed:

6. Type `./src/pad -e -g globalsFile.txt -w temp.cla sampleInput.txt`

This should create a file called temp.cla which is the trained model (classifier) from the sample input.

7. Type `./src/pad -r -p sampleInput.txt temp.cla`

This should provide the same output as above.

Usage Instructions The executable has two modes: "examples" mode (using the `-e` option) which reads in examples from a file and trains the model using that data, and "read" mode (using the `-r` option) which reads in a model from a file. The executable requires one argument, which is either the file or examples. The globals file (specified with the `-g` option) defines all of the global variables. These include the number of columns, the file names containing the column symbol definitions. Note that when reading in a model, the column symbol files must be in the current directory.

Options: Command line options: In addition to `-r` and `-e` which set the mode, the following are options that can be used from the command line:

<code>-g FILE</code>	set globals file
<code>-v</code>	toggle verbose output
<code>-s</code>	toggle use of second order predictors
<code>-w FILE</code>	write classifier to file.
<code>-p FILE</code>	predict files

Globals File Options: Below is the globals file included in the distribution. All lines starting with "`#`" are comments.

```

# Globals Definition
#The input Symbols.
NUM_COLUMNS 4
#Set Simple Mode
SIMPLE_MODE TRUE
#Set Use Second Order Consistency Checks
USE_SECOND_ORDER TRUE
#Set Verbose mode
VERBOSE FALSE
#Allow unknown symbols in testing
ALLOW_UNKNOWN_SYMBOLS TRUE
#Set Initial Count for Predictors. This is the virtual count
#that is added to all observed symbols.
INITIAL_PREDICTION_COUNT 1
#Set the bias to observed symbols
OBSERVED_BIAS 0.0
#Set the Column Symbol Files
COLUMN_SYMBOL_FILENAME 1:C1.txt
COLUMN_SYMBOL_FILENAME 2:C2.txt
COLUMN_SYMBOL_FILENAME 3:C3.txt

```

-continued

```

COLUMN_SYMBOL_FILENAME 4:C4.txt
#Sets the number of symbols in a column
COLUMN_NUM_SYMBOLS 1:40
#Sets the classifier to ignore a column
IGNORE_COLUMN 2
#Sets the symbol that represents an ignored symbol
IGNORE_SYMBOL **
#Sets the symbol to represent an unknown symbol
UNKNOWN_SYMBOL UKS
    
```

Input file description: Each line of the input file corresponds to a record. Each record consists of the features in a record separated by a space. This is followed by a tab after which there is an optional comment. This comment is preserved in prediction and can be used in experiments to keep track of the type of a record and where it came from. Below is the sample input:

```

a aa aaa ZZZZ          test
b aa fff dddd          test1
c aa fff tttt          test2
g aa aaa ZZZZ          test3
b aa aaa ZZZZ          test4
    
```

A symbol file defines all of the possible symbols. Each symbol is on a separate line in the file. A sample symbol file is below:

```

aaa
ccc
fff
ggg
    
```

There are several options related to symbol files. The IGNORE_COLUMN can set the classifier to ignore a column completely. In this case, each element of the column gets mapped to a special symbol which is ignored in the model. In a single record, some of the fields can be set to a special symbol (by default “**”) which tells the classifier to ignore its value. A special symbol (by default “UKS”) denotes an unknown symbol. In training, unknown symbols are not allowed and will cause the program to exit. In testing, the unknown symbols are treated as if it is a symbol that has observed count of 0. The option ALLOW_UNKNOWN_SYMBOLS toggles the automatic mapping of unseen symbols to the special unknown symbol during testing. This makes running experiments easier because it is not necessary to have the symbol files contain the data in the test set.

Package Description The software package contains the following:

```

/src/      directory consisting of all of the source files
/data/     directory consisting of a small sample input
/registry/ directory consisting of data and scripts to do
           the registry experiments.
/papers/   directory containing relevant papers to pad.
           In the/src/directory there are the following files:

Classifier.c  File that defines the Classifier (model)
Classifier.h  Header file for Classifier.c
Column.c     File that defines a single consistency check
Column.h     Header file for Column.c
Globals.c    Defines the global variables
    
```

-continued

```

Globals.h      Header file for Globals.c
HashTable.c    Hashable implementation
HashTable.h    Header file for HashTable.c
includes.h     Include file for all files
Input.c       Implementation of I/O
Input.h       Header file for Input.c
Makefile      Makefile
memwatch.c    Package to detect memory leaks
memwatch.h    Package to detect memory leaks
pad.c         Main executable file
pad.h         Header file for pad.c
SymbolTable.c Symbol Table implementation for mapping
              Symbols to Integers
SymbolTable.h Header file for SymbolTable.c
    
```

Registry Experiments: The following steps should be performed:

1. cd registry/
2. type ./src/pad -e -g regGlobs.txt -w model.cla registry.txt
This creates a file called model.cla which is the model that is trained on 800 k registry records. It should take about 190 MB of memory to train the model.
3. type ./src/pad -r -p registry.txt model.cla>predictions.txt
This reads in the model and evaluates all of the records. The file predictions.txt contains the values for all of the consistency checks.
4. Type ./computeResults.pl predictions.txt>final-predictions.txt
This determines the minimum value for a consistency check for each record and puts on each line this value and the comment.
5. Type sort -n final-predictions.txt>sorted-final-predictions.txt
This sorts the records in order of least consistent. This is the final output of the experiments.
6. Type ./computeROC.pl sorted-final-predictions.txt>roc.txt
This computes ROC points for the experiments.

We claim:

1. A method for detecting intrusions in the operation of a computer system comprising:
 - (a) gathering features from records of normal processes that access the file system of the computer;
 - (b) generating a probabilistic model of normal computer system usage based on occurrences of the features and determining the likelihood of observing an event that was not observed during the gathering of features from the records of normal processes; and
 - (c) analyzing features from a record of a process that accesses the file system to detect deviations from normal computer system usage to determine whether the access to the file system is an anomaly.
2. The method according to claim 1, wherein the step of gathering features from records of normal processes that access the file system of the computer comprises gathering a feature corresponding to a name of a process accessing the file system of the computer.
3. The method according to claim 1, wherein gathering features from records of normal processes that access the file system of the computer comprises gathering a feature corresponding to a type of query being sent to the file system of the computer.
4. The method according to claim 3, wherein gathering features from records of normal processes that access the file system of the computer comprises gathering a feature corresponding to an outcome of a query being sent to the file system of the computer.

US 7,913,306 B2

23

5. The method according to claim 1, wherein gathering features from records of normal processes that access the file system of the computer comprises gathering a feature corresponding to a name of a key being accessed in the file system of the computer.

6. The method according to claim 5, wherein gathering features from records of normal processes that access the file system of the computer comprises gathering a feature corresponding to a value of the key being accessed.

7. The method according to claim 1, wherein generating a probabilistic model of normal computer system usage comprises determining a likelihood of observing a feature in the records of processes that access the file system of the computer.

8. The method according to claim 7, wherein determining a likelihood of observing a feature comprises determining a conditional probability of observing a first feature in the

24

records of processes that access the file system of the computer given an occurrence of a second feature is the records.

9. The method according to claim 1, wherein analyzing a record of a process that accesses the file system of the computer comprises, for each feature, performing a check to determine if a value of the feature has been previously observed for the feature.

10. The method according to claim 9, further comprising, if the value of the feature has not been observed, computing a score based on a probability of observing the value of the feature.

11. The method according to claim 9, further comprising, if the score is greater than a predetermined threshold, labeling the access to the file system of the computer as anomalous and labeling the process that accessed the file system of the computer as malicious.

* * * * *

9



US008074115B2

(12) **United States Patent**
Stolfo et al.

(10) **Patent No.:** **US 8,074,115 B2**

(45) **Date of Patent:** **Dec. 6, 2011**

(54) **METHODS, MEDIA AND SYSTEMS FOR
DETECTING ANOMALOUS PROGRAM
EXECUTIONS**

(56) **References Cited**

(75) Inventors: **Salvatore J. Stolfo**, Ridgewood, NJ
(US); **Angelos D. Keromytis**, New York,
NY (US); **Stelios Sidiroglou**, New York,
NY (US)

(73) Assignee: **The Trustees of Columbia University
in the City of New York**, New York, NY
(US)

U.S. PATENT DOCUMENTS

5,968,113	A *	10/1999	Haley et al.	714/38
6,079,031	A *	6/2000	Haley et al.	714/38
6,154,876	A *	11/2000	Haley et al.	717/133
7,155,708	B2 *	12/2006	Hammes et al.	717/155
7,490,268	B2	2/2009	Keromytis et al.	
7,496,898	B1 *	2/2009	Vu	717/127
7,639,714	B2	12/2009	Stolfo et al.	
2005/0108562	A1 *	5/2005	Khazan et al.	713/200

OTHER PUBLICATIONS

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

Hangal et al., Tracking down software bugs using automatic anomaly detection, Proceedings of the 24th international conference on software engineering, May 2002, pp. 291-301. *

(21) Appl. No.: **12/091,150**

(Continued)

(22) PCT Filed: **Oct. 25, 2006**

Primary Examiner — Nadeem Iqbal

(86) PCT No.: **PCT/US2006/041591**

(74) *Attorney, Agent, or Firm* — Byrne Poh LLP

§ 371 (c)(1),
(2), (4) Date: **Jun. 15, 2009**

(57) **ABSTRACT**

(87) PCT Pub. No.: **WO2007/050667**

PCT Pub. Date: **May 3, 2007**

Methods, media, and systems for detecting anomalous program executions are provided. In some embodiments, methods for detecting anomalous program executions are provided, comprising: executing at least a part of a program in an emulator; comparing a function call made in the emulator to a model of function calls for the at least a part of the program; and identifying the function call as anomalous based on the comparison. In some embodiments, methods for detecting anomalous program executions are provided, comprising: modifying a program to include indicators of program-level function calls being made during execution of the program; comparing at least one of the indicators of program-level function calls made in the emulator to a model of function calls for the at least a part of the program; and identifying a function call corresponding to the at least one of the indicators as anomalous based on the comparison.

(65) **Prior Publication Data**

US 2010/0023810 A1 Jan. 28, 2010

Related U.S. Application Data

(60) Provisional application No. 60/730,289, filed on Oct. 25, 2005.

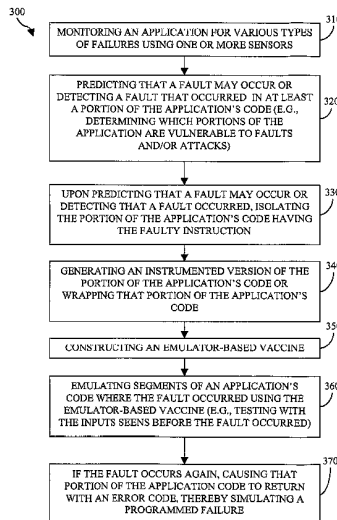
(51) **Int. Cl.**
G06F 11/00 (2006.01)

(52) **U.S. Cl.** **714/38.1**

(58) **Field of Classification Search** 714/2-10,
714/25-29, 32, 33, 37-39, 47

See application file for complete search history.

42 Claims, 8 Drawing Sheets



US 8,074,115 B2

Page 2

OTHER PUBLICATIONS

- Chan et al., A machine learning approach to anomaly detection, Technical Report, Dept. of computer science, Florida institute of technology, Mar. 2003, pp. 1-13.*
- M. Chew and D. Song, Mitigating Buffer Overflows by Operating System Randomization, Technical Report CMUCS-02-197, Carnegie Mellon University, Dec. 2002.
- V. Prevelakis, A Secure Station for Network Monitoring and Control, In Proceedings of the 8th USENIX Security Symposium, Aug. 1999.
- J. Reynolds, J. Just, L. Clough, and R. Maglich, On-Line Intrusion Detection and Attack Prevention Using Diversity, Generate-and-Test, and Generalization, In Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS), Jan. 2003.
- H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh, on the Effectiveness of Address-Space Randomization, In Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS), pp. 298-307, Oct. 2004.
- S. Sidiroglou, M. Locasto, S. Boyd, and A. Keromytis, Building A Reactive Immune System for Software Services, In Proceedings of the 11th USENIX Annual Technical Conference, Apr. 2005.
- M. Stamp, Risk of Monoculture, Communications of the ACM, 47(3):120, Mar. 2004.
- Using Network-Based Application Recognition and ACLs for Blocking the "Code Red" Worm, Technical report, Cisco Systems, Inc.
- Aleph One, Smashing the stack for fun and profit, Phrack, 7(49), 1996.
- K. Ashcraft and D. Engler, Detecting Lots of Security Holes Using System-Specific Static Analysis, In Proceedings of the IEEE Symposium on Security and Privacy, May 2002.
- S. M. Bellovin, Distributed Firewalls, ;login: magazine, special issue on security, Nov. 1999.
- M. Bhattacharyya, M. G. Schultz, E. Eskin, S. Hershkop, and S. J. Stolfo, MET: An Experimental System for Malicious Email Tracking, In Proceedings of the New Security Paradigms Workshop (NSPW), pp. 1-12, Sep. 2002.
- Bulba and Kil3r, Bypassing StackGuard and StackShield, Phrack, 5(56), May 2000.
- B. Chess, Improving Computer Security Using Extended Static Checking, In Proceedings of the IEEE Symposium on Security and Privacy, May 2002.
- M. Christodorescu and S. Jha, Static Analysis of Executables to Detect Malicious Patterns, In Proceedings of the 12th USENIX Security Symposium, pp. 169-186, Aug. 2003.
- F. Cohen, Computer Viruses: Theory and Practice, Computers & Security, 6:22-35, Feb. 1987.
- C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman, Formatguard: Automatic protection from printf format string vulnerabilities, In Proceedings of the 10th USENIX Security Symposium, Aug. 2001.
- C. Cowan, S. Beattie, C. Pu, P. Wagle, and V. Gligor, SubDomain: Parsimonious Security for Server Appliances, In Proceedings of the 14th USENIX System Administration Conference (LISA 2000), Mar. 2000.
- C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks, In Proceedings of the 7th USENIX Security Symposium, Jan. 1998.
- D. Engler and K. Ashcraft, RacerX: Effective, Static Detection of Race Conditions and Deadlocks, Proceedings of ACM SOS, Oct. 2003.
- S. Forrest, A. Somayaji, and D. Ackley, Building Diverse Computer Systems, In Proceedings of the 6th HotOS Workshop, 1997.
- M. Frantzen and M. Shuey, StackGhost: Hardware facilitated stack protection, In Proceedings of the 10th USENIX Security Symposium, pp. 55-66, Aug. 2001.
- T. Garfinkel, Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools, In Proceedings of the Symposium on Network and Distributed Systems Security (SNDSS), pp. 163-176, Feb. 2003.
- I. Goldberg, D. Wagner, R. Thomas, and E. Brewer, A Secure Environment for Untrusted Helper Applications, In Proceedings of the 1996 USENIX Annual Technical Conference, 1996.
- S. Ioannidis, A. Keromytis, S. Bellovin, and J. Smith, Implementing a Distributed Firewall, In Proceedings of the ACM Computer and Communications Security (CCS) Conference, pp. 190-199, Nov. 2000.
- R. Janakiraman, M. Waldvogel, and Q. Zhang, Indra: A peer-to-peer approach to network intrusion detection and prevention, In Proceedings of the IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Workshop on Enterprise Security, Jun. 2003.
- R. Jones and P. Kelly, Backwards-compatible bounds checking for arrays and pointers in C programs, In Third International Workshop on Automated Debugging, 1997.
- J. Just, L. Clough, M. Danforth, K. Levitt, R. Maglich, J. C. Reynolds, and J. Rowe, Learning Unknown Attacks—A Start, In Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID), Oct. 2002.
- J. Kephart, A Biologically Inspired Immune System for Computers, In Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems, pp. 130-139. MIT Press, 1994.
- M. Kodialam and T. V. Lakshman, Detecting Network Intrusions via Sampling: A Game Theoretic Approach, In Proceedings of the 22nd Annual Joint Conference of IEEE Computer and Communication Societies (INFOCOM), Apr. 2003.
- D. Laroche and D. Evans, Statically Detecting Likely Buffer Overflow Vulnerabilities, In Proceedings of the 10th Security Symposium, pp. 177-190, Aug. 2001.
- E. Larson and T. Austin, High Coverage Detection of Input-Related Security Faults, In Proceedings of the 12th Security Symposium, pp. 121-136, Aug. 2003.
- K. Lhee and S. J. Chapin, Type-Assisted Dynamic Buffer Overflow Detection, In Proceedings of the 11th Security Symposium, pp. 81-90, Aug. 2002.
- M.-J. Lin, A. Ricciardi, and K. Marzullo, A New Model for Availability in the Face of Self-Propagating Attacks, In Proceedings of the New Security Paradigms Workshop, Nov. 1998.
- A. J. Malton, The Denotational Semantics of a Functional Tree-Manipulation Language, Computer Languages, 19 (3):157-168, 1993.
- T. C. Miller and T. de Raadt, strcpy and strcat: Consistent, Safe, String Copy and Concatenation, In Proceedings of the USENIX Annual Technical Conference, Freenix Track, Jun. 1999.
- D. Moore, C. Shanning, and K. Claffy, Code-Red: a case study on the spread and victims of an Internet worm, In Proceedings of the 2nd Internet Measurement Workshop (IMW), pp. 273-284, Nov. 2002.
- D. Moore, C. Shannon, G. Voelker, and S. Savage, Internet Quarantine: Requirements for Containing Self-Propagating Code, In Proceedings of the IEEE Infocom Conference, Apr. 2003.
- C. Nachenberg, Computer Virus-Coevolution, Communications of the ACM, 50(1):46-51, 1997.
- D. Nofjiri, J. Rowe, and K. Levitt, Cooperative Response Strategies for Large Scale Attack Mitigation, In Proceedings of the 3rd DARPA Information Survivability Conference and Exposition (DISCEX), pp. 293-302, Apr. 2003.
- D. S. Peterson, M. Bishop, and R. Pandey, A Flexible Containment Mechanism for Executing Untrusted Code, In Proceedings of the 11th USENIX Security Symposium, pp. 207-225, Aug. 2002.
- M. Prasad and T. Chiueh, A Binary Rewriting Defense Against Stack-based Buffer Overflow Attacks, In Proceedings of the USENIX Annual Technical Conference, pp. 211-224, Jun. 2003.
- V. Prevelakis and D. Spinellis, Sandboxing Applications, In Proceedings of the USENIX Technical Annual Conference, Freenix Track, pp. 119-126, Jun. 2001.
- N. Provos, M. Friedl, and P. Honeyman, Preventing Privilege Escalation, In Proceedings of the 12th USENIX Security Symposium, pp. 231-242, Aug. 2003.
- J. Reynolds, J. Just, E. Lawson, L. Clough, and R. Maglich, The Design and Implementation of an Intrusion Tolerant System, In Proceedings of the International Conference on Dependable Systems and Networks (DSN), Jun. 2002.
- M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod, Using the SimOS Machine Simulator to Study Complex Computer Systems, Modeling and Computer Simulation, 7(1):78-103, 1997.

US 8,074,115 B2

Page 3

- R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVane, Model-Carrying Code: A Practice Approach for Safe Execution of Untrusted Applications, in Proceedings of ACM SOSP, Oct. 2003.
- N. Nethercote and J. Seward, Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation, PLDI '07, Jun. 2007.
- J. F. Shoch and J. A. Hupp, The "worm" programs—early experiments with a distributed computation, Communications of the ACM, 22(3):172-180, Mar. 1982.
- Song, R. Malan, and R. Stone, A Snapshot of Global Internet Worm Activity, Technical report, Arbor Networks, Nov. 2001.
- E. H. Spafford, The Internet Worm Program: An Analysis, Technical Report CSD-TR-823, Purdue University, 1988.
- S. Staniford, V. Paxson, and N. Weaver, How to Own the Internet in Your Spare Time, In Proceedings of the 11th USENIX Security Symposium, pp. 149-167, Aug. 2002.
- T. Toth and C. Kruegel, Connection-history Based Anomaly Detection. In Proceedings of the IEEE Workshop on Information Assurance and Security, Jun. 2002.
- H. Toyozumi and A. Kara, Predators: Good Will Mobile Codes Combat against Computer Viruses, In Proceedings of the New Security Paradigms Workshop (NSPW), pp. 13-21, Sep. 2002.
- J. Twycross and M. M. Williamson, Implementing and testing a virus throttle, In Proceedings of the 12th USENIX Security Symposium, pp. 285-294, Aug. 2003.
- G. Venkitachalam and B.-H. Lim, Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor.
- C. Wang, J. C. Knight, and M. C. Elder, on Computer Viral Infection and the Effect of Immunization, In Proceedings of the 16th Annual Computer Security Applications Conference (ACSAC), pp. 246-256, 2000.
- A. Whitaker, M. Shaw, and S. D. Gribble, Scale and Performance in the Denali Isolation Kernel, In Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI), Dec. 2002.
- J. Wilander and M. Kamkar, A Comparison of Publicly Available Tools for Dynamic Intrusion Prevention, In Proceedings of the Symposium on Network and Distributed Systems Security (SNDSS), pp. 123-130, Feb. 2003.
- M. Williamson, Throttling Viruses: Restricting Propagation to Defeat Malicious Mobile Code, Technical Report HPL-2002-172, HP Laboratories Bristol, 2002.
- C. C. Zou, L. Gao, W. Gong, and D. Towsley, Monitoring and Early Warning for Internet Worms, In Proceedings of the 10th ACM International Conference on Computer and Communications Security (CCS), pp. 190-199, Oct. 2003.
- C. C. Zou, W. Gong, and D. Towsley, Code Red Worm Propagation Modeling and Analysis, In Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS), pp. 138-147, Nov. 2002.
- S. Hangal and M. Lam, Tracking Down Software Bugs Using Automatic Anomaly Detection, ICSE '02, May 19-25, 2002, pp. 291-301.
- P. Chan, M. Mahoney, and M. Arshad, A Machine Learning Approach to Anomaly Detection, Technical Report CS-2003-06, Department of Computer Sciences, Florida Institute of Technology, Mar. 29, 2003.
- International Search Report and Written Opinion, International Application No. PCT/US06/41591, dated Jun. 25, 2008.
- F. Apap, A. Honig, S. Hershkop, E. Eskin, and S. Stolfo, Detecting malicious software by monitoring anomalous windows registry accesses, Proceedings of the Fifth International Symposium on Recent Advances in Intrusion Detection (RAID 2002), 2002.
- D. Denning, An intrusion detection model, IEEE Transactions on Software Engineering, SE-13:222-232, Feb. 1987.
- E. Eskin, Anomaly detection over noisy data using learned probability distributions, Proceedings of the Seventeenth International Conference on Machine Learning (ICML-2000), 2000.
- S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff, A sense of self for unix processes, Proceedings of the IEEE Symposium on Research in Security and Privacy, pp. 120-128, 1996.
- N. Friedman and Y. Singer, Efficient bayesian parameter estimation in large discrete domains, Advances in Neural Information Processing Systems, 11, 1999.
- S. Hofmeyr, S. Forrest, and A. Somayaji, Intrusion detection using sequences of system calls, Journal of Computer Security, 6:151-180, 1998.
- H. Javitz and A. Valdes, The nides statistical component: Description and justification, Technical Report, SRI International, Computer Science Laboratory, 1993.
- W. Lee, S. Stolfo, and P. Chan, Learning patterns from unix processes execution traces for intrusion detection, AAAI Workshop on AI Approaches to Fraud Detection and Risk Management, pp. 50-56, 1997.
- W. Lee, S. Stolfo, and K. Mok, A data mining framework for building intrusion detection models, IEEE Symposium on Security and Privacy, pp. 120-132, 1999.
- W. Lee, S. Stolfo, and K. Mok, Data mining in work flow environments: Experiences in intrusion detection, Proceedings of the 1999 Conference on Knowledge Discovery and Data Mining (KDD-99), 1999.
- M. Mahoney and P. Chan, Detecting novel attacks by identifying anomalous network packet headers, Technical Report CS-2001-2, 2001.
- B. Scholkopf, J. Platt, J. Shawe-Taylor, A. Smola, and R. Williamson, Estimating the support of a high dimensional distribution, Neural Computation, 13(7):1443-1472, 2001.
- C. Warrender, S. Forrest, B. Pearlmutter, Detecting intrusions using system calls: Alternative data models, IEEE Symposium on Security and Privacy, pp. 133-145, 1999.
- A. Honig, A. Howard, E. Eskin, and S. Stolfo, Adaptive model generation: An architecture for the deployment of data mining-based intrusion detection systems, in Data Mining for Security Applications, Kluwer, 2002.
- S. White, Open problems in computer virus research, in Virus Bulletin Conference, 1998.
- CERT Advisory CA-2003-21: W32/Blaster Worm, <http://www.cert.org/advisories/CA-2003-20.html>, Aug. 2003.
- A. Baratloo, N. Singh, and T. Tsai, Transparent Run-Time Defense Against Stack Smashing Attacks, In Proceedings of the Annual Technical Conference, Jun. 2000.
- E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi, Randomized Instruction Set Emulation to Distrust Binary Code Injection Attacks, in 10th ACM Conference on Computer and Communications Security (CCS), Oct. 2003.
- D. Bruening, T. Garnett, and S. Amarasinghe, An Infrastructure for Adaptive Dynamic Optimization, In Proceedings of the International Symposium on Code Generation and Optimization, pp. 265-275, 2003.
- G. Candea and A. Fox, Crash-Only Software, in Proceedings of the 9th Workshop on Hot Topics in Operating Systems, May 2003.
- H. Chen and D. Wagner, MOPS: an Infrastructure for Examining Security Properties of Software, In Proceedings of the ACM Computer and Communications Security (CCS) Conference, pp. 235-244, Nov. 2002.
- S. A. Crosby and D. S. Wallach, Denial of Service via Algorithmic Complexity Attacks, In Proceedings of the 12th USENIX Security Symposium, pp. 29-44, Aug. 2003.
- B. Demsky and M. C. Rinard, Automatic Detection and Repair of Errors in Data Structures, In Proceedings of the 18th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, Oct. 2003.
- G. W. Dunlap, S. King, S. Cinar, M. A. Basrai, and P. M. Chen, ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay, In Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI), Feb. 2002.
- C. Cowan et al., StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, In Proceedings of the 7th Security Symposium, Jan. 1998.
- T. Garfinkel and M. Rosenblum, A Virtual Machine Introspection Based Architecture for Intrusion Detection, in 10th ISOC Symposium on Network and Distributed Systems Security (SNDSS), Feb. 2003.
- T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, Cyclone: A safe dialect of C, In Proceedings of the Annual Technical Conference, pp. 275-288, Jun. 2002.

US 8,074,115 B2

Page 4

- G. S. Kc, A. D. Keromytis, and V. Prevelakis, Countering Code-Injection Attacks With Instruction-Set Randomization, in 10th ACM Conference on Computer and Communications Security (CCS), Oct. 2003.
- S. T. King and P. M. Chen, Backtracking Intrusions, In 19th ACM Symposium on Operating Systems Principles (SOSP), Oct. 2003.
- S. T. King, G. Dunlap, and P. Chen, Operating System Support for Virtual Machines, In Proceedings of the Annual Technical Conference, Jun. 2003.
- V. Kiriansky, D. Bruening, and S. Amarasinghe, Secure Execution Via Program Shepherding, In Proceedings of the 11th Security Symposium, Aug. 2002.
- D. Mosberger and T. Jin, httpf: A tool for measuring web server performance, In First Workshop on Internet Server Performance, pp. 59-67, ACM, Jun. 1998.
- N. Nethercote and J. Seward, Valgrind: A Program Supervision Framework, In Electronic Notes in Theoretical Computer Science, vol. 89, 2003.
- J. Newsome and D. Dong, Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software, In The 12th Annual Network and Distributed System Security Symposium, Feb. 2005.
- J. Oplinger and M. S. Lam, Enhancing Software Reliability with Speculative Threads, In Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X), Oct. 2002.
- N. Provos, Improving Host Security with System Call Policies, In Proceedings of the 12th USENIX Security Symposium, pp. 257-272, Aug. 2003.
- M. Rinard, C. Cadar, D. Dumitran, D. Roy, and T. Leu, A Dynamic Technique for Eliminating Buffer Overow Vulnerabilities (and Other Memory Errors), In Proceedings 20th Annual Computer Security Applications Conference (ACSAC), Dec. 2004.
- M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and J. W Beebe, Enhancing Server Availability and Security Through Failure-Oblivious Computing, In Proceedings 6th Symposium on Operating Systems Design and Implementation (OSDI), Dec. 2004.
- A. Rudys and D. S. Wallach, Transactional Rollback for Language-Based Systems, In ISOC Symposium on Network and Distributed Systems Security (SNDSS), Feb. 2001.
- A. Rudys and D. S. Wallach, Termination in Language-based Systems, ACM Transactions on Information and System Security, 5(2), May 2002.
- S. Sidiroglou and A. D. Keromytis, A Network Worm Vaccine Architecture. In Proceedings of the IEEE Workshop on Enterprise Technologies: Infrastructure for Collaborative Enterprises (WET-ICE), Workshop on Enterprise Security, pp. 220-225, Jun. 2003.
- A. Smimov and T. Chiueh, DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks, In The 12th Annual Network and Distributed System Security Symposium, Feb. 2005.
- G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, Secure program execution via dynamic information flow tracking, SIGOPS Oper. Syst. Rev., 38(5):85-96, 2004.
- T. Toth and C. Kruegel, Accurate Buffer Overflow Detection via Abstract Payload Execution, In Proceedings of the 5th Symposium on Recent Advances in Intrusion Detection (RAID), Oct. 2002.
- N. Wang, M. Fertig, and S. Patel, Y-Branches: When You Come to a Fork in the Road, Take It, In Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques, Sep. 2003.
- J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, Separating Agreement from Execution for Byzantine Fault Tolerant Services, in Proceedings of ACM SOSP, Oct. 2003.
- A. Avizienis, The n-version approach to fault-tolerant software, IEEE Transactionson Software Engineering, 11 (12):1491-1501, 1985.
- S. Bhatkar, D. C. DuVarney, and R. Sekar, Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits, In Proceedings of the 12th Security Symposium, pp. 105-120, Aug. 2003.
- S. Brilliant, J. C. Knight, and N. G. Leveson, Analysis of Faults in an N-Version Software Experiment, IEEE Transactions on Software Engineering, 16(2), Feb. 1990.

* cited by examiner

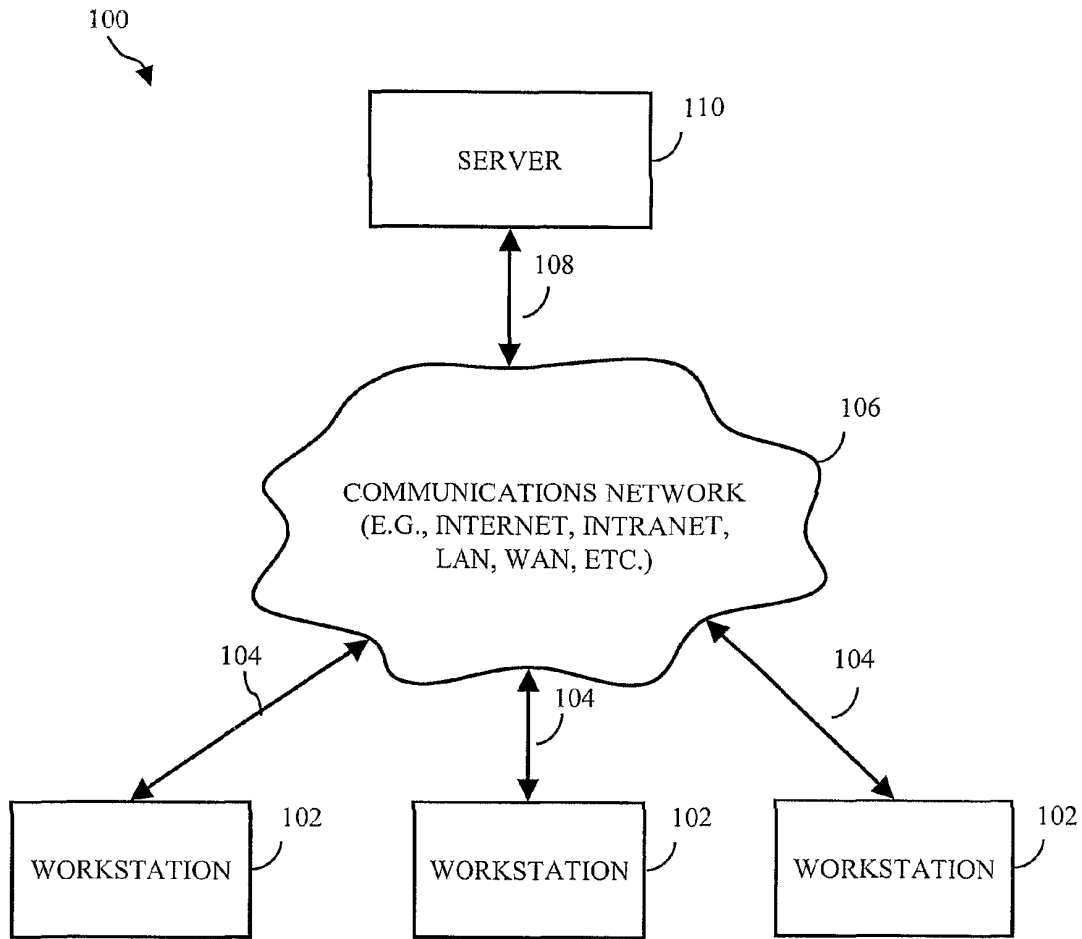


FIG. 1

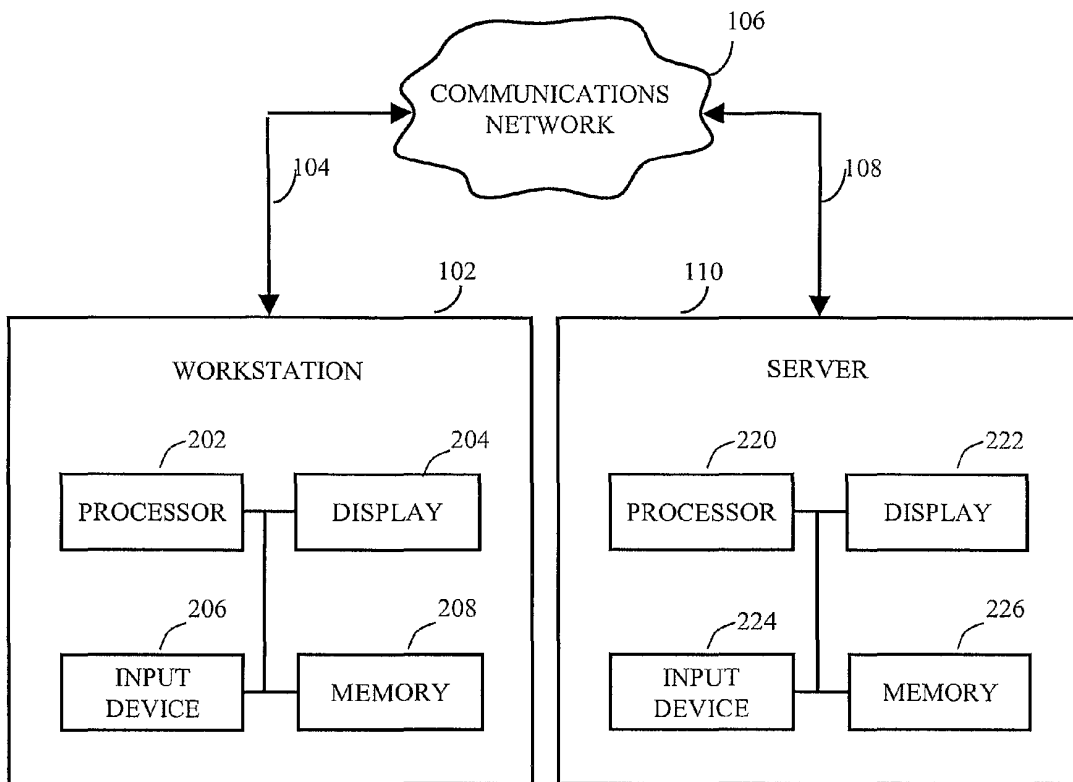


FIG. 2

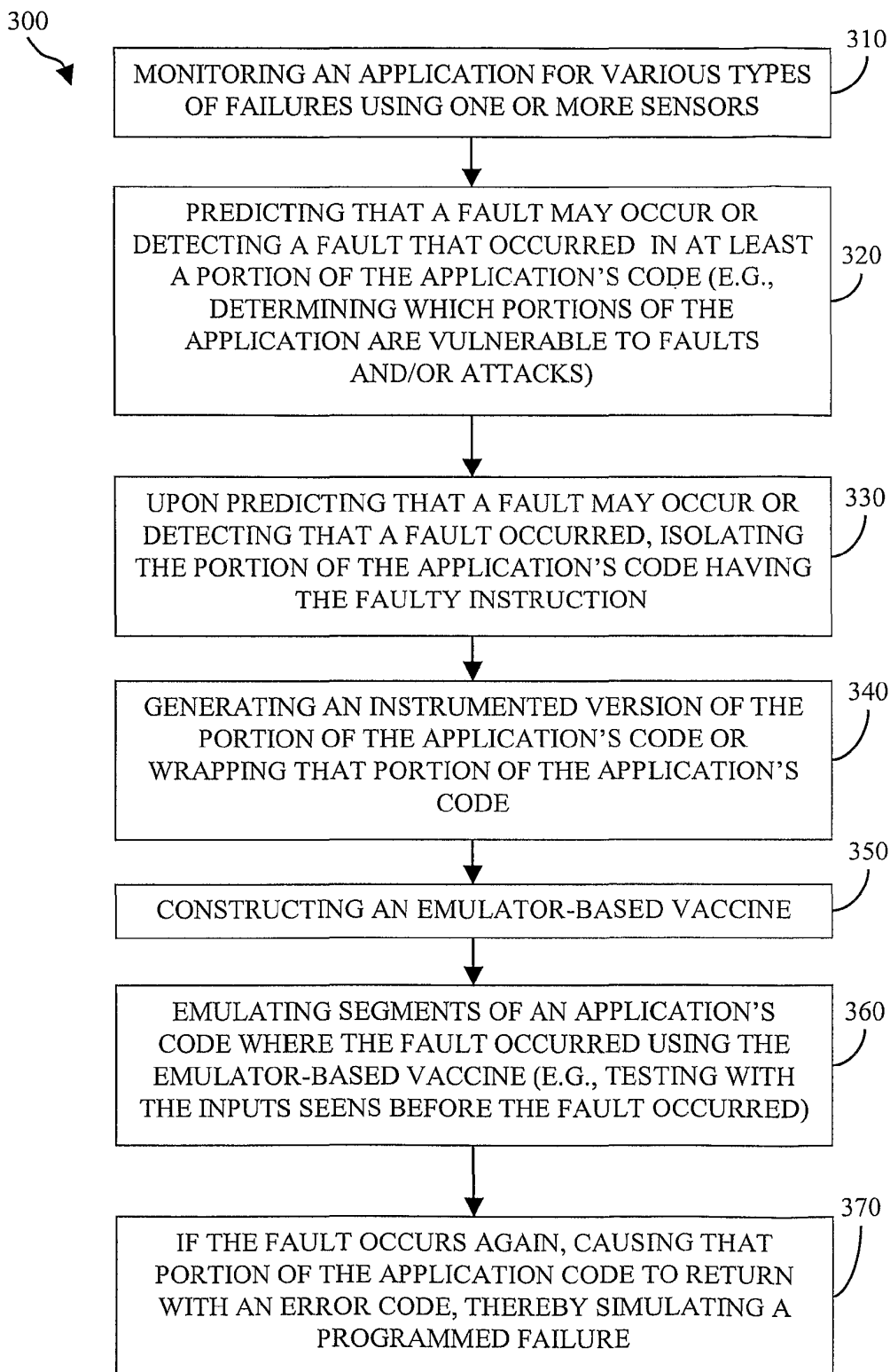


FIG. 3

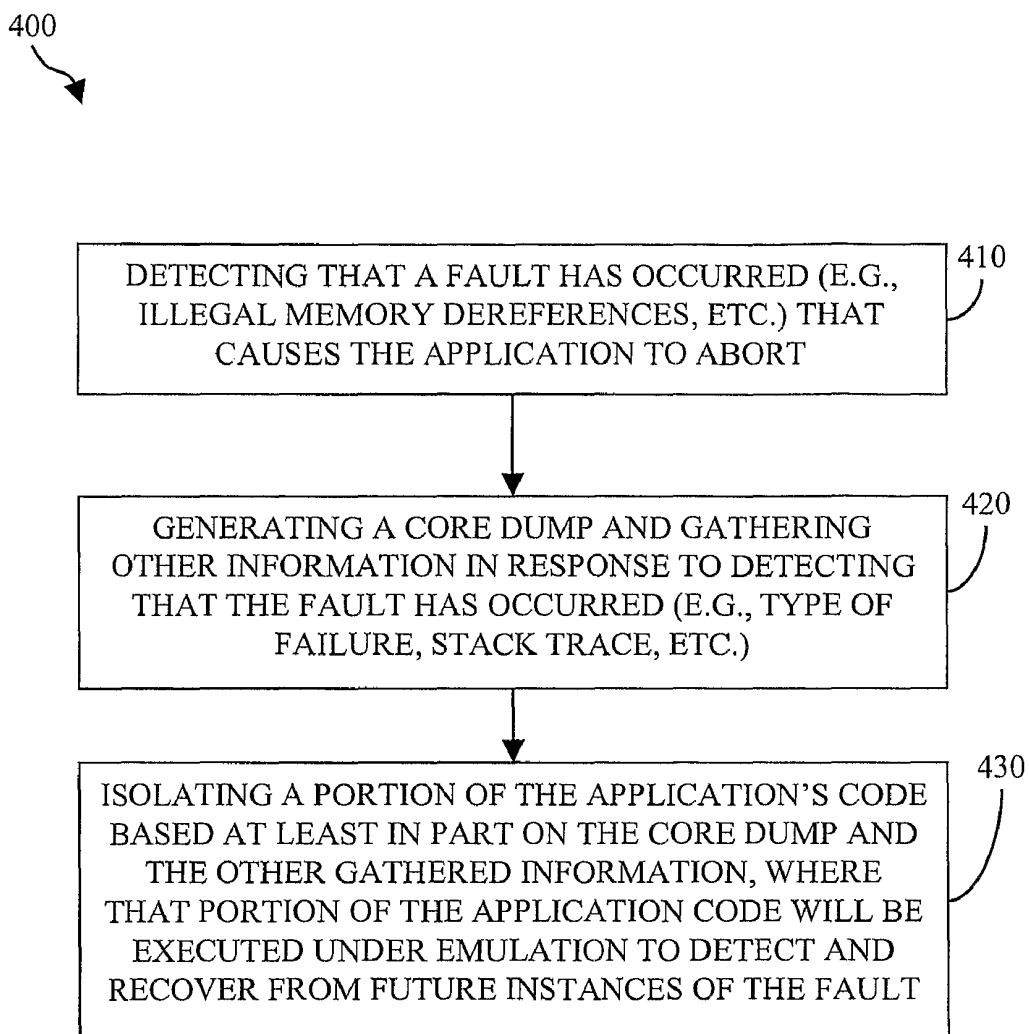


FIG. 4

500



```
void foo () {  
    int a = 1;  
    emulate_init();  
    emulate_begin(p_args);  
    a++;  
    emulate_end();  
    emulate_term();  
    printf("a = %d\n", a);  
}
```

FIG. 5

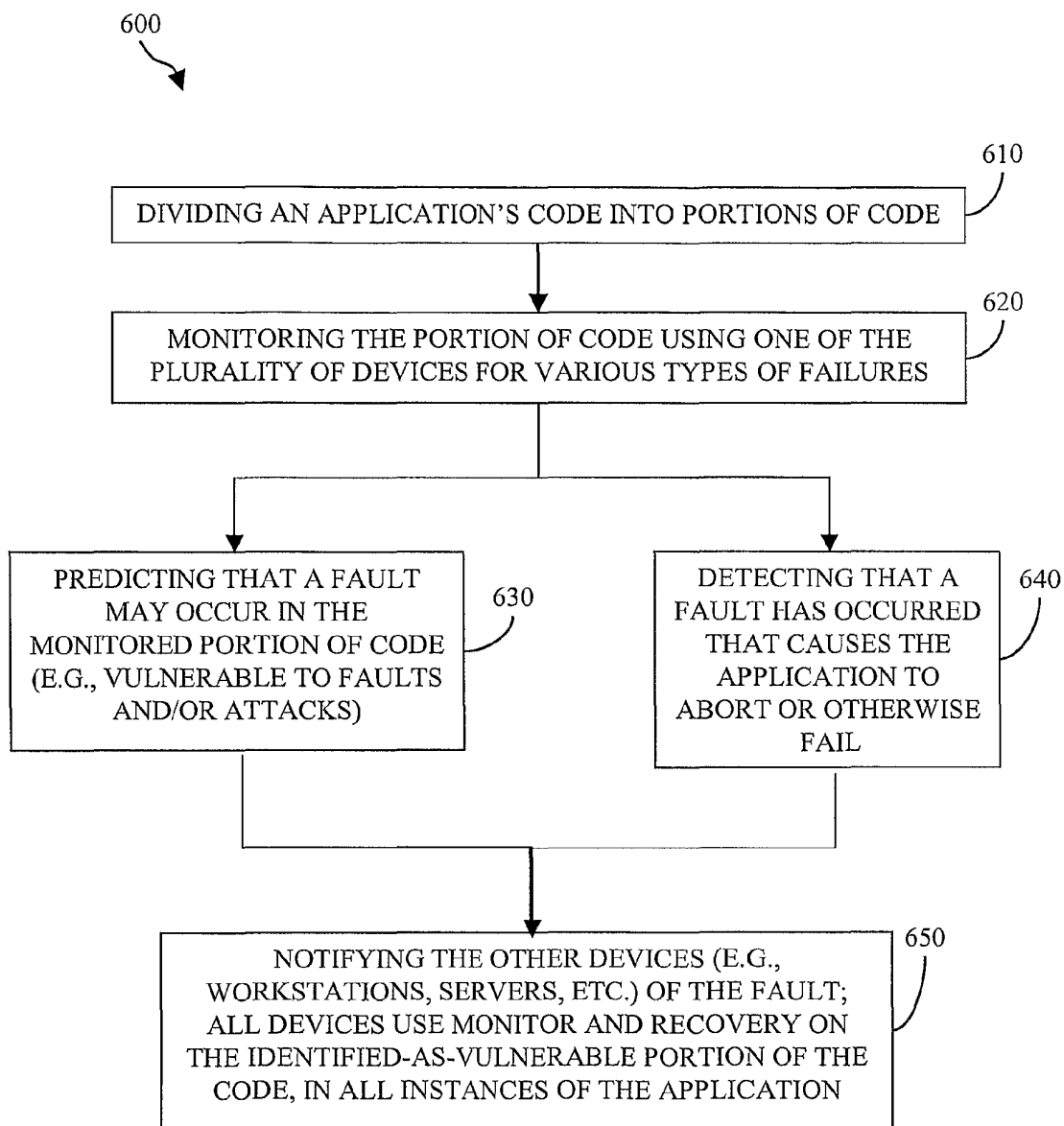


FIG. 6

700


f_i	x_i	r_i	v_i	T	$C(f_i x_i)$	$r_i * v_i$
a()	100	1	α_1	600	16	α_1
b()	200	2	α_2	600	33	$2\alpha_2$
c()	300	3	α_3	600	50	$3\alpha_3$

FIG. 7

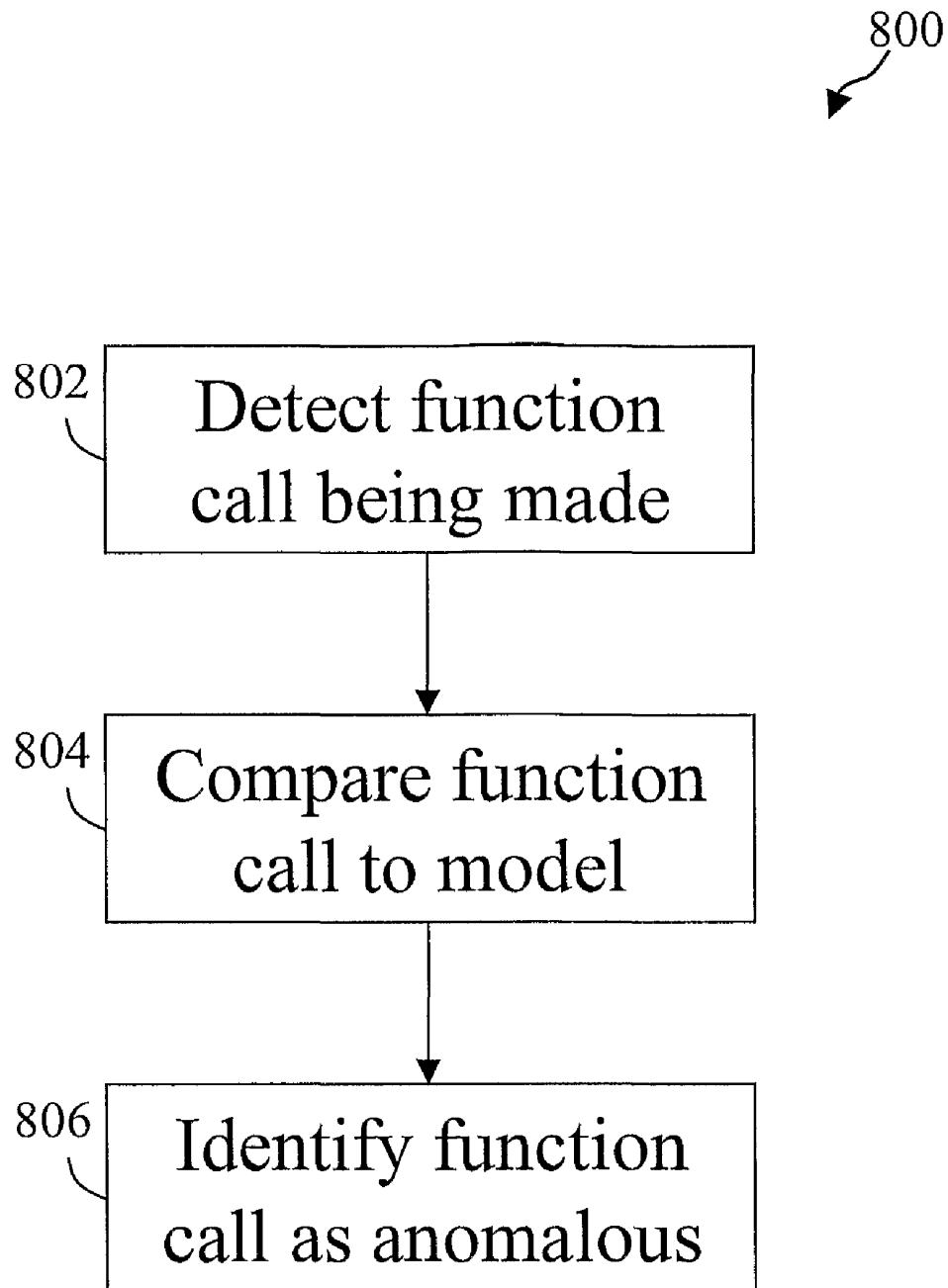


FIG. 8

US 8,074,115 B2

1

**METHODS, MEDIA AND SYSTEMS FOR
DETECTING ANOMALOUS PROGRAM
EXECUTIONS**

CROSS REFERENCE TO RELATED
APPLICATION

This application is a U.S. National Phase Application Under 35 U.S.C. §371 of International Patent Application No. PCT/US2006/041591, filed Oct. 25, 2006, which claims the benefit under 35 U.S.C. §119(e) of United States Provisional Patent Application No. 60/730,289, filed Oct. 25, 2005, each of which is hereby incorporated by reference herein in its entirety.

TECHNOLOGY AREA

The disclosed subject matter relates to methods, media, and systems for detecting anomalous program executions.

BACKGROUND

Applications may terminate due to any number of threats, program errors, software faults, attacks, or any other suitable software failure. Computer viruses, worms, trojans, hackers, key recovery attacks, malicious executables, probes, etc. are a constant menace to users of computers connected to public computer networks (such as the Internet) and/or private networks (such as corporate computer networks). In response to these threats, many computers are protected by antivirus software and firewalls. However, these preventative measures are not always adequate. For example, many services must maintain a high availability when faced by remote attacks, high-volume events (such as fast-spreading worms like Slammer and Blaster), or simple application-level denial of service (DoS) attacks.

Aside from these threats, applications generally contain errors during operation, which typically result from programmer error. Regardless of whether an application is attacked by one of the above-mentioned threats or contains errors during operation, these software faults and failures result in illegal memory access errors, division by zero errors, buffer overflows attacks, etc. These errors cause an application to terminate its execution or "crash."

SUMMARY

Methods, media, and systems for detecting anomalous program executions are provided. In some embodiments, methods for detecting anomalous program executions are provided, comprising: executing at least a part of a program in an emulator; comparing a function call made in the emulator to a model of function calls for the at least a part of the program; and identifying the function call as anomalous based on the comparison.

In some embodiments, computer-readable media containing computer-executable instructions that, when executed by a processor, cause the processor to perform a method for detecting anomalous program executions are provided, the method comprising: executing at least a part of a program in an emulator; comparing a function call made in the emulator to a model of function calls for the at least a part of the program; and identifying the function call as anomalous based on the comparison.

In some embodiments, systems for detecting anomalous program executions are provided, comprising: a digital processing device that: executes at least a part of a program in an

2

emulator; compares a function call made in the emulator to a model of function calls for the at least a part of the program; and identifies the function call as anomalous based on the comparison.

5 In some embodiments, methods for detecting anomalous program executions are provided, comprising: modifying a program to include indicators of program-level function calls being made during execution of the program; comparing at least one of the indicators of program-level function calls made in the emulator to a model of function calls for the at least a part of the program; and identifying a function call corresponding to the at least one of the indicators as anomalous based on the comparison.

10 In some embodiments, computer-readable media containing computer-executable instructions that, when executed by a processor, cause the processor to perform a method for detecting anomalous program executions are provided, the method comprising: modifying a program to include indicators of program-level function calls being made during execution of the program; comparing at least one of the indicators of program-level function calls made in the emulator to a model of function calls for the at least a part of the program; and identifying a function call corresponding to the at least one of the indicators as anomalous based on the comparison.

15 In some embodiments, systems for detecting anomalous program executions are provided, comprising: a digital processing device that: modifies a program to include indicators of program-level function calls being made during execution of the program; compares at least one of the indicators of program-level function calls made in the emulator to a model of function calls for the at least a part of the program; and identifies a function call corresponding to the at least one of the indicators as anomalous based on the comparison.

BRIEF DESCRIPTION OF THE DRAWINGS

The Detailed Description, including the description of various embodiments of the disclosed subject matter, will be best understood when read in reference to the accompanying figures wherein:

FIG. 1 is a schematic diagram of an illustrative system suitable for implementation of an application that monitors other applications and protects these applications against faults in accordance with some embodiments;

FIG. 2 is a detailed example of the server and one of the workstations of FIG. 1 that may be used in accordance with some embodiments;

FIG. 3 shows a simplified diagram illustrating repairing faults in an application and updating the application in accordance with some embodiments;

FIG. 4 shows a simplified diagram illustrating detecting and repairing an application in response to a fault occurring in accordance with some embodiments;

FIG. 5 shows an illustrative example of emulated code integrated into the code of an existing application in accordance with some embodiments;

FIG. 6 shows a simplified diagram illustrating detecting and repairing an application using an application community in accordance with some embodiments of the disclosed subject matter;

FIG. 7 shows an illustrative example of a table that may be calculated by a member of the application community for distributed bidding in accordance with some embodiments of the disclosed subject matter; and

US 8,074,115 B2

3

FIG. 8 shows a simplified diagram illustrating shows identifying a function call as being anomalous in accordance with some embodiments.

DETAILED DESCRIPTION

Methods, media, and systems for detecting anomalous program executions are provided. In some embodiments, systems and methods are provided that model application level computations and running programs, and that detect anomalous executions by, for example, instrumenting, monitoring and analyzing application-level program function calls and/or arguments. Such an approach can be used to detect anomalous program executions that may be indicative of a malicious attack or program fault.

The anomaly detection algorithm being used may be, for example, a probabilistic anomaly detection (PAD) algorithm or a one class support vector machine (OCSVM), which are described below, or any other suitable algorithm.

Anomaly detection may be applied to process execution anomaly detection, file system access anomaly detection, and/or network packet header anomaly detection. Moreover, as described herein, according to various embodiments, an anomaly detector may be applied to program execution state information. For example, as explained in greater detail below, an anomaly detector may model information on the program stack to detect anomalous program behavior.

In various embodiments, using PAD to model program stack information, such stack information may be extracted using, for example, Selective Transactional EMulation (STEM), which is described below and which permits the selective execution of certain parts, or all, of a program inside an instruction-level emulator, using the Valgrind emulator, by modifying a program's binary or source code to include indicators of what functions calls are being made (and any other suitable related information), or using any other suitable technique. In this manner, it is possible to determine dynamically (and transparently to the monitored program) the necessary information such as stack frames, function-call arguments, etc. For example, one or more of the following may be extracted from the program stack specific information: function name, the argument buffer name it may reference, and other features associated with the data sent to or returned from the called function (e.g., the length in bytes of the data, or the memory location of the data).

For example, as illustrated in FIG. 8, an anomaly detector may be applied, for example, by extracting data pushed onto the stack (e.g., by using an emulator or by modifying a program), and creating a data record provided to the anomaly detector for processing at 802. According to various embodiments, in a first phase, an anomaly detector models normal program execution stack behavior. In the detection mode, after a model has been computed, the anomaly detector can detect stacked function references as anomalous at 806 by comparing those references to the model based on the training data at 804.

Once an anomaly is detected, according to various embodiments, selective transactional emulation (STEM) and error virtualization may be used to reverse (undo) the effects of processing the malicious input (e.g., changes to program variables or the file system) in order to allow the program to recover execution in a graceful manner. In this manner, the precise location of the failed (or attacked) program at which an anomaly was found may be identified. Also, the application of an anomaly detector to function calls can enable rapid detection of malicious program executions, such that it is possible to mitigate against such faults or attacks (e.g., by

4

using patch generation systems, or content filtering signature generation systems). Moreover, given precise identification of a vulnerable location, the performance impact may be reduced by using STEM for parts or all of a program's execution.

As explained above, anomaly detection can involve the use of detection models. These models can be used in connection with automatic and unsupervised learning.

A probabilistic anomaly detection (PAD) algorithm can be used to train a model for detecting anomalies. This model may be, in essence, a density estimation, where the estimation of a density function $p(x)$ over normal data allows the definition of anomalies as data elements that occur with low probability. The detection of low probability data (or events) are represented as consistency checks over the normal data, where a record is labeled anomalous if it fails any one of these tests.

First and second order consistency checks can be applied. First order consistency checks verify that a value is consistent with observed values of that feature in the normal data set. These first order checks compute the likelihood of an observation of a given feature, $P(X_i)$, where X_i are the feature variables. Second order consistency checks determine the conditional probability of a feature value given another feature value, denoted by $P(X_i|X_j)$, where X_i and X_j are the feature variables.

One way to compute these probabilities is to estimate a multinomial that computes the ratio of the counts of a given element to the total counts. However, this results in a biased estimator when there is a sparse data set. Another approach is to use an estimator to determine these probability distributions. For example, let N be the total number of observations, N_i be the number of observations of symbol i , α be the "pseudo count" that is added to the count of each observed symbol, k^0 be the number of observed symbols, and L be the total number of possible symbols. Using these definitions, the probability for an observed element i can be given by:

$$P(X = i) = \frac{N_i + \alpha}{k^0 \alpha + N} C \quad (1)$$

and the probability for an unobserved element i can be:

$$P(X = i) = \frac{1}{L - k^0} (1 - C) \quad (2)$$

where C , the scaling factor, accounts for the likelihood of observing a previously observed element versus an unobserved element. C can be computed as:

$$C = \left(\sum_{k=k^0}^L \frac{k^0 \alpha + N}{k \alpha + N} m_k \right) \left(\sum_{k \geq k^0} m_k \right)^{-1} \quad (3)$$

where

$$m_k = P(S = k) \frac{k!}{k = k^0} \frac{\Gamma(k \alpha)}{\Gamma(k \alpha + N)} \Big|$$

and $P(S=k)$ is a prior probability associated with the size of the subset of elements in the alphabet that have non-zero probability.

Because this computation of C can be time consuming, C can also be calculated by:

$$C = \frac{N}{N + L - k^0} \tag{4}$$

The consistency check can be normalized to account for the number of possible outcomes of L by $\log(P/(1/L)) = \log(P) + \log(L)$.

Another approach that may be used instead of using PAD for model generation and anomaly detection is a one class SVM (OCSVM) algorithm. The OCSVM algorithm can be used to map input data into a high dimensional feature space (via a kernel) and iteratively find the maximal margin hyperplane which best separates the training data from the origin. The OCSVM may be viewed as a regular two-class SVM where all the training data lies in the first class, and the origin is taken as the only member of the second class. Thus, the hyperplane (or linear decision boundary) can correspond to the classification rule:

$$f(x) = \langle w, x \rangle + b \tag{5}$$

where w is the normal vector and b is a bias term. The OCSVM can be used to solve an optimization problem to find the rule f with maximal geometric margin. This classification rule can be used to assign a label to a test example x. If $f(x) < 0$, x can be labeled as an anomaly, otherwise it can be labeled as normal. In practice, there is a trade-off between maximizing the distance of the hyperplane from the origin and the number of training data points contained in the region separated from the origin by the hyperplane.

Solving the OCSVM optimization problem can be equivalent to solving the dual quadratic programming problem:

$$\min_{\alpha} \frac{1}{2} \sum_{ij} \alpha_i \alpha_j K(x_i, x_j) \tag{6}$$

subject to the constraints

$$0 \leq \alpha_i \leq \frac{1}{v_l} \tag{7}$$

and

$$\sum_i \alpha_i = 1 \tag{8}$$

where α_i is a lagrange multiplier (or “weight” on example i such that vectors associated with non-zero weights are called “support vectors” and solely determine the optimal hyperplane), v is a parameter that controls the trade-off between maximizing the distance of the hyperplane from the origin and the number of data points contained by the hyperplane, l is the number of points in the training dataset, and $K(x_i, x_j)$ is the kernel function. By using the kernel function to project input vectors into a feature space, nonlinear decision boundaries can be allowed for. Given a feature map:

$$\phi: X \rightarrow \mathbb{R}^N \tag{9}$$

where Φ maps training vectors from input space X to a high-dimensional feature space, the kernel function can be defined as:

$$K(x, y) = \langle \phi(x), \phi(y) \rangle \tag{10}$$

Feature vectors need not be computed explicitly, and computational efficiency can be improved by directly computing kernel values $K(x, y)$. Three common kernels can be used:

Linear kernel: $K(x, y) = \langle x, y \rangle$

Polynomial kernel: $K(x, y) = \langle x, y + 1 \rangle^d$, where d is the degree of the polynomial

Gaussian kernel: $K(x, y) = e^{-\|x - y\|^2 / (2\sigma^2)}$, where σ^2 is the variance

5 Kernels from binary feature vectors can be obtained by mapping a record into a feature space such that there is one dimension for every unique entry for each record value. A particular record can have the value 1 in the dimensions which correspond to each of its specific record entries, and the value 0 for every other dimension in feature space. Linear kernels, second order polynomial kernels, and gaussian kernels can be calculated using these feature vectors for each record. Kernels can also be calculated from frequency-based feature vectors such that, for any given record, each feature corresponds to the number of occurrences of the corresponding record component in the training set. For example, if the second component of a record occurs three times in the training set, the second feature value for that record is three. These frequency-based feature vectors can be used to compute linear and polynomial kernels.

According to various embodiments, “mimicry attacks” which might otherwise thwart OS system call level anomaly detectors by using normal appearing sequences of system calls can be detected. For example, mimicry attacks are less likely to be detected when the system calls are only modeled as tokens from an alphabet, without any information about arguments. Therefore, according to various embodiments, the models used are enriched with information about the arguments (data) such that it may be easier to detect mimicry attacks.

According to various embodiments, models are shared among many members of a community running the same application (referred to as an “application community”). In particular, some embodiments can share models with each other and/or update each other’s models such that the learning of anomaly detection models is relatively quick. For example, instead of running a particular application for days at a single site, according to various embodiments, thousands of replicated applications can be run for a short period of time (e.g., one hour), and the models created based on the distributed data can be shared. While only a portion of each application instance may be monitored, for example, the entire software body can be monitored across the entire community. This can enable the rapid acquisition of statistics, and relatively fast learning of an application profile by sharing, for example, aggregate information (rather than the actual raw data used to construct the model).

Model sharing can result in one standard model that an attacker could potentially access and use to craft a mimicry attack. Therefore, according to various embodiments, unique and diversified models can be created. For example, such unique and diversified models can be created by randomly choosing particular features from the application execution that is modeled, such that the various application instances compute distinct models. In this manner, attacks may need to avoid detection by multiple models, rather than just a single model. Creating unique and diversified models not only has the advantage of being more resistant to mimicry attacks, but also may be more efficient. For example, if only a portion of an application is modeled by each member of an application community, monitoring will generally be simpler (and cheaper) for each member of the community. In the event that one or more members of an application community are attacked, according to various embodiments, the attack (or fault) will be detected, and patches or a signature can be provided to those community members who are blind to the crafted attack (or fault).

US 8,074,115 B2

7

Random (distinct) model building and random probing may be controlled by a software registration key provided by a commercial off-the-shelf (COTS) software vendor or some other data providing “randomization.” For example, for each member of an application community, some particular randomly chosen function or functions and its associated data may be chosen for modeling, while others may simply be ignored. Moreover, because vendors can generate distinct keys and serial numbers when distributing their software, this feature can be used to create a distinct random subset of functions to be modeled. Also, according to various embodiments, even community members who model the same function or functions may exchange models.

According to various embodiments, when an application execution is being analyzed over many copies distributed among a number of application community members to profile the entire code of an application, it can be determined whether there are any segments of code that are either rarely or never executed, and a map can be provided of the code layout identifying “suspect code segments” for deeper analysis and perhaps deeper monitoring. Those segments identified as rarely or never executed may harbor vulnerabilities not yet executed or exploited. Such segments of code may have been designed to execute only for very special purposes such as error handling, or perhaps even for triggering malicious code embedded in the application. Since they are rarely or never executed, one may presume that such code segments have had less regression testing, and may have a higher likelihood of harboring faulty code.

Rarely or never executed code segments may be identified and may be monitored more thoroughly through, for example, emulation. This deep monitoring may have no discernible overhead since the code in question is rarely or never executed. But such monitoring performed in each community member may prevent future disasters by preventing such code (and its likely vulnerabilities) from being executed in a malicious/faulty manner. Identifying such code may be performed by a sensor that monitors loaded modules into the running application (e.g., DLL loads) as well as addresses (PC values) during code execution and creates a “frequency” map of ranges of the application code. For example, a set of such distributed sensors may communicate with each other (or through some site that correlates their collective information) to create a central, global MAP of the application execution profile. This profile may then be used to identify suspect code segments, and then subsequently, this information may be useful to assign different kinds of sensors/monitors to different code segments. For example, an interrupt service routine (ISR) may be applied to these suspect sections of code.

It is noted that a single application instance may have to be run many times (e.g., thousands of times) in order to compute an application profile or model. However, distributed sensors whose data is correlated among many (e.g., a thousand) application community members can be used to compute a substantially accurate code profile in a relatively short amount of time. This time may be viewed as a “training period” to create the code map.

According to various embodiments, models may be automatically updated as time progresses. For example, although a single site may learn a particular model over some period of time, application behavior may change over time. In this case, the previously learned model may no longer accurately reflect the application characteristics, resulting in, for example, the generation of an excessive amount of false alarms (and thus an increase in the false positive rate over time). A possible solution to this “concept drift” issue entails at least two possible approaches, both intended to update models over time. A

8

first approach to solving (or at least reducing the effects of) the “concept drift” issue involves the use of “incremental learning algorithms,” which are algorithms that piecemeal update their models with new data, and that may also “expire” parts of the computed model created by older data. This piecemeal incremental approach is intended to result in continuous updating using relatively small amounts of data seen by the learning system.

A second approach to solving (or at least reducing the effect of) the “concept drift” issue involves combining multiple models. For example, presuming that an older model has been computed from older data during some “training epoch,” a new model may be computed concurrently with a new epoch in which the old model is used to detect anomalous behavior. Once a new model is computed, the old model may be retired or expunged, and replaced by the new model. Alternatively, for example, multiple models such as described above may be combined. In this case, according to various embodiments, rather than expunging the old model, a newly created model can be algorithmically combined with the older model using any of a variety of suitable means. In the case of statistical models that are based upon frequency counts of individual data points, for example, an update may consist of an additive update of the frequency count table. For example, PAD may model data by computing the number of occurrences of a particular data item, “X.” Two independently learned PAD models can thus have two different counts for the same value, and a new frequency table can be readily computed by summing the two counts, essentially merging two tables and updating common values with the sum of their respective counts.

According to various embodiments, the concept of model updating that is readily achieved in the case of computed PAD models may be used in connection with model sharing. For example, rather than computing two models by the same device for a distinct application, two distinct models may be computed by two distinct instances of an application by two distinct devices, as described above. The sharing of models may thus be implemented by the model update process described herein. Hence, a device may continuously learn and update its models either by computing its own new model, or by downloading a model from another application community member (e.g., using the same means involved in the combining of models).

In the manners described above, an application community may be configured to continuously refresh and update all community members, thereby making mimicry attacks far more difficult to achieve.

As mentioned above, it is possible to mitigate against faults or attacks by using patch generation systems. In accordance with various embodiments, when patches are generated, validated, and deployed, the patches and/or the set of all such patches may serve the following.

First, according to various embodiments, each patch may be used as a “pattern” to be used in searching other code for other unknown vulnerabilities. An error (or design flaw) in programming that is made by a programmer and that creates a vulnerability may show up elsewhere in code. Therefore, once a vulnerability is detected, the system may use the detected vulnerability (and patch) to learn about other (e.g., similar) vulnerabilities, which may be patched in advance of those vulnerabilities being exploited. In this manner, over time, a system may automatically reduce (or eliminate) vulnerabilities.

Second, according to various embodiments, previously generated patches may serve as exemplars for generating new patches. For example, over time, a taxonomy of patches may

US 8,074,115 B2

9

be assembled that are related along various syntactic and semantic dimensions. In this case, the generation of new patches may be aided by prior examples of patch generation.

Additionally, according to various embodiments, generated patches may themselves have direct economic value. For example, once generated, patches may be “sold” back to the vendors of the software that has been patched.

As mentioned above, in order to alleviate monitoring costs, instead of running a particular application for days at a single site, many (e.g., thousands) replicated versions of the application may be run for a shorter period of time (e.g., an hour) to obtain the necessary models. In this case, only a portion of each replicated version of the application may be monitored, although the entire software body is monitored using the community of monitored software applications. Moreover, according to various embodiments, if a software module has been detected as faulty, and a patch has been generated to repair it, that portion of the software module, or the entire software module, may no longer need to be monitored. In this case, over time, patch generated systems may have fewer audit/monitoring points, and may thus improve in execution speed and performance. Therefore, according to various embodiments, software systems may be improved, where vulnerabilities are removed, and the need for monitoring is reduced (thereby reducing the costs and overheads involved with detecting faults).

It is noted that, although described immediately above with regard to an application community, the notion of automatically identifying faults of an application, improving the application over time by repairing the faults, and eliminating monitoring costs as repairs are deployed may also be applied to a single, standalone instance of an application (without requiring placements as part of a set of monitored application instances).

Selective transactional emulation (STEM) and error virtualization can be beneficial for reacting to detected failures/attacks in software. According to various embodiments, STEM and error virtualization can be used to provide enhanced detection of some types of attacks, and enhanced reaction mechanisms to some types of attacks/failures.

A learning technique can be applied over multiple executions of a piece of code (e.g., a function or collection of functions) that may previously have been associated with a failure, or that is being proactively monitored. By retaining knowledge on program behavior across multiple executions, certain invariants (or probable invariants) may be learned, whose violation in future executions indicates an attack or imminent software fault.

In the case of control hijacking attacks, certain control data that resides in memory is overwritten through some mechanism by an attacker. That control data is then used by the program for an internal operation, allowing the attacker to subvert the program. Various forms of buffer overflow attacks (stack and heap smashing, jump into libc, etc.) operate in this fashion. Such attacks can be detected when the corrupted control data is about to be used by the program (i.e., after the attack has succeeded). In various embodiments, such control data (e.g., memory locations or registers that hold such data) that is about to be overwritten with “tainted” data, or data provided by the network (which is potentially malicious) can be detected.

In accordance with various embodiments, how data modifications propagate throughout program execution can be monitored by maintaining a memory bit for every byte or word in memory. This bit is set for a memory location when a machine instruction uses as input data that was provided as input to the program (e.g., was received over the network, and

10

is thus possibly malicious) and produces output that is stored in this memory location. If a control instruction (such as a JUMP or CALL) uses as an argument a value in a memory location in which the bit is set (i.e., the memory location is “tainted”), the program or the supervisory code that monitors program behavior can recognize an anomaly and raises an exception.

Detecting corruption before it happens, rather than later (when the corrupted data is about to be used by a control instruction), makes it possible to stop an operation and to discard its results/output, without other collateral damage. Furthermore, in addition to simply retaining knowledge of what is control and what is non-control data, according to various embodiments, knowledge of which instructions in the monitored piece of code typically modify specific memory locations can also be retained. Therefore, it is possible to detect attacks that compromise data that are used by the program computation itself, and not just for the program control flow management.

According to various embodiments, the inputs to the instruction(s) that can fail (or that can be exploited in an attack) and the outputs (results) of such instructions can be correlated with the inputs to the program at large. Inputs to an instruction are registers or locations in memory that contain values that may have been derived (in full or partially) by the input to the program. By computing a probability distribution model on the program input, alternate inputs may be chosen to give to the instruction or the function (“input rewriting” or “input modification”) when an imminent failure is detected, thereby allowing the program to “sidestep” the failure. However, because doing so may still cause the program to fail, according to various embodiments, micro-speculation (e.g., as implemented by STEM) can optionally be used to verify the effect of taking this course of action. A recovery technique (with different input values or error virtualization, for example) can then be used. Alternatively, for example, the output of the instruction may be caused to be a value/result that is typically seen when executing the program (“output overloading”).

In both cases (input modification or output overloading), the values to use may be selected based on several different criteria, including but not limited to one or more of the following: the similarity of the program input that caused failure to other inputs that have not caused a failure; the most frequently seen input or output value for that instruction, based on contextual information (e.g., when particular sequence of functions are in the program call stack); and most frequently seen input or output value for that instruction across all executions of the instruction (in all contexts seen). For example, if a particular DIVIDE instruction is detected in a function that uses a denominator value of zero, which would cause a process exception, and subsequently program failure, the DIVIDE instruction can be executed with a different denominator (e.g., based on how similar the program input is to other program inputs seen in the past, and the denominator values that these executions used). Alternatively, the DIVIDE instruction may be treated as though it had given a particular division result. The program may then be allowed to continue executing, while its behavior is being monitored. Should a failure subsequently occur while still under monitoring, a different input or output value for the instruction can be used, for example, or a different repair technique can be used. According to various embodiments, if none of the above strategies is successful, the user or administrator may be notified, program execution may be terminated, a rollback to

US 8,074,115 B2

11

a known good state (ignoring the current program execution) may take place, and/or some other corrective action may be taken.

According to various embodiments, the techniques used to learn typical data can be implemented as designer choice. For example, if it is assumed that the data modeled is 32-bit words, a probability distribution of this range of values can be estimated by sampling from multiple executions of the program. Alternatively, various cluster-based analyses may partition the space of typical data into clusters that represent groups of similar/related data by some criteria. Vector Quantization techniques representing common and similar data based on some "similarity" measure or criteria may also be compiled and used to guide modeling.

FIG. 1 is a schematic diagram of an illustrative system 100 suitable for implementation of various embodiments. As illustrated in FIG. 1, system 100 may include one or more workstations 102. Workstations 102 can be local to each other or remote from each other, and can be connected by one or more communications links 104 to a communications network 106 that is linked via a communications link 108 to a server 110.

In system 100, server 110 may be any suitable server for executing the application, such as a processor, a computer, a data processing device, or a combination of such devices. Communications network 106 may be any suitable computer network including the Internet, an intranet, a wide-area network (WAN), a local-area network (LAN), a wireless network, a digital subscriber line (DSL) network, a frame relay network, an asynchronous transfer mode (ATM) network, a virtual private network (VPN), or any combination of any of the same. Communications links 104 and 108 may be any communications links suitable for communicating data between workstations 102 and server 110, such as network links, dial-up links, wireless links, hard-wired links, etc. Workstations 102 may be personal computers, laptop computers, mainframe computers, data displays, Internet browsers, personal digital assistants (PDAs), two-way pagers, wireless terminals, portable telephones, etc., or any combination of the same. Workstations 102 and server 110 may be located at any suitable location. In one embodiment, workstations 102 and server 110 may be located within an organization. Alternatively, workstations 102 and server 110 may be distributed between multiple organizations.

The server and one of the workstations, which are depicted in FIG. 1, are illustrated in more detail in FIG. 2. Referring to FIG. 2, workstation 102 may include digital processing device (such as a processor) 202, display 204, input device 206, and memory 208, which may be interconnected. In a preferred embodiment, memory 208 contains a storage device for storing a workstation program for controlling processor 202. Memory 208 may also contain an application for detecting and repairing application from faults according to various embodiments. In some embodiments, the application may be resident in the memory of workstation 102 or server 110.

Processor 202 may use the workstation program to present on display 204 the application and the data received through communication link 104 and commands and values transmitted by a user of workstation 102. It should also be noted that data received through communication link 104 or any other communications links may be received from any suitable source, such as web services. Input device 206 may be a computer keyboard, a cursor-controller, a dial, a switchbank, lever, or any other suitable input device as would be used by a designer of input systems or process control systems.

12

Server 110 may include processor 220, display 222, input device 224, and memory 226, which may be interconnected. In some embodiments, memory 226 contains a storage device for storing data received through communication link 108 or through other links, and also receives commands and values transmitted by one or more users. The storage device can further contain a server program for controlling processor 220.

In accordance with some embodiments, a self-healing system that allows an application to automatically recover from software failures and attacks is provided. By selectively emulating at least a portion or all of the application's code when the system detects that a fault has occurred, the system surrounds the detected fault to validate the operands to machine instructions, as appropriate for the type of fault. The system emulates that portion of the application's code with a fix and updates the application. This increases service availability in the presence of general software bugs, software failures, attacks.

Turning to FIGS. 3 and 4, simplified flowcharts illustrating various steps performed in detecting faults in an application and fixing the application in accordance with some embodiments are provided. These are generalized flow charts. It will be understood that the steps shown in FIGS. 3 and 4 may be performed in any suitable order, some may be deleted, and others added.

Generally, process 300 begins by detecting various types of failures in one or more applications at 310. In some embodiments, detecting for failures may include monitoring the one or more applications for failures, e.g., by using an anomaly detector as described herein. In some embodiments, the monitoring or detecting of failures may be performed using one or more sensors at 310. Failures include programming errors, exceptions, software faults (e.g., illegal memory accesses, division by zero, buffer overflow attacks, time-of-check-to-time-of-use (TOCTTOU) violations, etc.), threats (e.g., computer viruses, worms, trojans, hackers, key recovery attacks, malicious executables, probes, etc.), and any other suitable fault that may cause abnormal application termination or adversely affect the one or more applications.

Any suitable sensors may be used to detect failures or monitor the one or more applications. For example, in some embodiments, anomaly detectors as described herein can be used.

At 320, feedback from the sensors may be used to predict which parts of a given application's code may be vulnerable to a particular class of attack (e.g., remotely exploitable buffer overflows). In some embodiments, the sensors may also detect that a fault has occurred. Upon predicting that a fault may occur or detecting that a fault has occurred, the portion of the application's code having the faulty instruction or vulnerable function can be isolated, thereby localizing predicted faults at 330.

Alternatively, as shown and discussed in FIG. 4, the one or more sensor may monitor the application until it is caused to abnormally terminate. The system may detect that a fault has occurred, thereby causing the actual application to terminate. As shown in FIG. 4, at 410, the system forces a misbehaving application to abort. In response to the application terminating, the system generates a core dump file or produces other failure-related information, at 420. The core dump file may include, for example, the type of failure and the stack trace when that failure occurred. Based at least in part on the core dump file, the system isolates the portion of the application's code that contains the faulty instruction at 430. Using the core dump file, the system may apply selective emulation to the

US 8,074,115 B2

13

isolated portion or slice of the application. For example, the system may start with the top-most function in the stack trace.

Referring back to FIG. 3, in some embodiments, the system may generate an instrumented version of the application (340). For example, an instrumented version of the application may be a copy of a portion of the application's code or all of the application's code. The system may observe instrumented portions of the application. These portions of the application may be selected based on vulnerability to a particular class of attack. The instrumented application may be executed on the server that is currently running the one or more applications, a separate server, a workstation, or any other suitable device.

Isolating a portion of the application's code and using the emulator on the portion allows the system to reduce and/or minimize the performance impact on the immunized application. However, while this embodiment isolates a portion or a slice of the application's code, the entire application may also be emulated. The emulator may be implemented completely in software, or may take advantage of hardware features of the system processor or architecture, or other facilities offered by the operating system to otherwise reduce and/or minimize the performance impact of monitoring and emulation, and to improve accuracy and effectiveness in handling failures.

An attempt to exploit such a vulnerability exposes the attack or input vector and other related information (e.g., attacked buffer, vulnerable function, stack trace, etc.). The attack or input vector and other related information can then be used to construct an emulator-based vaccine or a fix that implements array bounds checking at the machine-instruction level at 350, or other fixes as appropriate for the detected type of failure. The vaccine can then be tested in the instrumented application using an instruction-level emulator (e.g., libtasvm x86 emulator, STEM x86 emulator, etc.) to determine whether the fault was fixed and whether any other functionality (e.g., critical functionality) has been impacted by the fix.

By continuously testing various vaccines using the instruction-level emulator, the system can verify whether the specific fault has been repaired by running the instrumented application against the event sequence (e.g., input vectors) that caused the specific fault. For example, to verify the effectiveness of a fix, the application may be restarted in a test environment or a sandbox with the instrumentation enabled, and is supplied with the one or more input vectors that caused the failure. A sandbox generally creates an environment in which there are strict limitations on which system resources the instrumented application or a function of the application may request or access.

At 360, the instruction-level emulator can be selectively invoked for segments of the application's code, thereby allowing the system to mix emulated and non-emulated code within the same code execution. The emulator may be used to, for example, detect and/or monitor for a specific type of failure prior to executing the instruction, record memory modifications during the execution of the instruction (e.g., global variables, library-internal state, libc standard I/O structures, etc.) and the original values, revert the memory stack to its original state, and simulate an error return from a function of the application. That is, upon entering the vulnerable section of the application's code, the instruction-level emulator can capture and store the program state and processes all instructions, including function calls, inside the area designated for emulation. When the program counter references the first instruction outside the bounds of emulation, the virtual processor copies its internal state back to the

14

device processor registers. While registers are updated, memory updates are also applied through the execution of the emulation. The program, unaware of the instructions executed by the virtual processor, continues normal execution on the actual processor.

In some embodiments, the instruction-level emulator may be linked with the application in advance. Alternatively, in response to a detected failure, the instruction-level emulator may be compiled in the code. In another suitable embodiment, the instruction-level emulator may be invoked in a manner similar to a modern debugger when a particular program instruction is executed. This can take advantage of breakpoint registers and/or other program debugging facilities that the system processor and architecture possess, or it can be a pure-software approach.

The use of an emulator allows the system to detect and/or monitor a wide array of software failures, such as illegal memory dereferences, buffer overflows, and buffer underflows, and more generic faults, such as divisions by zero. The emulator checks the operands of the instructions it is about to emulate using, at least partially, the vector and related information provided by the one or more sensors that detected the fault. For example, in the case of a division by zero, the emulator checks the value of the operand to the div instruction. In another example, in the case of illegal memory dereferencing, the emulator verifies whether the source and destination address of any memory access (or the program counter for instruction fetches) points to a page that is mapped to the process address space using the mincore() system call, or the appropriate facilities provided by the operating system. In yet another example, in the case of buffer overflow detection, the memory surrounding the vulnerable buffer, as identified by the one or more sensors, is padded by one byte. The emulator then watches for memory writes to these memory locations. This may require source code availability so as to insert particular variables (e.g., canary variables that launch themselves periodically and perform some typical user transaction to enable transaction-latency evaluation around the clock). The emulator can thus prevent the overflow before it overwrites the remaining locations in the memory stack and recovers the execution. Other approaches for detecting these failures may be incorporated in the system in a modular way, without impacting the high-level operation and characteristics of the system.

For example, the instruction-level emulator may be implemented as a statically-linked C library that defines special tags (e.g., a combination of macros and function calls) that mark the beginning and the end of selective emulation. An example of the tags that are placed around a segment of the application's code for emulation by the instruction-level emulator is shown in FIG. 5. As shown in FIG. 5, the C macro emulate_init() moves the program state (general, segment, eflags, and FPU registers) into an emulator-accessible global data structure to capture state immediately before the emulator takes control. The data structure can be used to initialize the virtual registers. emulate_begin() obtains the memory location of the first instruction following the call to itself. The instruction address may be the same as the return address and can be found in the activation record of emulate_begin(), four bytes above its base stack pointer. The fetch/decode/execute/retire cycle of instructions can continue until either emulate_end() is reached or when the emulator detects that control is returning to the parent function. If the emulator does not encounter an error during its execution, the emulator's instruction pointer references the emulate_term() macro at completion. To enable the instrumented application to continue execution at this address, the return address of the emu-

US 8,074,115 B2

15

late_begin() activation record can be replaced with the current value of the instruction pointer. By executing emulate_term(), the emulator's environment can be copied to the program registers and execution continues under normal conditions.

Although the emulator can be linked with the vulnerable application when the source code of the vulnerable application is available, in some embodiments the processor's programmable breakpoint register can be used to invoke the emulator without the running process even being able to detect that it is now running under an emulator.

In addition to monitoring for failures prior to executing instructions and reverting memory changes made by a particular function when a failure occurs (e.g., by having the emulator store memory modifications made during its execution), the emulator can also simulate an error return from the function. For example, some embodiments may generate a map between a set of errors that may occur during an application's execution and a limited set of errors that are explicitly handled by the application's code (sometimes referred to herein as "error virtualization"). As described below, the error virtualization features may be based on heuristics. However, any suitable approach for determining the return values for a function may be used. For example, aggressive source code analysis techniques to determine the return values that are appropriate for a function may be used. In another example, portions of code of specific functions can be marked as fail-safe and a specific value may be returned when an error return is forced (e.g., for code that checks user permissions). In yet another example, the error value returned for a function that has failed can be determined using information provided by a programmer, system administrator, or any other suitable user.

These error virtualization features allow an application to continue execution even though a boundary condition that was not originally predicted by a programmer allowed a fault to occur. In particular, error virtualization features allows for the application's code to be retrofitted with an exception catching mechanism, for faults that were unanticipated by the programmer. It should be noted that error virtualization is different from traditional exception handling as implemented by some programming languages, where the programmer must deliberately create exceptions in the program code and also add code to handle these exceptions. Under error virtualization, failures and exceptions that were unanticipated by, for example, the programmer can be caught, and existing application code can be used to handle them. In some embodiments, error virtualization can be implemented through the instruction-level emulator. Alternatively, error virtualization may be implemented through additional source code that is inserted in the application's source code directly. This insertion of such additional source code can be performed automatically, following the detection of a failure or following the prediction of a failure as described above, or it may be done under the direction of a programmer, system operator, or other suitable user having access to the application's source code.

Using error virtualization, when an exception occurs during the emulation or if the system detects that a fault has occurred, the system may return the program state to its original settings and force an error return from the currently executing function. To determine the appropriate error value, the system analyzes the declared type of function. In some embodiments, the system may analyze the declared type of function using, for example, a TXL script. Generally, TXL is a hybrid function and rule-based language that may be used for performing source-to-source transformation and for rapidly prototyping new languages and language processors.

16

Based on the declared type of function, the system determines the appropriate error value and places it in the stack frame of the returning function. The appropriate error value may be determined based at least in part on heuristics. For example, if the return type is an int, a value of -1 is returned. If the return type is an unsigned int, the system returns a 0. If the function returns a pointer, the system determines whether the returned pointer is further dereferenced by the parent function. If the returned pointer is further dereferenced, the system expands the scope of the emulation to include the parent function. In another example, the return error code may be determined using information embedded in the source code of the application, or through additional information provided to the system by the application programmer, system administrator or third party.

In some embodiments, the emulate_end() is located and the emulation terminates. Because the emulator saved the state of the application before starting and kept track of memory modification during the application's execution, the system is capable of reversing any memory changes made by the code function inside which the fault occurred by returning it to its original setting, thereby nullifying the effect of the instructions processed through emulation. That is, the emulated portion of the code is sliced off and the execution of the code along with its side effects in terms of changes to memory have been rolled back.

For example, the emulator may not be able to perform system calls directly without kernel-level permissions. Therefore, when the emulator decodes an interruption with an intermediate value of 0x80, the emulator releases control to the kernel. However, before the kernel executes the system call, the emulator can back-up the real registers and replace them with its own values. An INT 0x80 can be issued by the emulator and the kernel processes the system call. Once control returns to the emulator, the emulator can update its registers and restore the original values in the application's registers.

If the instrumented application does not crash after the forced return, the system has successfully found a vaccine for the specific fault, which may be used on the actual application running on the server. At 370, the system can then update the application based at least in part on the emulation.

In accordance with some embodiments, artificial diversity features may be provided to mitigate the security risks of software monoculture.

FIG. 6 is a simplified flowchart illustrating the various steps performed in using an application community to monitor an application for faults and repair the application in accordance with some embodiments. This is a generalized flow chart. It will be understood that the steps shown in FIG. 6 may be performed in any suitable order, some may be deleted, and others added.

Generally, the system may divide an application's code into portions of code at 610. Each portion or slice of the application's code may, for example, be assigned to one of the members of the application community (e.g., workstation, server, etc.). Each member of the application community may monitor the portion of the code for various types of failures at 620. As described previously, failures include programming errors, exceptions, software faults (e.g., illegal memory accesses, division by zero, buffer overflow attacks, TOCTTOU violations, etc.), threats (e.g., computer viruses, worms, trojans, hackers, key recovery attacks, malicious executables, probes, etc.), and any other suitable fault that may cause abnormal application termination or adversely affect the one or more applications.

US 8,074,115 B2

17

For example, the system may divide the portions of code based on the size of the application and the number of members in the application community (i.e., size of the application/members in the application community). Alternatively, the system may divide the portions of code based on the amount of available memory in each of the members of the application community. Any suitable approach for determining how to divide up the application's code may also be used. Some suitable approaches are described hereinafter.

For example, the system may examine the total work in the application community, W , by examining the cost of executing discrete slices of the application's code. Assuming a set of functions, F , that comprise an application's callgraph, the i^{th} member of F is denoted as f_i . The cost of executing each f_i is a function of the amount of computation present in f_i (i.e., x_i) and the amount of risk in f_i (i.e., v_i). The calculation of x_i can be driven by at least two metrics: o_i , the number of machine instructions executed as part of f_i , and t_i , the amount of time spent executing f_i . Both o_i and t_i may vary as a function of time or application workload according to the application's internal logic. For example, an application may perform logging or cleanup duties after the application passes a threshold number of requests.

In some embodiments, a cost function may be provided in two phases. The first phase calculates the cost due to the amount of computation for each f_i . The second phase normalizes this cost and applies the risk factor v_i to determine the final cost of each f_i and the total amount of work in the system. For example, let

$$T = \sum_{i=1}^N x_i$$

If $C(f_i, x_i) = x_i/T * 100$, each cost may be normalized by grouping a subset of F to represent one unit of work.

In some embodiments, the system may account for the measure of a function's vulnerability. For example, the system treats v_i as a discrete variable with a value of α , where α takes on a range of values according to the amount of risk such that:

$$v_i = \begin{cases} \alpha & \text{(if } f_i \text{ is vulnerable)} \\ 1 & \text{(if } f_i \text{ is not vulnerable)} \end{cases}$$

Given v_i for each function, the system may determine the total amount of work in the system and the total number of members needed for monitoring:

$$W = N_{\text{vuln}} = \sum_{i=1}^n v_i * r_i$$

After the system (e.g., a controller) or after each application community member has calculated the amount of work in the system, work units can be distributed. In one example, a central controller or one of the workstations may assign each node approximately W/N work units. In another suitable example, each member of the application community may determine its own work set. Each member may iterate through the list of work units flipping a coin that is weighted with the value $v_i * r_i$. Therefore, if the result of the flip is "true," then the member adds that work unit to its work set.

Alternatively, the system may generate a list having $n * W$ slots. Each function can be represented by a number of entries on the list (e.g., $v_i * r_i$). Every member of the application community can iterate through the list, for example, by ran-

18

domly selecting true or false. If true, the application community member monitors the function of the application for a given time slice. Because heavily weighted functions have more entries in the list, a greater number of users may be assigned to cover the application. The member may stop when its total work reaches W/N . Such an approach offers statistical coverage of the application.

In some embodiments, a distributed bidding approach may be used to distribute the workload of monitoring and repairing an application. Each node in the callgraph G has a weight $v_i * r_i$. Some subset of the nodes in F is assigned to each application community member such that each member does no more work than W/N work. The threshold can be relaxed to be within some range ϵ of W/N , where ϵ is a measure of system fairness. Upon calculating the globally fair amount of work W/N , each application community member may adjust its workload by bargaining with other members using a distributed bidding approach.

Two considerations impact the assignment of work units to application community members. First, the system can allocate work units with higher weights, as these work units likely have a heavier weight due to a high v_i . Even if the weight is derived solely from the performance cost, assigning more members to the work units with higher weights is beneficial because these members can round-robin the monitoring task so that any one member does not have to assume the full cost. Second, in some situations, $v_i * r_i$ may be greater than the average amount of work, W/N . Achieving fairness means that $v_i * r_i$ defines the quantity of application community members that is assigned to it and the sum of these quantities defines the minimum number of members in the application community.

In some embodiments, each application community member calculates a table. An example of such a table is shown in FIG. 7. Upon generating the table, application community members may place bids to adjust each of their respective workloads. For example, the system may use tokens for bidding. Tokens may map directly to the number of time quanta that an application community member is responsible for monitoring a work unit or a function of an application. The system ensures that each node does not accumulate more than the total number of tokens allowed by the choice of ϵ .

If an application community member monitors more than its share, then the system has increased coverage and can ensure that faults are detected as quickly as possible. As shown in 630 and 640, each application community member may predict that a fault may occur in the assigned portion of code or may detect that a fault has occurred causing the application to abort, where the assigned portion of the code was the source of the fault. As faults are detected, applications members may each proactively monitor assigned portions of code containing the fault to prevent the application from further failures. As discussed previously, the application community member may isolate the portion of the code that caused the fault and use the emulator to test vaccines or fixes. At 650, the application community member that detects or predicts the fault may notify the other application community members. Other application members that have succumbed to the fault may be restarted with the protection mechanisms or fixes generated by the application member that detected the fault.

Assuming a uniform random distribution of new faults across the application community members, the probability of a fault happening at a member, k , is: $P(\text{fault}) = 1/N$. Thus, the probability of k detecting a new fault is the probability that the fault happens at k and that k detects the fault: $P(\text{fault at$

US 8,074,115 B2

19

$k \Delta \text{detection} = 1/N * k_i$, where k_i is the percentage of coverage at k . The probability of the application community detecting the fault is:

$$P(\text{AC detect}) = \sum_{i=1}^N \frac{1}{N} * k_i$$

As each k_i goes to 100%, the above-equation becomes

$$\sum_{i=1}^N \frac{1}{N}$$

or N/N , a probability of 1 that the fault is detected when it first occurs.

It will also be understood that various embodiments may be presented in terms of program procedures executed on a computer or network of computers.

A procedure is here, and generally, conceived to be a self-consistent sequence of steps leading to a desired result. These steps are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared and otherwise manipulated. It proves convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like. However, all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities.

Further, the manipulations performed are often referred to in terms, such as adding or comparing, which are commonly associated with mental operations performed by a human operator. No such capability of a human operator is necessary, or desirable in many cases, in any of the operations described herein in connection with various embodiments; the operations are machine operations. Useful machines for performing the operation of various embodiments include general purpose digital computers or similar devices.

Some embodiments also provide apparatuses for performing these operations. These apparatuses may be specially constructed for the required purpose or it may comprise a general purpose computer as selectively activated or reconfigured by a computer program stored in the computer. The procedures presented herein are not inherently related to a particular computer or other apparatus. Various general purpose machines may be used with programs written in accordance with the teachings herein, or it may prove more convenient to construct more specialized apparatus to perform the described method. The required structure for a variety of these machines will appear from the description given.

Some embodiments may include a general purpose computer, or a specially programmed special purpose computer. The user may interact with the system via e.g., a personal computer or over PDA, e.g., the Internet an Intranet, etc. Either of these may be implemented as a distributed computer system rather than a single computer. Similarly, the communications link may be a dedicated link, a modem over a POTS line, the Internet and/or any other method of communicating between computers and/or users. Moreover, the processing could be controlled by a software program on one or more computer systems or processors, or could even be partially or wholly implemented in hardware.

20

Although a single computer may be used, systems according to one or more embodiments are optionally suitably equipped with a multitude or combination of processors or storage devices. For example, the computer may be replaced by, or combined with, any suitable processing system operative in accordance with the concepts of various embodiments, including sophisticated calculators, hand held, laptop/notebook, mini, mainframe and super computers, as well as processing system network combinations of the same. Further, portions of the system may be provided in any appropriate electronic format, including, for example, provided over a communication line as electronic signals, provided on CD and/or DVD, provided on optical disk memory, etc.

Any presently available or future developed computer software language and/or hardware components can be employed in such embodiments. For example, at least some of the functionality mentioned above could be implemented using Visual Basic, C, C++ or any assembly language appropriate in view of the processor being used. It could also be written in an object oriented and/or interpretive environment such as Java and transported to multiple destinations to various users.

Other embodiments, extensions, and modifications of the ideas presented above are comprehended and within the reach of one skilled in the field upon reviewing the present disclosure. Accordingly, the scope of the present invention in its various aspects is not to be limited by the examples and embodiments presented above. The individual aspects of the present invention, and the entirety of the invention are to be regarded so as to allow for modifications and future developments within the scope of the present disclosure. For example, the set of features, or a subset of the features, described above may be used in any suitable combination. The present invention is limited only by the claims that follow.

What is claimed is:

1. A method for detecting anomalous program executions, comprising:
 - executing at least a part of a program in an emulator;
 - comparing a function call made in the emulator to a model of function calls for the at least a part of the program;
 - identifying the function call as anomalous based on the comparison; and
 - upon identifying the anomalous function call, notifying an application community that includes a plurality of computers of the anomalous function call.
2. The method of claim 1, further comprising creating a combined model from at least two models created using different computers.
3. The method of claim 1, further comprising creating a combined model from at least two models created at different times.
4. The method of claim 1, further comprising modifying the function call so that the function call becomes non-anomalous.
5. The method of claim 1, further comprising generating a virtualized error in response to the function call being identified as being anomalous.
6. The method of claim 1, wherein the comparing compares the function call name and arguments to the model.
7. The method of claim 1, wherein the model reflects normal activity of the at least a part of the program.
8. The method of claim 1, wherein the model reflects attacks against the at least a part of the program.
9. The method of claim 1, further comprising randomly selecting the model as to be used in the comparison from a plurality of different models relating to the program.

21

22

10. The method of claim 1, further comprising randomly selecting a portion of the model to be used in the comparison.

11. A non-transitory computer-readable medium containing computer-executable instructions that, when executed by a processor, cause the processor to perform a method for detecting anomalous program executions, comprising:

- executing at least a part of a program in an emulator;
- comparing a function call made in the emulator to a model of function calls for the at least a part of the program;
- identifying the function call as anomalous based on the comparison; and
- upon identifying the anomalous function call, notifying an application community that includes a plurality of computers of the anomalous function call.

12. The medium of claim 11, wherein the method further comprises creating a combined model from at least two models created using different computers.

13. The medium of claim 11, wherein the method further comprises creating a combined model from at least two models created at different times.

14. The medium of claim 11, wherein the method further comprises modifying the function call so that the function call becomes non-anomalous.

15. The medium of claim 11, wherein the method further comprises generating a virtualized error in response to the function call being identified as being anomalous.

16. The medium of claim 11, wherein the comparing compares the function call name and arguments to the model.

17. The medium of claim 11, wherein the model reflects normal activity of the at least a part of the program.

18. The medium of claim 11, wherein the model reflects attacks against the at least a part of the program.

19. The medium of claim 11, wherein the method further comprises randomly selecting the model as to be used in the comparison from a plurality of different models relating to the program.

20. The medium of claim 11, wherein the method further comprises randomly selecting a portion of the model to be used in the comparison.

21. A system for detecting anomalous program executions, comprising:

- a digital processing device that:
 - executes at least a part of a program in an emulator;
 - compares a function call made in the emulator to a model of function calls for the at least a part of the program; and
 - identifies the function call as anomalous based on the comparison; and
 - upon identifying the anomalous function call, notifies an application community that includes a plurality of computers of the anomalous function call.

22. A method for detecting anomalous program executions, comprising:

- modifying a program to include indicators of program-level function calls being made during execution of the program;
- comparing at least one of the indicators of program-level function calls made in an emulator to a model of function calls for at least a part of the program; and
- identifying a function call corresponding to the at least one of the indicators as anomalous based on the comparison.

23. The method of claim 22, further comprising creating a combined model from at least two models created using different computers.

24. The method of claim 22, further comprising creating a combined model from at least two models created at different times.

25. The method of claim 22, further comprising modifying the function call so that the function call becomes non-anomalous.

26. The method of claim 22, further comprising generating a virtualized error in response to the function call being identified as being anomalous.

27. The method of claim 22, wherein the comparing compares the function call name and arguments to the model.

28. The method of claim 22, wherein the model reflects normal activity of the at least a part of the program.

29. The method of claim 22, wherein the model reflects attacks against the at least a part of the program.

30. The method of claim 22, further comprising randomly selecting the model as to be used in the comparison from a plurality of different models relating to the program.

31. The method of claim 22, further comprising randomly selecting a portion of the model to be used in the comparison.

32. A non-transitory computer-readable medium containing computer-executable instructions that, when executed by a processor, cause the processor to perform a method for detecting anomalous program executions, comprising:

- modifying a program to include indicators of program-level function calls being made during execution of the program;
- comparing at least one of the indicators of program-level function calls made in an emulator to a model of function calls for at least a part of the program; and
- identifying a function call corresponding to the at least one of the indicators as anomalous based on the comparison.

33. The medium of claim 32, wherein the method further comprises creating a combined model from at least two models created using different computers.

34. The medium of claim 32, wherein the method further comprises creating a combined model from at least two models created at different times.

35. The medium of claim 32, wherein the method further comprises modifying the function call so that the function call becomes non-anomalous.

36. The medium of claim 32, wherein the method further comprises generating a virtualized error in response to the function call being identified as being anomalous.

37. The medium of claim 32, wherein the comparing compares the function call name and arguments to the model.

38. The medium of claim 32, wherein the model reflects normal activity of the at least a part of the program.

39. The medium of claim 32, wherein the model reflects attacks against the at least a part of the program.

40. The medium of claim 32, wherein the method further comprises randomly selecting the model as to be used in the comparison from a plurality of different models relating to the program.

41. The medium of claim 32, wherein the method further comprises randomly selecting a portion of the model to be used in the comparison.

42. A system for detecting anomalous program executions, comprising:

- a digital processing device that:
 - modifies a program to include indicators of program-level function calls being made during execution of the program;
 - compares at least one of the indicators of program-level function calls made in an emulator to a model of function calls for at least a part of the program; and
 - identifies a function call corresponding to the at least one of the indicators as anomalous based on the comparison.

* * * * *

10



(12) **United States Patent**
Stolfo et al.

(10) **Patent No.:** **US 8,601,322 B2**
(45) **Date of Patent:** ***Dec. 3, 2013**

(54) **METHODS, MEDIA, AND SYSTEMS FOR
DETECTING ANOMALOUS PROGRAM
EXECUTIONS**

(56) **References Cited**

U.S. PATENT DOCUMENTS

(75) Inventors: **Salvatore J. Stolfo**, Ridgewood, NJ
(US); **Angelos D. Keromytis**, New York,
NY (US); **Stylianios Sidiroglou**, Astoria,
NY (US)

5,398,196 A 3/1995 Chambers
5,696,822 A 12/1997 Nachenberg
(Continued)

FOREIGN PATENT DOCUMENTS

(73) Assignee: **The Trustees of Columbia University
in the City of New York**, New York, NY
(US)

GB 2277151 6/1997
JP 2002368820 12/2002
KR 20010089062 9/2001

OTHER PUBLICATIONS

(*) Notice: Subject to any disclaimer, the term of this
patent is extended or adjusted under 35
U.S.C. 154(b) by 0 days.

This patent is subject to a terminal dis-
claimer.

"Trojan/XTCP", Panda Software's Center for Virus Control, Jun. 22,
2002, available at: <http://www.ntsecurity.net/Panda/Index.cfm?FuseAction=Virus&VirusID=659>.

(Continued)

(21) Appl. No.: **13/301,741**

Primary Examiner — Nadeem Iqbal

(22) Filed: **Nov. 21, 2011**

(74) *Attorney, Agent, or Firm* — Byrne Poh LLP

(65) **Prior Publication Data**

(57) **ABSTRACT**

US 2012/0151270 A1 Jun. 14, 2012

Methods, media, and systems for detecting anomalous pro-
gram executions are provided. In some embodiments, meth-
ods for detecting anomalous program executions are pro-
vided, comprising: executing at least a part of a program in an
emulator; comparing a function call made in the emulator to
a model of function calls for the at least a part of the program;
and identifying the function call as anomalous based on the
comparison. In some embodiments, methods for detecting
anomalous program executions are provided, comprising:
modifying a program to include indicators of program-level
function calls being made during execution of the program;
comparing at least one of the indicators of program-level
function calls made in the emulator to a model of function
calls for the at least a part of the program; and identifying a
function call corresponding to the at least one of the indicators
as anomalous based on the comparison.

Related U.S. Application Data

(63) Continuation of application No. 12/091,150, filed as
application No. PCT/US2006/041591 on Oct. 25,
2006, now Pat. No. 8,074,115.

(60) Provisional application No. 60/730,289, filed on Oct.
25, 2005.

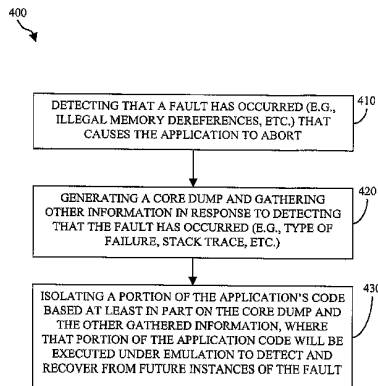
(51) **Int. Cl.**
G06F 11/00 (2006.01)

(52) **U.S. Cl.**
USPC **714/38.1**

(58) **Field of Classification Search**
USPC 714/28-34, 37, 38.1, 38.11, 38.12, 39,
714/42, 47.2, 49-51

See application file for complete search history.

27 Claims, 8 Drawing Sheets



US 8,601,322 B2

Page 2

(56)

References Cited

U.S. PATENT DOCUMENTS

5,964,889	A	10/1999	Nachenberg	
5,968,113	A	10/1999	Haley et al.	
5,978,917	A	11/1999	Chi	
6,067,535	A	5/2000	Hobson et al.	
6,079,031	A	6/2000	Haley et al.	
6,154,876	A	11/2000	Haley et al.	
6,357,008	B1	3/2002	Nachenberg	
6,718,469	B2	4/2004	Pak et al.	
6,952,776	B1	10/2005	Chess	
7,069,583	B2	6/2006	Yann et al.	
7,096,215	B2	8/2006	Bates et al.	
7,155,708	B2	12/2006	Hammes et al.	
7,331,062	B2	2/2008	Alagna et al.	
7,356,736	B2	4/2008	Natvig	
7,373,524	B2*	5/2008	Motsinger et al.	713/194
7,409,717	B1	8/2008	Szor	
7,412,723	B2	8/2008	Blake et al.	
7,490,268	B2	2/2009	Keromytis et al.	
7,496,898	B1*	2/2009	Vu	717/127
7,523,500	B1	4/2009	Szor et al.	
7,526,758	B2*	4/2009	Hasse et al.	717/133
7,603,715	B2	10/2009	Costa et al.	
7,639,714	B2	12/2009	Stolfo et al.	
7,644,441	B2	1/2010	Schmid et al.	
7,647,589	B1	1/2010	Dobrovolskiy et al.	
7,716,736	B2	5/2010	Radatti et al.	
7,748,038	B2	6/2010	Olivier et al.	
7,818,781	B2*	10/2010	Golan et al.	726/1
7,832,012	B2	11/2010	Huddleston	
7,877,807	B2	1/2011	Shipp	
7,975,059	B2	7/2011	Wang et al.	
8,074,115	B2*	12/2011	Stolfo et al.	714/38.1
8,135,994	B2*	3/2012	Keromytis et al.	714/38.11
8,214,900	B1	7/2012	Satish et al.	
8,341,743	B2	12/2012	Rogers et al.	
2001/0020255	A1	9/2001	Hofmann et al.	
2002/0095607	A1	7/2002	Lin-Hendel	
2002/0194490	A1	12/2002	Halperin et al.	
2004/0153644	A1	8/2004	McCorkendale et al.	
2004/0153823	A1	8/2004	Ansari	
2005/0086333	A1	4/2005	Chefalas et al.	
2005/0086630	A1	4/2005	Chefalas et al.	
2005/0108562	A1	5/2005	Khazan et al.	
2006/0010495	A1	1/2006	Cohen et al.	
2006/0168329	A1	7/2006	Tan et al.	
2006/0265694	A1*	11/2006	Chilimbi et al.	717/124
2007/0283338	A1*	12/2007	Gupta et al.	717/154
2008/0016574	A1	8/2008	Tomaselli	
2009/0037682	A1	2/2009	Armstrong et al.	
2009/0038008	A1	2/2009	Pike	
2012/0167120	A1	6/2012	Hentunen	

OTHER PUBLICATIONS

"Using Network-Based Application Recognition and Access Control Lists for Blocking the 'Code Red' Worm at network Ingress Points", Technical Report No. 27842, Cisco Systems, Inc., Aug. 2, 2006, available at: http://www.cisco.com/image/gif/paws/27842/nbar_acl_codered.pdf.

Aleph One, "Smashing the Stack for Fun and Profit", In Phrack, vol. 7, No. 49, Nov. 1996.

Amarsinghe, S.P., "On the Run—Building Dynamic Program Modifiers for Optimization, Introspection, and Security", In the Conference on Programming Language Design and Implementation (PLDI), 2002.

Apap, F., et al., "Detecting Malicious Software by Monitoring Anomalous Windows Registry Accesses", In Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID '02), Zurich, CH, Oct. 16-18, 2002, pp. 36-53.

Armstrong, D., et al., "Controller-Based Autonomic Defense System", In Proceedings of the Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX'03), vol. 2, Washington, DC, USA, Apr. 22-24, 2003, pp. 21-23.

Ashcraft, K. and Engler, D., "Using Programmer-Written Compiler Extensions to Catch Security Holes", In Proceedings of the IEEE Symposium on Security and Privacy, Berkeley, CA, USA, May 12-15, 2002, pp. 143-159.

Avizienis, A., "The N-Version Approach to Fault-Tolerant Software", In IEEE Transactions on Software Engineering, vol. SE-11, No. 12, Dec. 1985, pp. 1491-1501.

Baecher, P. and Koetter, M., "libemu", Jul. 22, 2011, available at: <http://libemu.carnivore.it/>.

Baratloo, A., et al., "Transparent Run-Time Defense Against Stack Smashing Attacks", In Proceedings of the 2000 USENIX Annual Technical Conference (ATEC '00), San Diego, CA, USA, Jun. 18-23, 2000, pp. 251-262.

Barrantes, E.G., et al., "Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks", In Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS '03) Conference, Washington, DC, USA, Oct. 27-31, 2003, pp. 281-289.

Baumgartner, K., "The ROP Pack", Abstract, In Proceedings of the 20th Virus Bulletin International Conference, Vancouver, BC, CA, Sep. 29-Oct. 1, 2010.

Baumgartner, K., "The ROP Pack", Presentation, In Proceedings of the 20th Virus Bulletin International Conference, Vancouver, BC, CA, Sep. 29-Oct. 1, 2010.

Bellovin, S.M., "Distributed Firewalls", In ;login: Magazine, Nov. 1999, pp. 37-39.

Bhatkar, S., et al., "Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits", In Proceedings of the 12th USENIX Security Symposium (SSYM '03), Washington, DC, USA, Aug. 4-8, 2003, pp. 105-120.

Bhattacharyya, M., et al., "MET: An Experimental System for Malicious Email Tracking", In Proceedings of the 2002 Workshop on New Security Paradigms (NSPW '02), Virginia Beach, VA, USA, Sep. 23-26, 2002, pp. 3-10.

Brilliant, S.S., et al., "Analysis of Faults in an N-Version Software Experiment", In IEEE Transactions on Software Engineering, vol. 16, No. 2, Feb. 1990, pp. 238-247.

Bruening, D., et al., "An Infrastructure for Adaptive Dynamic Optimization", In Proceedings of the International Symposium on Code Generation and Optimization (CGO 2003), Mar. 23-26, 2003, San Francisco, CA, USA, pp. 265-275.

Buchanan, E., et al., "Return-Oriented Programming: Exploits Without Code Injection", Presentation, Black Hat USA, Las Vegas, NV, USA, Aug. 2-7, 2008.

Bulba and Kil3r, "Bypassing StackGuard and StackShield", In Phrack, No. 56, May 1, 2000.

Candea, G. and Fox, A., "Crash-Only Software", In Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-IX), Lihue, Hawaii, USA, May 18-21, 2003, pp. 67-72.

CERT Advisory CA-2001-19, "'Code Red' Worm Exploiting Buffer Overflow in IIS Indexing Service DLL", Jul. 19, 2001, available at: <http://www.cert.org/advisories/CA-2001-19.html>.

CERT Advisory CA-2003-04, "MS-SQL Server Worm", Jan. 27, 2003, available at: <http://www.cert.org/advisories/CA-2003-04.html>.

CERT Advisory CA-2003-19, "Exploitation of Vulnerabilities in Microsoft RPC Interface", Jul. 31, 2003, available at: <http://www.cert.org/advisories/CA-2003-19.html>.

CERT Advisory CA-2003-20: "W32/Blaster Worm", Aug. 14, 2003, available at: <http://www.cert.org/advisories/CA-2003-20.html>.

Chan, P.K., et al., "A Machine Learning Approach to Anomaly Detection", Technical Report CS-2003-06, Department of Computer Science, Florida Institute of Technology, Mar. 29, 2003, pp. 1-13.

Checkoway, S., et al., "Return-Oriented Programming Without Returns", In Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10) Chicago, IL, USA, Oct. 4-8, 2010, pp. 559-572.

Chen, H., et al., "MOPS: An Infrastructure for Examining Security Properties of Software", In Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS '02), Washington, DC, USA, Nov. 18-22, 2002, pp. 235-244.

US 8,601,322 B2

Page 3

(56)

References Cited

OTHER PUBLICATIONS

- Chess, B.V., "Improving Computer Security Using Extended Static Checking", In Proceedings of the IEEE Symposium on Security and Privacy, Berkeley, CA, USA, May 12-15, 2002, pp. 160-173.
- Chew, M. and Song, D., "Mitigating Buffer Overflows by Operating System Randomization", Technical Report, CMU-CS-02-197, Carnegie Mellon University, Dec. 2002.
- Christodorescu, M. and Jha, S., "Static Analysis of Executables to Detect Malicious Patterns", In Proceedings of the 12th USENIX Security Symposium (SSYM '03), Washington, DC, USA, Aug. 4-8, 2003, pp. 169-186.
- Cohen, F., "Computer Viruses: Theory and Experiments", In Computers & Security, vol. 6, No. 1, Feb. 1987, pp. 22-35.
- Conover, M., "w00w00 on Heap Overflows", Technical Report, Jan. 1999, available at: <http://www.w00w00.org/files/articles/heap.txt>.
- Corelan Team, "Corelan ROPDB", 2012, available at: <https://www.corelan.be/index.php/security/corelan-ropdb/>.
- Cova, M., et al., "Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code", In Proceedings of the 19th International Conference on World Wide Web (WWW '10), Raleigh, NC, USA, Apr. 26-30, 2010, pp. 281-290.
- Cowan, C., et al., "FormatGuard: Automatic Protection From printf Format String Vulnerabilities", In Proceedings of the 10th USENIX Security Symposium (SSYM '01), Washington, DC, USA, Aug. 13-17, 2001, pp. 191-199.
- Cowan, C., et al., "PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities", In Proceedings of the 12th USENIX Security Symposium (SSYM '03), Washington, DC, USA, Aug. 4-8, 2003, pp. 91-104.
- Cowan, C., et al., "Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks", In Proceedings of the 7th USENIX Security Symposium (SSYM '98), San Antonio, TX, USA, Jan. 26-29, 1998, pp. 63-78.
- Cowan, C., et al., "SubDomain: Parsimonious Server Security", In Proceedings of the 14th USENIX System Administration Conference (LISA 2000), New Orleans, LA, USA, Dec. 3-8, 2000, pp. 341-354.
- Crosby, S.A., and Wallach, D.S., "Denial of Service via Algorithmic Complexity Attacks", In Proceedings of the 12th USENIX Security Symposium (SSYM '03), Washington, DC, USA, Aug. 4-8, 2003, pp. 29-44.
- Damashek, M., "Gauging Similarity with N-Grams: Language-Independent Categorization of Text", In Science, vol. 267, No. 5199, Feb. 10, 1995, pp. 843-848.
- Demsky, B. and Rinard, M., "Automatic Data Structure Repair for Self-Healing Systems", In Proceedings of the First Workshop on Algorithms and Architectures for Self-Managing Systems, San Diego, CA, USA, Jun. 2003.
- Demsky, B., and Rinard, M., "Automatic Detection and Repair of Errors in Data Structures", In Proceedings of the 18th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA '03), Anaheim, CA, USA, Oct. 26-30, 2003, pp. 78-95.
- Denning, D.E., "An Intrusion Detection Model", In IEEE Transactions on Software Engineering, vol. SE-13, No. 2, Feb. 1987, pp. 222-232.
- Dierks, T., and Allen, C., "The TLS Protocol Version 1.0", Technical Report, RFC 2246, Jan. 1999.
- Dunlap, G.W., et al., "ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay", In Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02), Boston, MA, USA, Dec. 9-11, 2002, pp. 211-224.
- Egele, M., et al., "Defending Browsers against Drive-By Downloads: Mitigating Heap-Spraying Code Injection Attacks", In Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '09), Como, IT, Jul. 9-10, 2009, pp. 88-106.
- Engler, D. and Ashcraft, K., "RacerX: Effective, Static Detection of Race Conditions and Deadlocks", In Proceedings of the ACM Symposium on Operating Systems Principles (SOSP 2003), Bolton Landing, NY, USA, Oct. 19-22, 2003.
- Erlingsson, U., "Low-Level Software Security: Attack and Defenses", Technical Report MSR-TR-07-153, Microsoft Corporation, Nov. 2007, available at: <http://research.microsoft.com/pubs/64363/tr-2007-153.pdf>.
- Eskin, E., "Anomaly Detection Over Noisy Data Using Learned Probability Distributions", In Proceedings of the Seventeenth International Conference on Machine Learning (ICML-2000), Stanford, CA, USA, Jun. 29-Jul. 2, 2000, pp. 255-262.
- Etoh, J., "GCC Extension for Protecting Applications From Stack-Smashing Attacks", IBM Research, Aug. 22, 2005, available at: <http://www.tr.ibm.com/projects/security/ssp>.
- Forrest, S., et al., "A Sense of Self for Unix Processes", in Proceedings of the IEEE Symposium on Security and Privacy, May 6-8, 1996, Oakland, CA, USA, pp. 120-128.
- Forrest, S., et al., "Building Diverse Computer Systems", In Proceedings of the Sixth Workshop on Hot Topics in Operating Systems, Cape Cod, MA, USA, May 5-6, 1997, pp. 67-72.
- Frantzen, M. and Shuey, M., "StackGhost: Hardware Facilitated Stack Protection", In Proceedings of the 10th USENIX Security Symposium (SSYM '01), Washington, DC, USA, Aug. 13-17, 2001, pp. 55-66.
- Friedman, N., and Singer, Y., "Efficient Bayesian Parameter Estimation in Large Discrete Domains", In Proceedings of the 1998 Conference on Advances in Neural Information Processing Systems II, Denver, CO, USA, Nov. 30-Dec. 5, 1998, pp. 417-423.
- Garfinkel, T. and Rosenblum, M., "A Virtual Machine Introspection Based Architecture for Intrusion Detection", in Proceedings of the 10th Network and Distributed System Security Symposium (NDSS '03), San Diego, CA, USA, Feb. 6-7, 2003, pp. 191-206.
- Garfinkel, T., "Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools", In Proceedings of the Network and Distributed Systems Security Symposium (NDSS '03), San Diego, CA, USA, Feb. 6-7, 2003, pp. 163-176.
- Geer, Jr., D.E., "Monopoly Considered Harmful", In IEEE Security & Privacy, vol. 1, No. 6, Nov./Dec. 2003, pp. 14, 17.
- Ghosh, A.K. and Schwartzbard, A., "A Study in Using Neural Networks for Anomaly and Misuse Detection", In Proceedings of the Proceedings of the 8th USENIX Security Symposium, Washington, DC, USA, Aug. 23-26, 1999.
- Goldberg, I., et al., "A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker)", In Proceedings of the Sixth USENIX Security Symposium (SSYM '96), San Jose, CA, USA, Jul. 22-25, 1996.
- Goth, G., "Addressing the Monoculture", In IEEE Security & Privacy, vol. 99, No. 6, Nov./Dec. 2003, pp. 8-10.
- Hangal, S., and Lam, M.S., "Tracking Down Software Bugs Using Automatic Anomaly Detection", In Proceedings of the 24th International Conference on Software Engineering (ICSE '02), Orlando, FL, USA, May 19-25, 2002, pp. 291-301.
- Hensing, R., "Understanding DEP as a Mitigation Technology", Microsoft, Jun. 12, 2009, available at: <http://blogs.technet.com/b/srd/archive/2009/06/12/understanding-dep-as-a-mitigation-technology-part-1.aspx>.
- Hofmeyr, S.A., et al., "Intrusion Detection System Using Sequences of System Calls", In Journal of Computer Security, vol. 6, No. 3, Aug. 18, 1998, pp. 151-180.
- Honig, A., et al., "Adaptive Model Generation: An Architecture for the Deployment of Data Mining-Based Intrusion Detection Systems", In Applications of Data Mining in Computer Security, 2002, pp. 153-194.
- International Patent Application No. PCT/US2012/055824, filed Sep. 17, 2012.
- International Preliminary Report on Patentability in International Patent Application No. PCT/US2006/041591, filed Oct. 25, 2006, mailed Apr. 2, 2009.
- International Search Report in International Patent Application No. PCT/US2006/041591, filed Oct. 25, 2006, mailed Jun. 25, 2008.
- International Search Report in International Patent Application No. PCT/US2012/055824, filed Sep. 17, 2012, mailed Dec. 7, 2012.
- Internet Engineering Task Force, "Intrusion Detection Exchange Format", IETF.org, Oct. 15, 2010, available at: <http://datatracker.ietf.org/wg/idwg/charter/>.

Columbia EQ 0000099

Exhibit Page 199

Symantec v. Columbia

A137

IPR2015-00375

US 8,601,322 B2

Page 4

(56)

References Cited

OTHER PUBLICATIONS

- Ioannidis, J. and Bellovin, S.M., "Implementing Push-Back: Router-Based Defense Against DDoS Attacks", In Proceedings of the ISOC Symposium on Network and Distributed System Security (SNDSS), San Diego, CA, USA, Feb. 2002.
- Ioannidis, S., et al., "Implementing a Distributed Firewall", In Proceedings of the 7th ACM Conference on Computer and Communications Security (CCS'00), Athens, GR, Nov. 1-4, 2000, pp. 190-199.
- Janakiraman, R., et al., "Indra: A Peer-to-Peer Approach to Network Intrusion Detection and Prevention", In Proceedings of the 12th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Linz, AT, Jun. 9-11, 2003, pp. 226-231.
- Javitz, H.S. and Valdes, A., "The NIDES Statistical Component: Description and Justification", Technical Report 3131, SRI International, Computer Science Laboratory, Mar. 7, 1994, pp. 1-47.
- Jim, T., et al., "Cyclone: A Safe Dialect of C", In Proceedings of the USENIX Annual Technical Conference (ATEC '02), Monterey, CA, USA, Jun. 10-15, 2002, pp. 275-288.
- Jones, R.W.M. and Kelly, P.H.J., "Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs", In Proceedings of the Third International Workshop on Automated Debugging, Linköping, SE, May 26-28, 1997, pp. 13-26.
- Just, J.E., et al., "Learning Unknown Attacks—A Start", In Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID'02), Zurich, CH, Oct. 16-18, 2002, pp. 158-176.
- Kc, G.S., et al., "Countering Code-Injection Attacks With Instruction-Set Randomization", In Proceedings of the ACM Computer and Communications Security (CCS '03) Conference, Washington, DC, USA, Oct. 27-30, 2003, pp. 272-280.
- Kent, S., and Atkinson, R., "Security Architecture for the Internet Protocol", Technical Report, RFC 2401, Nov. 1998.
- Kephart, J.O., et al., "A Biologically Inspired Immune System for Computers", In Artificial Life IV, MIT Press, 1994, pp. 130-139.
- King, S.T. and Chen, P.M., "Backtracking Intrusions", In Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003 (SOSP 2003), Bolton Landing, NY, USA, Oct. 19-22, 2003, pp. 223-236.
- King, S.T., et al., "Operating System Support for Virtual Machines", In Proceedings of the USENIX Annual Technical Conference (ATEC '03), San Antonio, TX, USA, Jun. 9-14, 2003, pp. 71-84.
- Kiriansky, V., et al., "Secure Execution Via Program Shepherding", In Proceedings of the 11th USENIX Security Symposium (SSYM '02), San Francisco, CA, USA, Aug. 5-9, 2002, pp. 191-205.
- Kodialam, M. and Lakshman, T.V., "Detecting Network Intrusions via Sampling: A Game Theoretic Approach", In the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2003), San Francisco, CA, USA, Mar. 30-Apr. 3, 2003.
- Kruegel, C., et al., "Polymorphic Worm Detection Using Structural Information of Executables", In Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID'05), Seattle, WA, USA, Sep. 7-9, 2005, pp. 207-226.
- Krugel, C., et al., "Service Specific Anomaly Detection for Network Intrusion Detection", In Proceedings of the 2002 ACM Symposium on Applied Computing (SAC 2002), Madrid, ES, Mar. 10-14, 2002, pp. 201-208.
- Larochelle, D. and Evans, D., "Statically Detecting Likely Buffer Overflow Vulnerabilities", In Proceedings of the 10th USENIX Security Symposium (SSYM '01), Washington, DC, USA, Aug. 13-17, 2001, pp. 177-190.
- Larson, E. and Austin, T., "High Coverage Detection of Input-Related Security Faults", In Proceedings of the 12th Conference on USENIX Security Symposium (SSYM'03), vol. 12, Aug. 2003, pp. 121-136.
- Lee, W. and Stolfo, S.J., "A Framework for Constructing Features and Models for Intrusion Detection Systems", In ACM Transactions on Information and System Security, vol. 3, No. 4, Nov. 2000, pp. 227-261.
- Lee, W. and Stolfo, S.J., "Data Mining Approaches for Intrusion Detection", In Proceedings of the 7th Conference on USENIX Security Symposium, San Antonio, TX, USA, Jan. 26-29, 1998.
- Lee, W., "A Data Mining Framework for Constructing Features and Models for Intrusion Detection Systems", PhD Thesis, Columbia University, 1999, pp. 1-177.
- Lee, W., et al., "A Data Mining Framework for Building Intrusion Detection Models", In Proceedings of the 1999 IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 9-12, 1999, pp. 120-132.
- Lee, W., et al., "Learning Patterns from Unix Process Execution Traces for Intrusion Detection", In Proceedings of the AAAI97 Workshop on AI Methods in Fraud and Risk Management, Providence, RI, USA, Jul. 27, 1997, pp. 50-56.
- Lee, W., et al., "Mining in a Data-Flow Environment: Experiences in Intrusion Detection", In Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '99), San Diego, CA, Aug. 1999, pp. 114-124.
- Lhee, K. and Chapin, S.J., "Type-Assisted Dynamic Buffer Overflow Detection", In Proceedings of the 11th USENIX Security Symposium (SSYM '02), San Francisco, CA, USA, Aug. 5-9, 2002, pp. 81-90.
- Lin, M.J., et al., "A New Model for Availability in the Face of Self-Propagating Attacks", In Proceedings of the 1998 Workshop on New Security Paradigms (NSPW '98), Charlottesville, VA, USA, Sep. 22-25, 1998, pp. 134-137.
- Lippmann, R., et al., "The 1999 DARPA Off-Line Intrusion Detection Evaluation", In Computer Networks, vol. 34, No. 4, Oct. 2000, pp. 579-595.
- Liston, T., "Welcome to My Tarpit: The Tactical and Strategic Use of LaBrea", Feb. 17, 2003, available at: <http://download.polytechnic.edu/na/pub4/download.sourceforge.net/pub/sourceforge/1/la/labrea/OldFiles/LaBrea-Tom-Liston-Whitepaper-Welcome-to-my-tarpit.txt>.
- Mahoney, M. V. and Chan, P.K., "An Analysis of the 1999 DARPA/Lincoln Laboratory Evaluation Data for Network Anomaly Detection", In Proceedings of the 6th International Symposium Recent Advances in Intrusion Detection (RAID '03), Pittsburgh, PA, USA, Sep. 8-10, 2003, pp. 220-237.
- Mahoney, M.V. and Chan, P.K., "Detecting Novel Attacks by Identifying Anomalous Network Packet Headers", Technical Report CS-2001-2, Florida Institute of Technology, Melbourne, FL, 2001.
- Mahoney, M.V. and Chan, P.K., "Learning Models of Network Traffic for Detecting Novel Attacks", Technical Report CS-2002-08, Florida Institute of Technology, 2002, pp. 1-48.
- Mahoney, M.V. and Chan, P.K., "Learning Nonstationary Models of Normal Network Traffic for Detecting Novel Attacks", In Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Edmonton, AB, CA, Jul. 23-26, 2002, pp. 376-385.
- Mahoney, M.V., "Network Traffic Anomaly Detection Based on Packet Bytes", In Proceedings of the 2003 ACM Symposium on Applied Computing (SAC '03), Melbourne, FL, USA, Mar. 9-12, 2003, pp. 346-350.
- Malton, A., "The Denotational Semantics of a Functional Tree-Manipulation Language", In Computer Languages, vol. 19, No. 3, Jul. 1993, pp. 157-168.
- Miller, T.C. and de Raadt, T., "strncpy and strncat-Consistent, Safe, String Copy and Concatenation", In Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference, Monterey, CA, USA, Jun. 6-11, 1999.
- Moore, D., "The Spread of the Sapphire/Slammer Worm", Technical Report, Apr. 3, 2003, available at: <http://www.silicondefense.com/research/worms/slammer.php>.
- Moore, D., et al., "Code-Red: A Case Study on the Spread and Victims of an Internet Worm", In Proceedings of the 2nd Internet Measurement Workshop (IMW'02), Marseille, FR, Nov. 6-8, 2002, pp. 273-284.
- Moore, D., et al., "Internet Quarantine: Requirements for Containing Self-Propagating Code", In IEEE Societies Twenty-Second Annual Joint Conference of the IEEE Computer and Communications, vol. 3, Mar. 30-Apr. 3, 2003, pp. 1901-1910.

US 8,601,322 B2

Page 5

(56)

References Cited

OTHER PUBLICATIONS

- Mosberger, D., and Jin, T., "httpperf—A Tool for Measuring Web Server Performance", In ACM SIGMETRICS Performance Evaluation Review, vol. 26, No. 3, Dec. 1998, pp. 31-37.
- Nachenberg, C., "Computer Virus—Coevolution", In Communications of the ACM, vol. 40, No. 1, Jan. 1997, pp. 46-51.
- Nethercote, N., and Seward, J., "Valgrind: A Program Supervision Framework", In Electronic Notes in Theoretical Computer Science, vol. 89, No. 2, 2003, pp. 44-66.
- Newsome, J., and Song, D., "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software", In Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS 2005), Feb. 2005.
- Nojiri, D., et al., "Cooperative Response Strategies for Large Scale Attack Mitigation", In 3rd DARPA Information Survivability Conference and Exposition (DISCEX-III 2003), vol. 1, Washington, DC, USA, Apr. 22-24, 2003, pp. 293-302.
- Oplinger, J. and Lam, M.S., "Enhancing Software Reliability with Speculative Threads", In Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), San Jose, CA, USA, Oct. 5-9, 2002, pp. 184-196.
- Paxson, V. "Bro: A System for Detecting Network Intruders in Real-Time", In Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, Jan. 26-29, 1998.
- Peterson, D.S., et al., "A Flexible Containment Mechanism for Executing Untrusted Code", In Proceedings of the 11th USENIX Security Symposium (SSYM '02), San Francisco, CA, USA, Aug. 5-9, 2002, pp. 207-225.
- Polychronakis, M., et al., "Emulation-Based Detection of Non-Self-Contained Polymorphic Shellcode", In Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection (RAID'07), Gold Coast, AU, Sep. 5-7, 2007, pp. 87-106.
- Polychronakis, M., et al., "Comprehensive Shellcode Detection Using Runtime Heuristics", In Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC '10), Austin, TX, USA, Dec. 6-10, 2010, pp. 287-296.
- Polychronakis, M., et al., "Network-Level Polymorphic Shellcode Detection Using Emulation", In Proceedings of the Third Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA '06), Berlin, DE, Jul. 13-14, 2006, pp. 54-73.
- Porras, P.A. and Neumann, P.G., "EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances", In Proceedings of the 20th National Information Systems Security Conference, Oct. 9, 1997, pp. 353-365.
- Prasad, M. and Chiueh, T., "A Binary Rewriting Defense Against Stack Based Buffer Overflow Attacks", In Proceedings of the USENIX Annual Technical Conference (ATEC '03), Boston, MA, USA, Jun. 9-14, 2003, pp. 211-224.
- Prevelakis, V. and Spinellis, D., "Sandboxing Applications", In Proceedings of the USENIX Annual Technical Conference (ATEC '01), Boston, MA, USA, Jun. 25-30, 2001, pp. 119-126.
- Prevelakis, V., "A Secure Station for Network Monitoring and Control", In Proceedings of the 8th USENIX Security Symposium, Washington, DC, USA, Aug. 23-26, 1999, pp. 1-8.
- Provos, N., "Improving Host Security with System Call Policies", In Proceedings of the 12th USENIX Security Symposium (SSYM '03), Washington, DC, USA, Aug. 4-8, 2003, pp. 257-272.
- Provos, N., et al., "Preventing Privilege Escalation", In Proceedings of the 12th conference on USENIX Security Symposium (SSYM'03), Washington, DC, USA, Aug. 4-8, 2003.
- Ratanaworabhan, P., et al., "NOZZLE: A Defense Against Heap-Spraying Code Injection Attacks", In Proceedings of the 18th USENIX Security Symposium, Montreal, CA, Aug. 10-14, 2009, pp. 169-186.
- Reynolds, J., et al., "Online Intrusion Protection by Detecting Attacks with Diversity", In Proceedings of the Sixteenth International Conference on Data and Applications Security, Cambridge, UK, Jul. 28-31, 2002, pp. 245-256.
- Reynolds, J., et al., "The Design and Implementation of an Intrusion Tolerant System", In Proceedings of the International Conference on Dependable Systems and Networks (DSN 2002), Jun. 23-26, 2002, Bethesda, MD, USA, pp. 285-292.
- Rinard, M., et al. "Enhancing Server Availability and Security Through Failure-Oblivious Computing", In Proceedings 6th Symposium on Operating systems Design and Implementation (OSDI), Dec. 2004, pp. 303-316.
- Rinard, M., et al., "A Dynamic Technique for Eliminating Buffer Overflow Vulnerabilities (and Other Memory Errors)", In Proceedings 20th Annual Computer Security Applications Conference (ACSAC '04), Tucson, AZ, USA, Dec. 6-10, 2004, pp. 82-90.
- Roesch, M., "Snort: Lightweight Intrusion Detection for Networks", In Proceedings of the 13th Conference on Systems Administration (LISA '99), Seattle, WA, USA, Nov. 7-12, 1999, pp. 229-238.
- Rosenblum, M., et al., "Using the SimOS Machine Simulator to Study Complex Computer Systems", In ACM Transactions on Modeling and Computer Simulation, vol. 7, No. 1, Jan. 1997, pp. 78-103.
- Rudys, A., and Wallach, D.S., "Termination in Language-based Systems", In ACM Transactions on Information and System Security (TISSEC), vol. 5, No. 2, May 2002, pp. 138-168.
- Rudys, A., and Wallach, D.S., "Transactional Rollback for Language-Based Systems", In Proceedings of the International Conference on Dependable Systems and Networks (DSN '02), Bethesda, MD, USA, Jun. 23-26, 2002, pp. 439-448.
- Scholkopf, B., et al., "Estimating the Support of a High-Dimensional Distribution", Technical Report MSR-TR-99-87, Microsoft Research, Sep. 18, 2000, pp. 1-30.
- Schultz, M.G., et al. "MEF: Malicious Email Filter—A UNIX Mail Filter that Detects Malicious Windows Executables", In Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, Boston, MA, USA, Jun. 25-30, 2001, pp. 245-252.
- Sekar, R., et al. "Specification-Based Anomaly Detection: A New Approach for Detecting Network Intrusions", In ACM Conference on Computer and Communications Security (CCS'02), Washington, DC, USA, Nov. 18-22, 2002.
- Sekar, R., et al., "Model-Carrying Code: A Practical Approach for Safe Execution of Untrusted Applications", In Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03), Bolton Landing, NY, USA, Oct. 19-22, 2003, pp. 15-28.
- Seward, J. and Nethercote, N., "Valgrind, An Open-Source Memory Debugger for x86-GNU/Linux", May 5, 2003, available at: <http://developer.kde.org/~sewardj/>.
- Shacham, H., "The Geometry of Innocent Flesh on the Bone: Return-into-Libc Without Function Calls (On the x86)", Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07), Alexandria, VA, USA, Oct. 29-Nov. 2, 2007, pp. 552-561.
- Shacham, H., et al., "On the Effectiveness of Address-Space Randomization", In Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS '04), Washington, DC, USA, Oct. 25-29, 2004, pp. 298-307.
- Shoch, J.F. and Hupp, J.A., "The 'Worm' Programs—Early Experiments with a Distributed Computation", In Communications of the ACM, vol. 22, No. 3, Mar. 1982, pp. 172-180.
- Sidiroglou, S. and Keromytis, A.D., "A Network Worm Vaccine Architecture", In Proceedings of the Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2003), Workshop on Enterprise Security, Linz, AT, Jun. 9-11, 2003, pp. 220-225.
- Sidiroglou, S., et al. "Building a Reactive Immune System for Software Services", In Proceedings of the 2005 USENIX Annual Technical Conference (USENIX 2005), Anaheim, CA, USA, Apr. 10-15, 2005, pp. 149-161.
- Smirnov, A. and Chiueh, T., "DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks", In Proceedings of the 12th Symposium on Network and Distributed System Security (NDSS'05), Feb. 2005.
- Snow, K.Z., et al., "ShellOS: Enabling Fast Detection and Forensic Analysis of Code Injection Attacks", In Proceedings of the 20th USENIX Security Symposium (SEC'11), San Francisco, CA, USA, Aug. 8-12, 2011.

US 8,601,322 B2

Page 6

(56)

References Cited

OTHER PUBLICATIONS

- Solar Designer, "Getting Around Non-Executable Stack (and Fix)", Seclists.org, Aug. 10, 1997, available at: <http://seclists.org/bugtraq/1997/Aug/63>.
- Sole, P., "Hanging on a ROPE", Presentation, Immunity.com, Sep. 20, 2010, available at: http://www.immunitysec.com/downloads/DEPLIB20_ekoparty.pdf.
- Song, D., et al., "A Snapshot of Global Internet Worm Activity", Technical Report, Arbor Networks, Nov. 13, 2001, pp. 1-7.
- Spafford, E.H., "The Internet Worm Program: An Analysis", Technical Report, Purdue University Technical Report CSD-TR-823, Dec. 8, 1988, pp. 1-40.
- Stamp, M., "Risks of Monoculture", in Communications of the ACM, vol. 47, No. 3, Mar. 2004, p. 120.
- Stanford, S., et al., "How to Own the Internet in Your Spare Time", In Proceedings of the 11th USENIX Security Symposium, Aug. 5-9, 2002, San Francisco, CA, USA, pp. 149-167.
- Sugerman, J., et al., "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor", In Proceedings of the 2001 USENIX Annual Technical Conference (USENIX 2001), Boston, MA, USA, Jun. 25-30, 2001, pp. 1-14.
- Suh, G.E., et al., "Secure Program Execution via Dynamic Information Flow Tracking", In Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLoS 2004) Boston, MA, USA, Oct. 7-13, 2004, pp. 85-96.
- Symantec, "Happy99.Worm", Symantec.com, Feb. 13, 2007, available at: <http://www.symantec.com/qvcenter/venc/data/happy99.worm.html>.
- Taylor, C. and Alves-Foss, J., "NATE—Network Analysis of Anomalous Traffic Events, A Low-Cost Approach", In New Security Paradigms Workshop (NSPW'01), Cloudcroft, NM, USA, Sep. 10-13, 2002, pp. 89-96.
- Toth, T. and Kruegel, C., "Accurate Buffer Overflow Detection via Abstract Payload Execution", In Proceedings of the 5th Symposium on Recent Advances in Intrusion Detection (Raid '02) Zurich, CH, Oct. 16-18, 2002, pp. 274-291.
- Toth, T. and Kruegel, C., "Connection-History Based Anomaly Detection", In Proceedings of the 2002 IEEE Workshop on Information Assurance and Security, West Point, NY, USA, Jun. 17-19, 2002, pp. 30-35.
- Toyoizumi, H. and Kara, A., "Predators: Good Will Mobile Codes Combat against Computer Viruses", In Proceedings of the 2002 Workshop on New Security Paradigms (NSPW '02), Virginia Beach, VA, USA, Sep. 23-26, 2002, pp. 11-17.
- Twycross, T. and Williamson, M.M., "Implementing and Testing a Virus Throttle", In Proceedings of the 12th USENIX Security Symposium, Washington, DC, USA, Aug. 4-8, 2003, pp. 285-294.
- Tzermias, Z., et al., "Combining Static and Dynamic Analysis for the Detection of Malicious Documents", In Proceedings of the Fourth European Workshop on System Security (EUROSEC '11), Salzburg, AT, Apr. 2011.
- U.S. Appl. No. 12/091,150, filed Apr. 22, 2008.
- U.S. Appl. No. 60/730,289, filed Oct. 25, 2005.
- U.S. Appl. No. 61/535,288, filed Sep. 15, 2011.
- Vendicator, "Stack Shield: A 'Stack Smashing' Technique Protection Tool for Linux", Jan. 7, 2000, available at: <http://angelfire.com/sk/stackshield>.
- Vigna, G. and Kemmerer, R., "NetSTAT: A Network-Based Intrusion Detection System", In Journal of Computer Security, vol. 7, No. 1, 1999, pp. 37-71.
- Wang, C., et al., "On Computer Viral Infection and the Effect of Immunization", In Proceedings of the 16th Annual Computer Security Applications Conference (ACSAC '00), New Orleans, LA, USA, Dec. 11-15, 2000, pp. 246-256.
- Wang, N., et al., "Y-Branched: When You Come to a Fork in the Road, Take It", In Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT '03), New Orleans, LA, USA, Sep. 27-Oct. 1, 2003, pp. 56-66.
- Wang, X., et al., "SigFree: A Signature-free Buffer Overflow Attack Blocker", In Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, CA, Jul. 31-Aug. 4, 2006, pp. 225-240.
- Warrender, C., et al., "Detecting Intrusions Using System Calls: Alternative Data Models", In IEEE Symposium on Security and Privacy (S&P '99), Oakland, CA, USA, May 9-12, 1999, pp. 133-145.
- Whittaker, A., et al., "Scale and Performance in the Denali Isolation Kernel", In Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02), vol. 36, No. SI, Boston, MA, USA, Dec. 9-11, 2002, pp. 195-209.
- White, S.R., "Open Problems in Computer Virus Research", Technical Report, IBM Thomas J. Watson Research Center, Oct. 1998, available at: <http://www.research.ibm.com/antivirus/SciPapers/White/Problems/Problems.html>.
- Whittaker, J.A., "No Clear Answers on Monoculture Issues", In IEEE Security & Privacy, vol. 1, No. 6, Nov./Dec. 2003, pp. 18-19.
- Wilander, J. and Kamkar, M., "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention", In Proceedings of the 10th Network and Distributed System Security Symposium (NDSS '03), San Diego, CA, USA, Feb. 6-7, 2003.
- Williamson, M.M., "Throttling Viruses: Restricting Propagation to Defeat Malicious Mobile Code", Technical Report HPL-2002-172, HP Laboratories Bristol, Dec. 10, 2002, pp. 1-8.
- Written Opinion in International Patent Application No. PCT/US2006/041591, filed Oct. 25, 2006, mailed Jun. 25, 2008.
- Written Opinion in International Patent Application No. PCT/US2012/055824, filed Sep. 17, 2012, mailed Dec. 7, 2012.
- Yin, J., et al., "Separating Agreement from Execution for Byzantine Fault Tolerant Services", In Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03), Bolton Landing, NY, USA, Oct. 19-22, 2003, pp. 253-267.
- Yuan, L., et al., "Security Breaches as PMU Deviation: Detecting and Identifying Security Attacks Using Performance Counters", In Proceedings of the Second Asia-Pacific Workshop on Systems (APSys'11), Shanghai, CN, Jul. 11-12, 2011.
- Zhang, Q., et al., "Analyzing Network Traffic to Detect Self-Decrypting Exploit Code", In Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security (ASIACCS '07), Singapore, Mar. 20-22, 2007, pp. 4-12.
- Zou, C.C., et al., "Code Red Worm Propagation Modeling and Analysis", In Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS '02), Washington, DC, USA, Nov. 18-22, 2002, pp. 138-147.
- Zou, C.C., et al., "Monitoring and Early Warning for Internet Worms", In Proceedings of the 10th ACM International Conference on Computer and Communications Security (CCS '03), Washington, DC, USA, Oct. 27-30, 2003, pp. 190-199.
- Zovi, D.A.D., "Practical Return-Oriented Programming", Presentation, RSA Conference, Mar. 17, 2010, available at: <http://365.rsaconference.com/servlet/JiveServlet/previewBody/2573-102-3-3232/RR-304.pdf>.
- Office Action dated Aug. 23, 2010 in U.S. Appl. No. 12/091,150.
- Ghosh, A.K., et al., "Learning Program Behavior Profiles for Intrusion Detection", In Proceedings of the Workshop on Intrusion Detection and Network Monitoring, Santa Clara, CA, US, Apr. 9-12, 1999, pp. 51-62.
- Inoue, H. and Forrest, S., "Anomaly Intrusion Detection in Dynamic Execution Environments", In New Security Paradigms Workshop, Virginia Beach, VA, US, Sep. 23-26, 2002, pp. 52-60.
- Kolter, J.Z. and Maloof, M., "Learning to Detect Malicious Executables in the Wild", In Proceedings of the 10th International Conference on Knowledge Discovery and Data Mining (KDD '04), Seattle, WA, US, Aug. 22-25, 2004, pp. 470-480.
- Lee, J.S., et al., "A Generic Virus Detection Agent on the Internet", In Proceedings of the 30th Annual Hawaii International Conference on System Sciences, Maui, HI, US, Jan. 7-10, 1997, pp. 210-220.

* cited by examiner

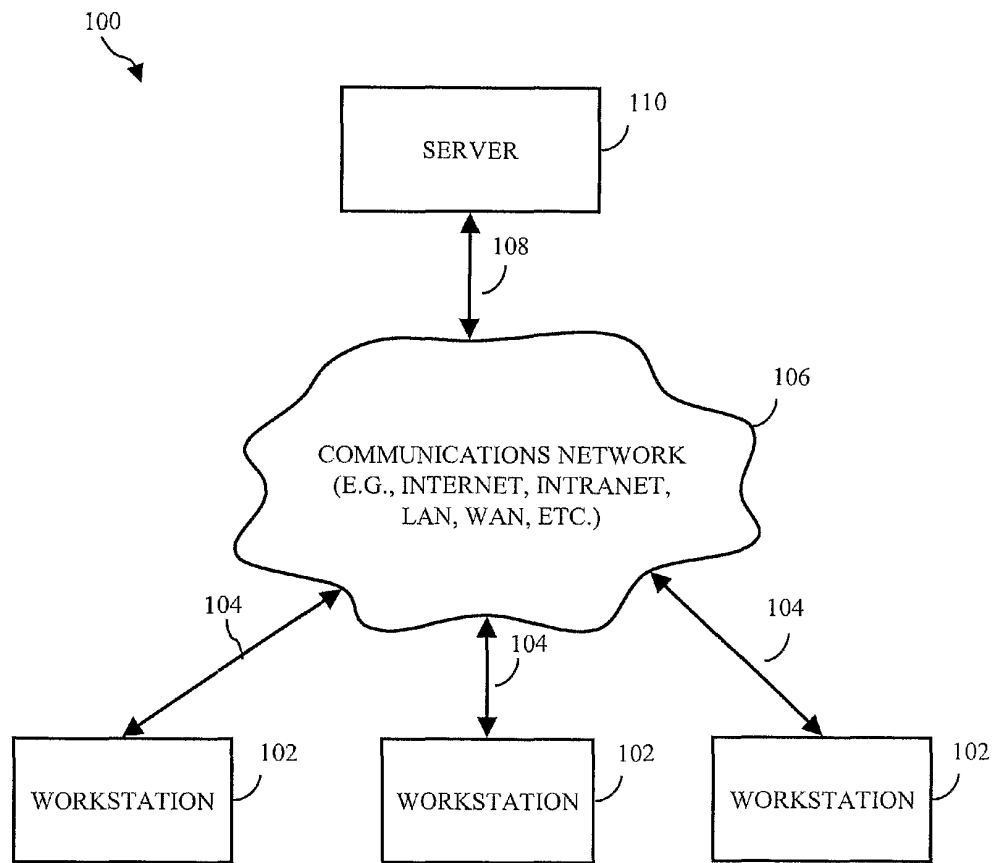


FIG. 1

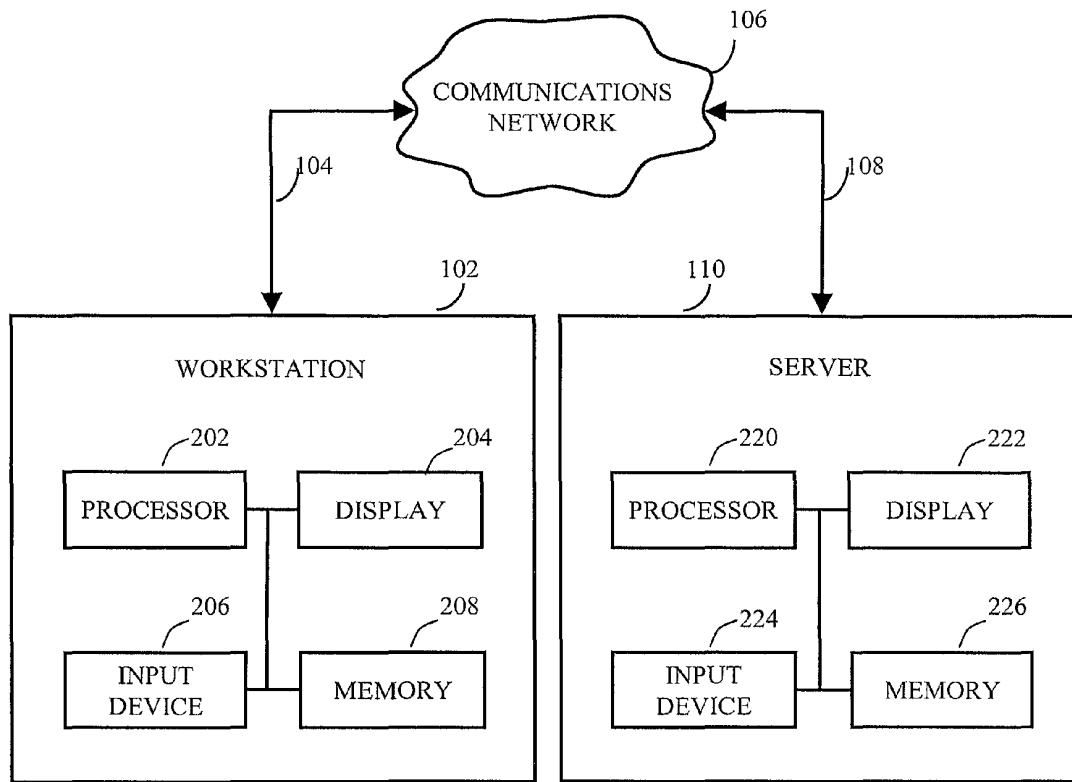


FIG. 2

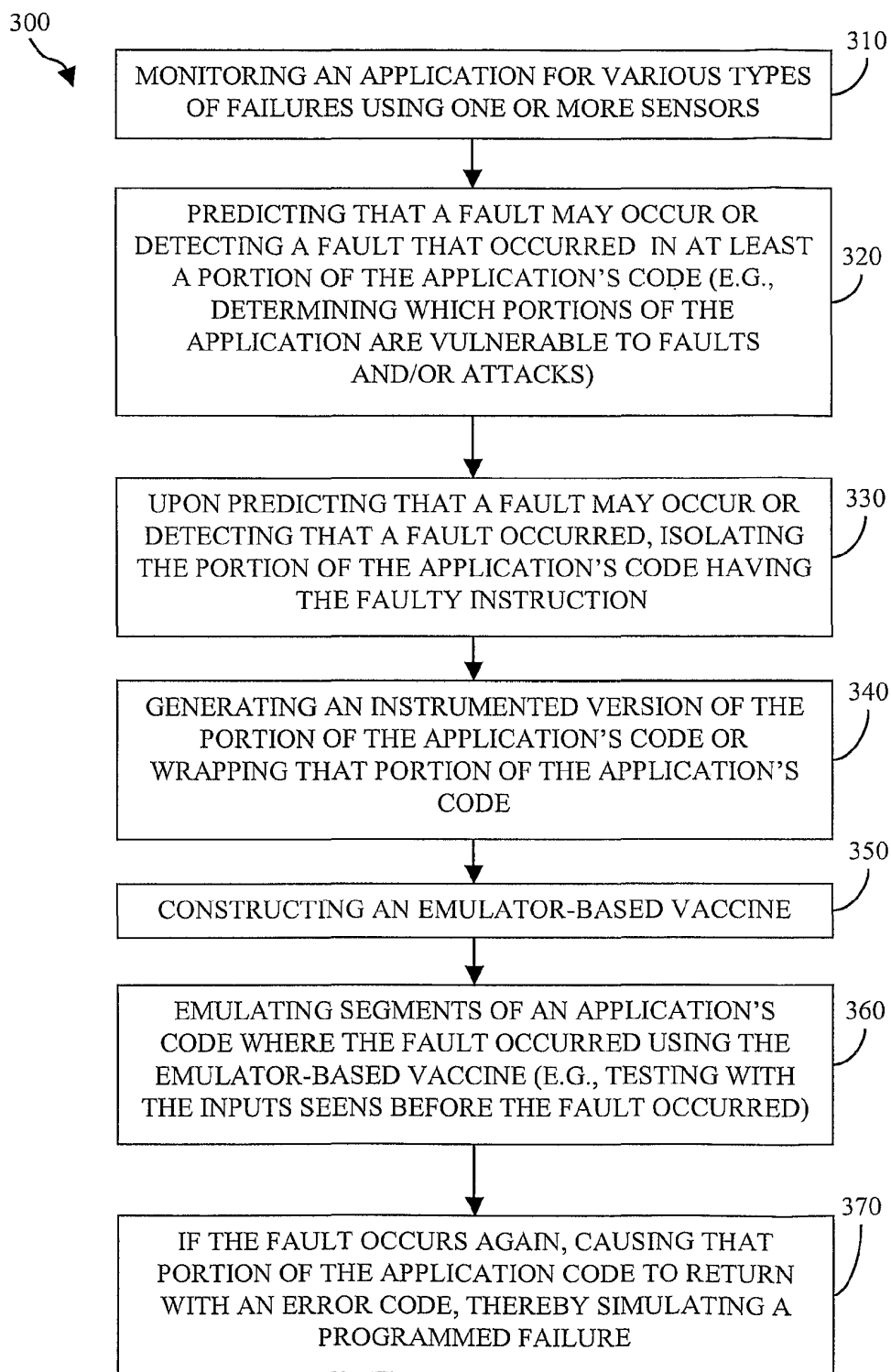


FIG. 3

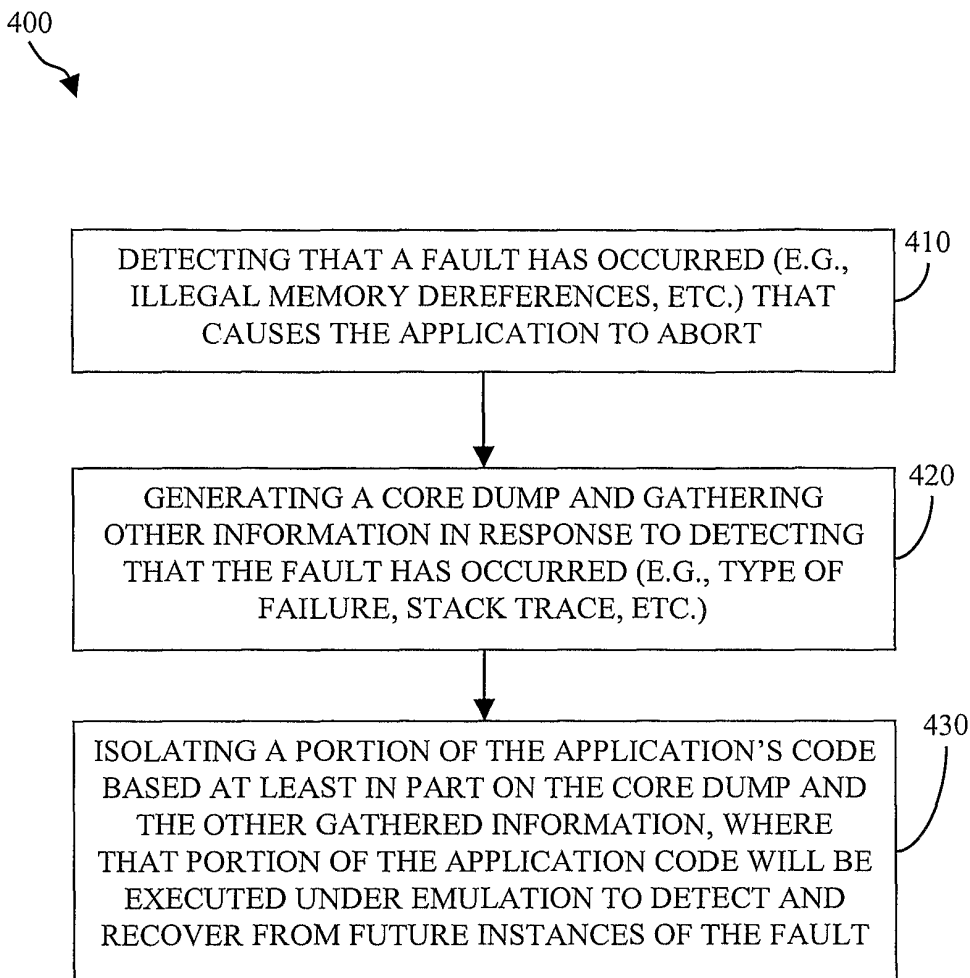


FIG. 4

500
↙

```
void foo() {  
    int a = 1;  
    emulate_init();  
    emulate_begin(p_args);  
    a++;  
    emulate_end();  
    emulate_term();  
    printf("a = %d\n", a);  
}
```

FIG. 5

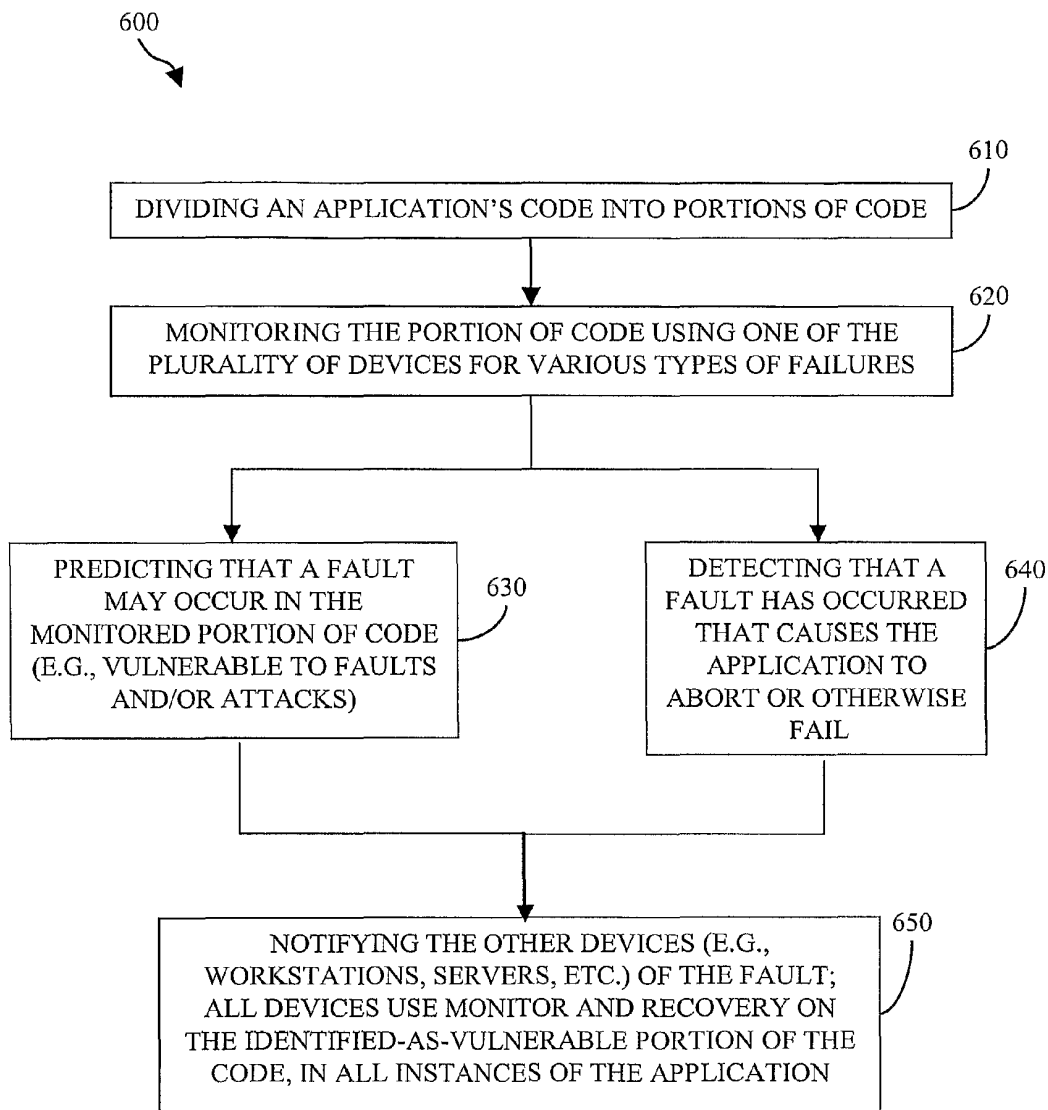


FIG. 6

700


f_i	x_i	r_i	v_i	T	$C(f_i, x_i)$	$r_i * v_i$
a)	100	1	α_1	600	16	α_1
b)	200	2	α_2	600	33	$2\alpha_2$
c)	300	3	α_3	600	50	$3\alpha_3$

FIG. 7

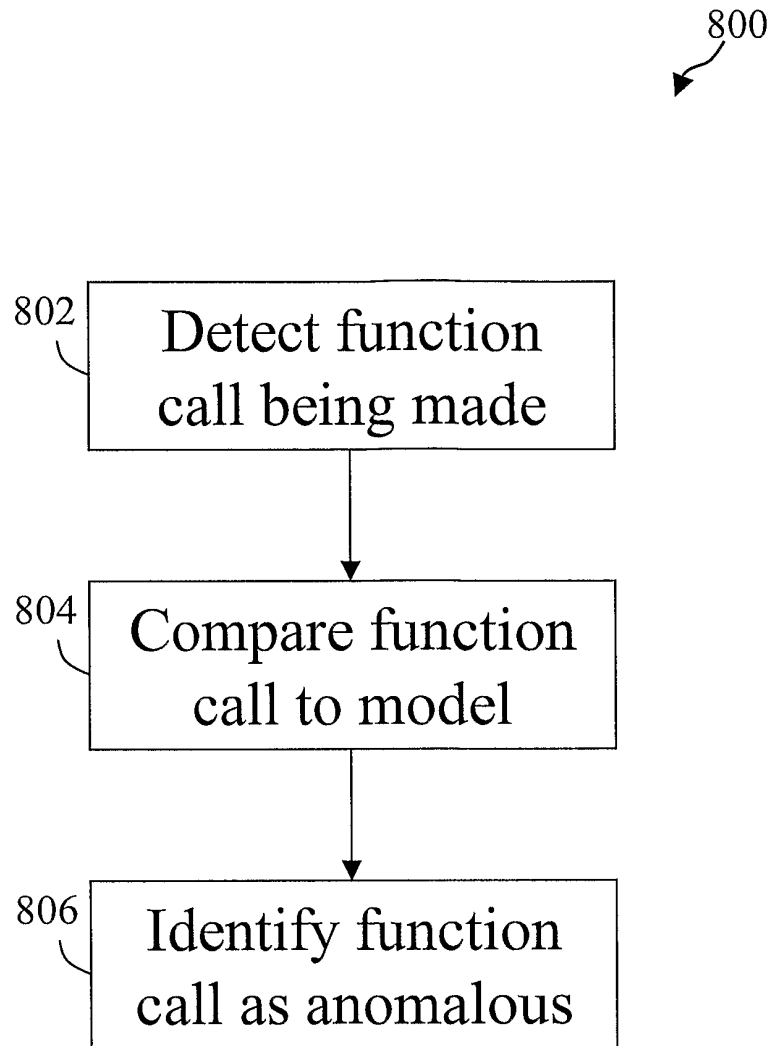


FIG. 8

US 8,601,322 B2

1

**METHODS, MEDIA, AND SYSTEMS FOR
DETECTING ANOMALOUS PROGRAM
EXECUTIONS**

CROSS REFERENCE TO RELATED
APPLICATION

This application is a continuation of U.S. patent application Ser. No. 12/091,150, filed Jun. 15, 2009, which is the U.S. National Phase Application Under 35 U.S.C. §371 of International Application No. PCT/US2006/041591, filed Oct. 25, 2006, which claims the benefit under 35 U.S.C. §119(e) of U.S. Provisional Patent Application. No. 60/730,289, filed Oct. 25, 2005, which are hereby incorporated by reference herein in their entireties.

TECHNOLOGY AREA

The disclosed subject matter relates to methods, media, and systems for detecting anomalous program executions.

BACKGROUND

Applications may terminate due to any number of threats, program errors, software faults, attacks, or any other suitable software failure. Computer viruses, worms, trojans, hackers, key recovery attacks, malicious executables, probes, etc. are a constant menace to users of computers connected to public computer networks (such as the Internet) and/or private networks (such as corporate computer networks). In response to these threats, many computers are protected by antivirus software and firewalls. However, these preventative measures are not always adequate. For example, many services must maintain a high availability when faced by remote attacks, high-volume events (such as fast-spreading worms like Slammer and Blaster), or simple application-level denial of service (DoS) attacks.

Aside from these threats, applications generally contain errors during operation, which typically result from programmer error. Regardless of whether an application is attacked by one of the above-mentioned threats or contains errors during operation, these software faults and failures result in illegal memory access errors, division by zero errors, buffer overflows attacks, etc. These errors cause an application to terminate its execution or "crash."

SUMMARY

Methods, media, and systems for detecting anomalous program executions are provided. In some embodiments, methods for detecting anomalous program executions are provided, comprising: executing at least a part of a program in an emulator; comparing a function call made in the emulator to a model of function calls for the at least a part of the program; and identifying the function call as anomalous based on the comparison.

In some embodiments, computer-readable media containing computer-executable instructions that, when executed by a processor, cause the processor to perform a method for detecting anomalous program executions are provided, the method comprising: executing at least a part of a program in an emulator; comparing a function call made in the emulator to a model of function calls for the at least a part of the program; and identifying the function call as anomalous based on the comparison.

In some embodiments, systems for detecting anomalous program executions are provided, comprising: a digital pro-

2

cessing device that: executes at least a part of a program in an emulator; compares a function call made in the emulator to a model of function calls for the at least a part of the program; and identifies the function call as anomalous based on the comparison.

In some embodiments, methods for detecting anomalous program executions are provided, comprising: modifying a program to include indicators of program-level function calls being made during execution of the program; comparing at least one of the indicators of program-level function calls made in the emulator to a model of function calls for the at least a part of the program; and identifying a function call corresponding to the at least one of the indicators as anomalous based on the comparison.

In some embodiments, computer-readable media containing computer-executable instructions that, when executed by a processor, cause the processor to perform a method for detecting anomalous program executions are provided, the method comprising: modifying a program to include indicators of program-level function calls being made during execution of the program; comparing at least one of the indicators of program-level function calls made in the emulator to a model of function calls for the at least a part of the program; and identifying a function call corresponding to the at least one of the indicators as anomalous based on the comparison.

In some embodiments, systems for detecting anomalous program executions are provided, comprising: a digital processing device that: modifies a program to include indicators of program-level function calls being made during execution of the program; compares at least one of the indicators of program-level function calls made in the emulator to a model of function calls for the at least a part of the program; and identifies a function call corresponding to the at least one of the indicators as anomalous based on the comparison.

BRIEF DESCRIPTION OF THE DRAWINGS

The Detailed Description, including the description of various embodiments of the disclosed subject matter, will be best understood when read in reference to the accompanying figures wherein:

FIG. 1 is a schematic diagram of an illustrative system suitable for implementation of an application that monitors other applications and protects these applications against faults in accordance with some embodiments;

FIG. 2 is a detailed example of the server and one of the workstations of FIG. 1 that may be used in accordance with some embodiments;

FIG. 3 shows a simplified diagram illustrating repairing faults in an application and updating the application in accordance with some embodiments;

FIG. 4 shows a simplified diagram illustrating detecting and repairing an application in response to a fault occurring in accordance with some embodiments;

FIG. 5 shows an illustrative example of emulated code integrated into the code of an existing application in accordance with some embodiments;

FIG. 6 shows a simplified diagram illustrating detecting and repairing an application using an application community in accordance with some embodiments of the disclosed subject matter;

FIG. 7 shows an illustrative example of a table that may be calculated by a member of the application community for distributed bidding in accordance with some embodiments of the disclosed subject matter; and

US 8,601,322 B2

3

FIG. 8 shows a simplified diagram illustrating shows identifying a function call as being anomalous in accordance with some embodiments.

DETAILED DESCRIPTION

Methods, media, and systems for detecting anomalous program executions are provided. In some embodiments, systems and methods are provided that model application level computations and running programs, and that detect anomalous executions by, for example, instrumenting, monitoring and analyzing application-level program function calls and/or arguments. Such an approach can be used to detect anomalous program executions that may be indicative of a malicious attack or program fault.

The anomaly detection algorithm being used may be, for example, a probabilistic anomaly detection (PAD) algorithm or a one class support vector machine (OCSVM), which are described below, or any other suitable algorithm.

Anomaly detection may be applied to process execution anomaly detection, file system access anomaly detection, and/or network packet header anomaly detection. Moreover, as described herein, according to various embodiments, an anomaly detector may be applied to program execution state information. For example, as explained in greater detail below, an anomaly detector may model information on the program stack to detect anomalous program behavior.

In various embodiments, using PAD to model program stack information, such stack information may be extracted using, for example, Selective Transactional EMulation (STEM), which is described below and which permits the selective execution of certain parts, or all, of a program inside an instruction-level emulator, using the Valgrind emulator, by modifying a program's binary or source code to include indicators of what functions calls are being made (and any other suitable related information), or using any other suitable technique. In this manner, it is possible to determine dynamically (and transparently to the monitored program) the necessary information such as stack frames, function-call arguments, etc. For example, one or more of the following may be extracted from the program stack specific information: function name, the argument buffer name it may reference, and other features associated with the data sent to or returned from the called function (e.g., the length in bytes of the data, or the memory location of the data).

For example, as illustrated in FIG. 8, an anomaly detector may be applied, for example, by extracting data pushed onto the stack (e.g., by using an emulator or by modifying a program), and creating a data record provided to the anomaly detector for processing at 802. According to various embodiments, in a first phase, an anomaly detector models normal program execution stack behavior. In the detection mode, after a model has been computed, the anomaly detector can detect stacked function references as anomalous at 806 by comparing those references to the model based on the training data at 804.

Once an anomaly is detected, according to various embodiments, selective transactional emulation (STEM) and error virtualization may be used to reverse (undo) the effects of processing the malicious input (e.g., changes to program variables or the file system) in order to allow the program to recover execution in a graceful manner. In this manner, the precise location of the failed (or attacked) program at which an anomaly was found may be identified. Also, the application of an anomaly detector to function calls can enable rapid detection of malicious program executions, such that it is possible to mitigate against such faults or attacks (e.g., by

4

using patch generation systems, or content filtering signature generation systems). Moreover, given precise identification of a vulnerable location, the performance impact may be reduced by using STEM for parts or all of a program's execution.

As explained above, anomaly detection can involve the use of detection models. These models can be used in connection with automatic and unsupervised learning.

A probabilistic anomaly detection (PAD) algorithm can be used to train a model for detecting anomalies. This model may be, in essence, a density estimation, where the estimation of a density function $p(x)$ over normal data allows the definition of anomalies as data elements that occur with low probability. The detection of low probability data (or events) are represented as consistency checks over the normal data, where a record is labeled anomalous if it fails any one of these tests.

First and second order consistency checks can be applied. First order consistency checks verify that a value is consistent with observed values of that feature in the normal data set. These first order checks compute the likelihood of an observation of a given feature, $P(X_i)$, where X_i are the feature variables. Second order consistency checks determine the conditional probability of a feature value given another feature value, denoted by $P(X_i|X_j)$, where X_i and X_j are the feature variables.

One way to compute these probabilities is to estimate a multinomial that computes the ratio of the counts of a given element to the total counts. However, this results in a biased estimator when there is a sparse data set. Another approach is to use an estimator to determine these probability distributions. For example, let N be the total number of observations, N_i be the number of observations of symbol i , α be the "pseudo count" that is added to the count of each observed symbol, k^0 be the number of observed symbols, and L be the total number of possible symbols. Using these definitions, the probability for an observed element i can be given by:

$$P(X = i) = \frac{N_i + \alpha}{k^0 \alpha + N} C \quad (1)$$

and the probability for an unobserved element i can be:

$$P(X = i) = \frac{1}{L - k^0} (1 - C) \quad (2)$$

where C , the scaling factor, accounts for the likelihood of observing a previously observed element versus an unobserved element. C can be computed as:

$$C = \left(\sum_{k=k^0}^L \frac{k^0 \alpha + N}{k \alpha + N} m_k \right) \left(\sum_{k=k^0} m_k \right)^{-1} \quad (3)$$

where

$$m_k = P(S = k) \frac{k!}{k = k^0} \frac{\Gamma(k \alpha)}{\Gamma(k \alpha + N)} \Big|$$

and $P(s=k)$ is a prior probability associated with the size of the subset of elements in the alphabet that have non-zero probability.

US 8,601,322 B2

5

Because this computation of C can be time consuming, C can also be calculated by:

$$C = \frac{N}{N+L-k^0} \tag{4}$$

The consistency check can be normalized to account for the number of possible outcomes of L by $\log(P/(1/L)) = \log(P) + \log(L)$.

Another approach that may be used instead of using PAD for model generation and anomaly detection is a one class SVM (OCSVM) algorithm. The OCSVM algorithm can be used to map input data into a high dimensional feature space (via a kernel) and iteratively find the maximal margin hyperplane which best separates the training data from the origin. The OCSVM may be viewed as a regular two-class SVM where all the training data lies in the first class, and the origin is taken as the only member of the second class. Thus, the hyperplane (or linear decision boundary) can correspond to the classification rule:

$$f(x) = \langle w, x \rangle + b \tag{5}$$

where w is the normal vector and b is a bias term. The OCSVM can be used to solve an optimization problem to find the rule f with maximal geometric margin. This classification rule can be used to assign a label to a test example x. If $f(x) < 0$, x can be labeled as an anomaly, otherwise it can be labeled as normal. In practice, there is a trade-off between maximizing the distance of the hyperplane from the origin and the number of training data points contained in the region separated from the origin by the hyperplane.

Solving the OCSVM optimization problem can be equivalent to solving the dual quadratic programming problem:

$$\min_{\alpha} \frac{1}{2} \sum_{ij} \alpha_i \alpha_j K(x_i, x_j) \tag{6}$$

subject to the constraints

$$0 \leq \alpha_i \leq \frac{1}{v_i} \tag{7}$$

and

$$\sum_i \alpha_i = 1 \tag{8}$$

where α_i is a lagrange multiplier (or “weight” on example i such that vectors associated with non-zero weights are called “support vectors” and solely determine the optimal hyperplane), v is a parameter that controls the trade-off between maximizing the distance of the hyperplane from the origin and the number of data points contained by the hyperplane, l is the number of points in the training dataset, and $K(x_i, x_j)$ is the kernel function. By using the kernel function to project input vectors into a feature space, nonlinear decision boundaries can be allowed for. Given a feature map:

$$\Phi: X \rightarrow \mathbb{R}^n \tag{9}$$

where Φ maps training vectors from input space X to a high-dimensional feature space, the kernel function can be defined as:

$$K(x, y) = \langle \Phi(x), \Phi(y) \rangle \tag{10}$$

6

Feature vectors need not be computed explicitly, and computational efficiency can be improved by directly computing kernel values $K(x, y)$. Three common kernels can be used:

Linear kernel: $K(x, y) = \langle x, y \rangle$

Polynomial kernel: $K(x, y) = \langle x, y + 1 \rangle^d$, where d is the degree of the polynomial

Gaussian kernel: $K(x, y) = e^{-\|x-y\|^2 / (2\sigma^2)}$, where σ^2 is the variance

Kernels from binary feature vectors can be obtained by mapping a record into a feature space such that there is one dimension for every unique entry for each record value. A particular record can have the value 1 in the dimensions which correspond to each of its specific record entries, and the value 0 for every other dimension in feature space. Linear kernels, second order polynomial kernels, and gaussian kernels can be calculated using these feature vectors for each record. Kernels can also be calculated from frequency-based feature vectors such that, for any given record, each feature corresponds to the number of occurrences of the corresponding record component in the training set. For example, if the second component of a record occurs three times in the training set, the second feature value for that record is three. These frequency-based feature vectors can be used to compute linear and polynomial kernels.

According to various embodiments, “mimicry attacks” which might otherwise thwart OS system call level anomaly detectors by using normal appearing sequences of system calls can be detected. For example, mimicry attacks are less likely to be detected when the system calls are only modeled as tokens from an alphabet, without any information about arguments. Therefore, according to various embodiments, the models used are enriched with information about the arguments (data) such that it may be easier to detect mimicry attacks.

According to various embodiments, models are shared among many members of a community running the same application (referred to as an “application community”). In particular, some embodiments can share models with each other and/or update each other’s models such that the learning of anomaly detection models is relatively quick. For example, instead of running a particular application for days at a single site, according to various embodiments, thousands of replicated applications can be run for a short period of time (e.g., one hour), and the models created based on the distributed data can be shared. While only a portion of each application instance may be monitored, for example, the entire software body can be monitored across the entire community. This can enable the rapid acquisition of statistics, and relatively fast learning of an application profile by sharing, for example, aggregate information (rather than the actual raw data used to construct the model).

Model sharing can result in one standard model that an attacker could potentially access and use to craft a mimicry attack. Therefore, according to various embodiments, unique and diversified models can be created. For example, such unique and diversified models can be created by randomly choosing particular features from the application execution that is modeled, such that the various application instances compute distinct models. In this manner, attacks may need to avoid detection by multiple models, rather than just a single model. Creating unique and diversified models not only has the advantage of being more resistant to mimicry attacks, but also may be more efficient. For example, if only a portion of an application is modeled by each member of an application community, monitoring will generally be simpler (and

US 8,601,322 B2

7

cheaper) for each member of the community. In the event that one or more members of an application community are attacked, according to various embodiments, the attack (or fault) will be detected, and patches or a signature can be provided to those community members who are blind to the crafted attack (or fault).

Random (distinct) model building and random probing may be controlled by a software registration key provided by a commercial off-the-shelf (COTS) software vendor or some other data providing "randomization." For example, for each member of an application community, some particular randomly chosen function or functions and its associated data may be chosen for modeling, while others may simply be ignored. Moreover, because vendors can generate distinct keys and serial numbers when distributing their software, this feature can be used to create a distinct random subset of functions to be modeled. Also, according to various embodiments, even community members who model the same function or functions may exchange models.

According to various embodiments, when an application execution is being analyzed over many copies distributed among a number of application community members to profile the entire code of an application, it can be determined whether there are any segments of code that are either rarely or never executed, and a map can be provided of the code layout identifying "suspect code segments" for deeper analysis and perhaps deeper monitoring. Those segments identified as rarely or never executed may harbor vulnerabilities not yet executed or exploited. Such segments of code may have been designed to execute only for very special purposes such as error handling, or perhaps even for triggering malicious code embedded in the application. Since they are rarely or never executed, one may presume that such code segments have had less regression testing, and may have a higher likelihood of harboring faulty code.

Rarely or never executed code segments may be identified and may be monitored more thoroughly through, for example, emulation. This deep monitoring may have no discernible overhead since the code in question is rarely or never executed. But such monitoring performed in each community member may prevent future disasters by preventing such code (and its likely vulnerabilities) from being executed in a malicious/faulty manner. Identifying such code may be performed by a sensor that monitors loaded modules into the running application (e.g., DLL loads) as well as addresses (PC values) during code execution and creates a "frequency" map of ranges of the application code. For example, a set of such distributed sensors may communicate with each other (or through some site that correlates their collective information) to create a central, global MAP of the application execution profile. This profile may then be used to identify suspect code segments, and then subsequently, this information may be useful to assign different kinds of sensors/monitors to different code segments. For example, an interrupt service routine (ISR) may be applied to these suspect sections of code.

It is noted that a single application instance may have to be run many times (e.g., thousands of times) in order to compute an application profile or model. However, distributed sensors whose data is correlated among many (e.g., a thousand) application community members can be used to compute a substantially accurate code profile in a relatively short amount of time. This time may be viewed as a "training period" to create the code map.

According to various embodiments, models may be automatically updated as time progresses. For example, although a single site may learn a particular model over some period of time, application behavior may change over time. In this case,

8

the previously learned model may no longer accurately reflect the application characteristics, resulting in, for example, the generation of an excessive amount of false alarms (and thus an increase in the false positive rate over time). A possible solution to this "concept drift" issue entails at least two possible approaches, both intended to update models over time. A first approach to solving (or at least reducing the effects of) the "concept drift" issue involves the use of "incremental learning algorithms," which are algorithms that piecemeal update their models with new data, and that may also "expire" parts of the computed model created by older data. This piecemeal incremental approach is intended to result in continuous updating using relatively small amounts of data seen by the learning system.

A second approach to solving (or at least reducing the effect of) the "concept drift" issue involves combining multiple models. For example, presuming that an older model has been computed from older data during some "training epoch," a new model may be computed concurrently with a new epoch in which the old model is used to detect anomalous behavior. Once a new model is computed, the old model may be retired or expunged, and replaced by the new model. Alternatively, for example, multiple models such as described above may be combined. In this case, according to various embodiments, rather than expunging the old model, a newly created model can be algorithmically combined with the older model using any of a variety of suitable means. In the case of statistical models that are based upon frequency counts of individual data points, for example, an update may consist of an additive update of the frequency count table. For example, PAD may model data by computing the number of occurrences of a particular data item, "X." Two independently learned PAD models can thus have two different counts for the same value, and a new frequency table can be readily computed by summing the two counts, essentially merging two tables and updating common values with the sum of their respective counts.

According to various embodiments, the concept of model updating that is readily achieved in the case of computed PAD models may be used in connection with model sharing. For example, rather than computing two models by the same device for a distinct application, two distinct models may be computed by two distinct instances of an application by two distinct devices, as described above. The sharing of models may thus be implemented by the model update process described herein. Hence, a device may continuously learn and update its models either by computing its own new model, or by downloading a model from another application community member (e.g., using the same means involved in the combining of models).

In the manners described above, an application community may be configured to continuously refresh and update all community members, thereby making mimicry attacks far more difficult to achieve.

As mentioned above, it is possible to mitigate against faults or attacks by using patch generation systems. In accordance with various embodiments, when patches are generated, validated, and deployed, the patches and/or the set of all such patches may serve the following.

First, according to various embodiments, each patch may be used as a "pattern" to be used in searching other code for other unknown vulnerabilities. An error (or design flaw) in programming that is made by a programmer and that creates a vulnerability may show up elsewhere in code. Therefore, once a vulnerability is detected, the system may use the detected vulnerability (and patch) to learn about other (e.g., similar) vulnerabilities, which may be patched in advance of

US 8,601,322 B2

9

those vulnerabilities being exploited. In this manner, over time, a system may automatically reduce (or eliminate) vulnerabilities.

Second, according to various embodiments, previously generated patches may serve as exemplars for generating new patches. For example, over time, a taxonomy of patches may be assembled that are related along various syntactic and semantic dimensions. In this case, the generation of new patches may be aided by prior examples of patch generation.

Additionally, according to various embodiments, generated patches may themselves have direct economic value. For example, once generated, patches may be “sold” back to the vendors of the software that has been patched.

As mentioned above, in order to alleviate monitoring costs, instead of running a particular application for days at a single site, many (e.g., thousands) replicated versions of the application may be run for a shorter period of time (e.g., an hour) to obtain the necessary models. In this case, only a portion of each replicated version of the application may be monitored, although the entire software body is monitored using the community of monitored software applications. Moreover, according to various embodiments, if a software module has been detected as faulty, and a patch has been generated to repair it, that portion of the software module, or the entire software module, may no longer need to be monitored. In this case, over time, patch generated systems may have fewer audit/monitoring points, and may thus improve in execution speed and performance. Therefore, according to various embodiments, software systems may be improved, where vulnerabilities are removed, and the need for monitoring is reduced (thereby reducing the costs and overheads involved with detecting faults).

It is noted that, although described immediately above with regard to an application community, the notion of automatically identifying faults of an application, improving the application over time by repairing the faults, and eliminating monitoring costs as repairs are deployed may also be applied to a single, standalone instance of an application (without requiring placements as part of a set of monitored application instances).

Selective transactional emulation (STEM) and error virtualization can be beneficial for reacting to detected failures/attacks in software. According to various embodiments, STEM and error virtualization can be used to provide enhanced detection of some types of attacks, and enhanced reaction mechanisms to some types of attacks/failures.

A learning technique can be applied over multiple executions of a piece of code (e.g., a function or collection of functions) that may previously have been associated with a failure, or that is being proactively monitored. By retaining knowledge on program behavior across multiple executions, certain invariants (or probable invariants) may be learned, whose violation in future executions indicates an attack or imminent software fault.

In the case of control hijacking attacks, certain control data that resides in memory is overwritten through some mechanism by an attacker. That control data is then used by the program for an internal operation, allowing the attacker to subvert the program. Various forms of buffer overflow attacks (stack and heap smashing, jump into libc, etc.) operate in this fashion. Such attacks can be detected when the corrupted control data is about to be used by the program (i.e., after the attack has succeeded). In various embodiments, such control data (e.g., memory locations or registers that hold such data) that is about to be overwritten with “tainted” data, or data provided by the network (which is potentially malicious) can be detected.

10

In accordance with various embodiments, how data modifications propagate throughout program execution can be monitored by maintaining a memory bit for every byte or word in memory. This bit is set for a memory location when a machine instruction uses as input data that was provided as input to the program (e.g., was received over the network, and is thus possibly malicious) and produces output that is stored in this memory location. If a control instruction (such as a JUMP or CALL) uses as an argument a value in a memory location in which the bit is set (i.e., the memory location is “tainted”), the program or the supervisory code that monitors program behavior can recognize an anomaly and raises an exception.

Detecting corruption before it happens, rather than later (when the corrupted data is about to be used by a control instruction), makes it possible to stop an operation and to discard its results/output, without other collateral damage. Furthermore, in addition to simply retaining knowledge of what is control and what is non-control data, according to various embodiments, knowledge of which instructions in the monitored piece of code typically modify specific memory locations can also be retained. Therefore, it is possible to detect attacks that compromise data that are used by the program computation itself, and not just for the program control flow management.

According to various embodiments, the inputs to the instruction(s) that can fail (or that can be exploited in an attack) and the outputs (results) of such instructions can be correlated with the inputs to the program at large. Inputs to an instruction are registers or locations in memory that contain values that may have been derived (in full or partially) by the input to the program. By computing a probability distribution model on the program input, alternate inputs may be chosen to give to the instruction or the function (“input rewriting” or “input modification”) when an imminent failure is detected, thereby allowing the program to “sidestep” the failure. However, because doing so may still cause the program to fail, according to various embodiments, micro-speculation (e.g., as implemented by STEM) can optionally be used to verify the effect of taking this course of action. A recovery technique (with different input values or error virtualization, for example) can then be used. Alternatively, for example, the output of the instruction may be caused to be a value/result that is typically seen when executing the program (“output overloading”).

In both cases (input modification or output overloading), the values to use may be selected based on several different criteria, including but not limited to one or more of the following: the similarity of the program input that caused failure to other inputs that have not caused a failure; the most frequently seen input or output value for that instruction, based on contextual information (e.g., when particular sequence of functions are in the program call stack); and most frequently seen input or output value for that instruction across all executions of the instruction (in all contexts seen). For example, if a particular DIVIDE instruction is detected in a function that uses a denominator value of zero, which would cause a process exception, and subsequently program failure, the DIVIDE instruction can be executed with a different denominator (e.g., based on how similar the program input is to other program inputs seen in the past, and the denominator values that these executions used). Alternatively, the DIVIDE instruction may be treated as though it had given a particular division result. The program may then be allowed to continue executing, while its behavior is being monitored. Should a failure subsequently occur while still under monitoring, a different input or output value for the instruction can be used,

US 8,601,322 B2

11

for example, or a different repair technique can be used. According to various embodiments, if none of the above strategies is successful, the user or administrator may be notified, program execution may be terminated, a rollback to a known good state (ignoring the current program execution) may take place, and/or some other corrective action may be taken.

According to various embodiments, the techniques used to learn typical data can be implemented as designer choice. For example, if it is assumed that the data modeled is 32-bit words, a probability distribution of this range of values can be estimated by sampling from multiple executions of the program. Alternatively, various cluster-based analyses may partition the space of typical data into clusters that represent groups of similar/related data by some criteria. Vector Quantization techniques representing common and similar data based on some "similarity" measure or criteria may also be compiled and used to guide modeling.

FIG. 1 is a schematic diagram of an illustrative system 100 suitable for implementation of various embodiments. As illustrated in FIG. 1, system 100 may include one or more workstations 102. Workstations 102 can be local to each other or remote from each other, and can be connected by one or more communications links 104 to a communications network 106 that is linked via a communications link 108 to a server 110.

In system 100, server 110 may be any suitable server for executing the application, such as a processor, a computer, a data processing device, or a combination of such devices. Communications network 106 may be any suitable computer network including the Internet, an intranet, a wide-area network (WAN), a local-area network (LAN), a wireless network, a digital subscriber line (DSL) network, a frame relay network, an asynchronous transfer mode (ATM) network, a virtual private network (VPN), or any combination of any of the same. Communications links 104 and 108 may be any communications links suitable for communicating data between workstations 102 and server 110, such as network links, dial-up links, wireless links, hard-wired links, etc. Workstations 102 may be personal computers, laptop computers, mainframe computers, data displays, Internet browsers, personal digital assistants (PDAs), two-way pagers, wireless terminals, portable telephones, etc., or any combination of the same. Workstations 102 and server 110 may be located at any suitable location. In one embodiment, workstations 102 and server 110 may be located within an organization. Alternatively, workstations 102 and server 110 may be distributed between multiple organizations.

The server and one of the workstations, which are depicted in FIG. 1, are illustrated in more detail in FIG. 2. Referring to FIG. 2, workstation 102 may include digital processing device (such as a processor) 202, display 204, input device 206, and memory 208, which may be interconnected. In a preferred embodiment, memory 208 contains a storage device for storing a workstation program for controlling processor 202. Memory 208 may also contain an application for detecting and repairing application from faults according to various embodiments. In some embodiments, the application may be resident in the memory of workstation 102 or server 110.

Processor 202 may use the workstation program to present on display 204 the application and the data received through communication link 104 and commands and values transmitted by a user of workstation 102. It should also be noted that data received through communication link 104 or any other communications links may be received from any suitable source, such as web services. Input device 206 may be a

12

computer keyboard, a cursor-controller, a dial, a switchbank, lever, or any other suitable input device as would be used by a designer of input systems or process control systems.

Server 110 may include processor 220, display 222, input device 224, and memory 226, which may be interconnected. In some embodiments, memory 226 contains a storage device for storing data received through communication link 108 or through other links, and also receives commands and values transmitted by one or more users. The storage device can further contain a server program for controlling processor 220.

In accordance with some embodiments, a self-healing system that allows an application to automatically recover from software failures and attacks is provided. By selectively emulating at least a portion or all of the application's code when the system detects that a fault has occurred, the system surrounds the detected fault to validate the operands to machine instructions, as appropriate for the type of fault. The system emulates that portion of the application's code with a fix and updates the application. This increases service availability in the presence of general software bugs, software failures, attacks.

Turning to FIGS. 3 and 4, simplified flowcharts illustrating various steps performed in detecting faults in an application and fixing the application in accordance with some embodiments are provided. These are generalized flow charts. It will be understood that the steps shown in FIGS. 3 and 4 may be performed in any suitable order, some may be deleted, and others added.

Generally, process 300 begins by detecting various types of failures in one or more applications at 310. In some embodiments, detecting for failures may include monitoring the one or more applications for failures, e.g., by using an anomaly detector as described herein. In some embodiments, the monitoring or detecting of failures may be performed using one or more sensors at 310. Failures include programming errors, exceptions, software faults (e.g., illegal memory accesses, division by zero, buffer overflow attacks, time-of-check-to-time-of-use (TOCTTOU) violations, etc.), threats (e.g., computer viruses, worms, trojans, hackers, key recovery attacks, malicious executables, probes, etc.), and any other suitable fault that may cause abnormal application termination or adversely affect the one or more applications.

Any suitable sensors may be used to detect failures or monitor the one or more applications. For example, in some embodiments, anomaly detectors as described herein can be used.

At 320, feedback from the sensors may be used to predict which parts of a given application's code may be vulnerable to a particular class of attack (e.g., remotely exploitable buffer overflows). In some embodiments, the sensors may also detect that a fault has occurred. Upon predicting that a fault may occur or detecting that a fault has occurred, the portion of the application's code having the faulty instruction or vulnerable function can be isolated, thereby localizing predicted faults at 330.

Alternatively, as shown and discussed in FIG. 4, the one or more sensor may monitor the application until it is caused to abnormally terminate. The system may detect that a fault has occurred, thereby causing the actual application to terminate. As shown in FIG. 4, at 410, the system forces a misbehaving application to abort. In response to the application terminating, the system generates a core dump file or produces other failure-related information, at 420. The core dump file may include, for example, the type of failure and the stack trace when that failure occurred. Based at least in part on the core dump file, the system isolates the portion of the application's

US 8,601,322 B2

13

code that contains the faulty instruction at 430. Using the core dump file, the system may apply selective emulation to the isolated portion or slice of the application. For example, the system may start with the top-most function in the stack trace.

Referring back to FIG. 3, in some embodiments, the system may generate an instrumented version of the application (340). For example, an instrumented version of the application may be a copy of a portion of the application's code or all of the application's code. The system may observe instrumented portions of the application. These portions of the application may be selected based on vulnerability to a particular class of attack. The instrumented application may be executed on the server that is currently running the one or more applications, a separate server, a workstation, or any other suitable device.

Isolating a portion of the application's code and using the emulator on the portion allows the system to reduce and/or minimize the performance impact on the immunized application. However, while this embodiment isolates a portion or a slice of the application's code, the entire application may also be emulated. The emulator may be implemented completely in software, or may take advantage of hardware features of the system processor or architecture, or other facilities offered by the operating system to otherwise reduce and/or minimize the performance impact of monitoring and emulation, and to improve accuracy and effectiveness in handling failures.

An attempt to exploit such a vulnerability exposes the attack or input vector and other related information (e.g., attacked buffer, vulnerable function, stack trace, etc.). The attack or input vector and other related information can then be used to construct an emulator-based vaccine or a fix that implements array bounds checking at the machine-instruction level at 350, or other fixes as appropriate for the detected type of failure. The vaccine can then be tested in the instrumented application using an instruction-level emulator (e.g., libtasvm x86 emulator, STEM x86 emulator, etc.) to determine whether the fault was fixed and whether any other functionality (e.g., critical functionality) has been impacted by the fix.

By continuously testing various vaccines using the instruction-level emulator, the system can verify whether the specific fault has been repaired by running the instrumented application against the event sequence (e.g., input vectors) that caused the specific fault. For example, to verify the effectiveness of a fix, the application may be restarted in a test environment or a sandbox with the instrumentation enabled, and is supplied with the one or more input vectors that caused the failure. A sandbox generally creates an environment in which there are strict limitations on which system resources the instrumented application or a function of the application may request or access.

At 360, the instruction-level emulator can be selectively invoked for segments of the application's code, thereby allowing the system to mix emulated and non-emulated code within the same code execution. The emulator may be used to, for example, detect and/or monitor for a specific type of failure prior to executing the instruction, record memory modifications during the execution of the instruction (e.g., global variables, library-internal state, libc standard I/O structures, etc.) and the original values, revert the memory stack to its original state, and simulate an error return from a function of the application. That is, upon entering the vulnerable section of the application's code, the instruction-level emulator can capture and store the program state and processes all instructions, including function calls, inside the area designated for emulation. When the program counter

14

references the first instruction outside the bounds of emulation, the virtual processor copies its internal state back to the device processor registers. While registers are updated, memory updates are also applied through the execution of the emulation. The program, unaware of the instructions executed by the virtual processor, continues normal execution on the actual processor.

In some embodiments, the instruction-level emulator may be linked with the application in advance. Alternatively, in response to a detected failure, the instruction-level emulator may be compiled in the code. In another suitable embodiment, the instruction-level emulator may be invoked in a manner similar to a modern debugger when a particular program instruction is executed. This can take advantage of breakpoint registers and/or other program debugging facilities that the system processor and architecture possess, or it can be a pure-software approach.

The use of an emulator allows the system to detect and/or monitor a wide array of software failures, such as illegal memory dereferences, buffer overflows, and buffer underflows, and more generic faults, such as divisions by zero. The emulator checks the operands of the instructions it is about to emulate using, at least partially, the vector and related information provided by the one or more sensors that detected the fault. For example, in the case of a division by zero, the emulator checks the value of the operand to the div instruction. In another example, in the case of illegal memory dereferencing, the emulator verifies whether the source and destination address of any memory access (or the program counter for instruction fetches) points to a page that is mapped to the process address space using the mincore() system call, or the appropriate facilities provided by the operating system. In yet another example, in the case of buffer overflow detection, the memory surrounding the vulnerable buffer, as identified by the one or more sensors, is padded by one byte. The emulator then watches for memory writes to these memory locations. This may require source code availability so as to insert particular variables (e.g., canary variables that launch themselves periodically and perform some typical user transaction to enable transaction-latency evaluation around the clock). The emulator can thus prevent the overflow before it overwrites the remaining locations in the memory stack and recovers the execution. Other approaches for detecting these failures may be incorporated in the system in a modular way, without impacting the high-level operation and characteristics of the system.

For example, the instruction-level emulator may be implemented as a statically-linked C library that defines special tags (e.g., a combination of macros and function calls) that mark the beginning and the end of selective emulation. An example of the tags that are placed around a segment of the application's code for emulation by the instruction-level emulator is shown in FIG. 5. As shown in FIG. 5, the C macro emulate_init() moves the program state (general, segment, eflags, and FPU registers) into an emulator-accessible global data structure to capture state immediately before the emulator takes control. The data structure can be used to initialize the virtual registers. emulate_begin() obtains the memory location of the first instruction following the call to itself. The instruction address may be the same as the return address and can be found in the activation record of emulate_begin(), four bytes above its base stack pointer. The fetch/decode/execute/retire cycle of instructions can continue until either emulate_end() is reached or when the emulator detects that control is returning to the parent function. If the emulator does not encounter an error during its execution, the emulator's instruction pointer references the emulate_term() macro at

US 8,601,322 B2

15

completion. To enable the instrumented application to continue execution at this address, the return address of the emulate_begin() activation record can be replaced with the current value of the instruction pointer. By executing emulate_term(), the emulator's environment can be copied to the program registers and execution continues under normal conditions.

Although the emulator can be linked with the vulnerable application when the source code of the vulnerable application is available, in some embodiments the processor's programmable breakpoint register can be used to invoke the emulator without the running process even being able to detect that it is now running under an emulator.

In addition to monitoring for failures prior to executing instructions and reverting memory changes made by a particular function when a failure occurs (e.g., by having the emulator store memory modifications made during its execution), the emulator can also simulate an error return from the function. For example, some embodiments may generate a map between a set of errors that may occur during an application's execution and a limited set of errors that are explicitly handled by the application's code (sometimes referred to herein as "error virtualization"). As described below, the error virtualization features may be based on heuristics. However, any suitable approach for determining the return values for a function may be used. For example, aggressive source code analysis techniques to determine the return values that are appropriate for a function may be used. In another example, portions of code of specific functions can be marked as fail-safe and a specific value may be returned when an error return is forced (e.g., for code that checks user permissions). In yet another example, the error value returned for a function that has failed can be determined using information provided by a programmer, system administrator, or any other suitable user.

These error virtualization features allow an application to continue execution even though a boundary condition that was not originally predicted by a programmer allowed a fault to occur. In particular, error virtualization features allows for the application's code to be retrofitted with an exception catching mechanism, for faults that were unanticipated by the programmer. It should be noted that error virtualization is different from traditional exception handling as implemented by some programming languages, where the programmer must deliberately create exceptions in the program code and also add code to handle these exceptions. Under error virtualization, failures and exceptions that were unanticipated by, for example, the programmer can be caught, and existing application code can be used to handle them. In some embodiments, error virtualization can be implemented through the instruction-level emulator. Alternatively, error virtualization may be implemented through additional source code that is inserted in the application's source code directly. This insertion of such additional source code can be performed automatically, following the detection of a failure or following the prediction of a failure as described above, or it may be done under the direction of a programmer, system operator, or other suitable user having access to the application's source code.

Using error virtualization, when an exception occurs during the emulation or if the system detects that a fault has occurred, the system may return the program state to its original settings and force an error return from the currently executing function. To determine the appropriate error value, the system analyzes the declared type of function. In some embodiments, the system may analyze the declared type of function using, for example, a TXL script. Generally, TXL is a hybrid function and rule-based language that may be used

16

for performing source-to-source transformation and for rapidly prototyping new languages and language processors. Based on the declared type of function, the system determines the appropriate error value and places it in the stack frame of the returning function. The appropriate error value may be determined based at least in part on heuristics. For example, if the return type is an int, a value of -1 is returned. If the return type is an unsigned int, the system returns a 0. If the function returns a pointer, the system determines whether the returned pointer is further dereferenced by the parent function. If the returned pointer is further dereferenced, the system expands the scope of the emulation to include the parent function. In another example, the return error code may be determined using information embedded in the source code of the application, or through additional information provided to the system by the application programmer, system administrator or third party.

In some embodiments, the emulate_end() is located and the emulation terminates. Because the emulator saved the state of the application before starting and kept track of memory modification during the application's execution, the system is capable of reversing any memory changes made by the code function inside which the fault occurred by returning it to its original setting, thereby nullifying the effect of the instructions processed through emulation. That is, the emulated portion of the code is sliced off and the execution of the code along with its side effects in terms of changes to memory have been rolled back.

For example, the emulator may not be able to perform system calls directly without kernel-level permissions. Therefore, when the emulator decodes an interruption with an intermediate value of 0x80, the emulator releases control to the kernel. However, before the kernel executes the system call, the emulator can back-up the real registers and replace them with its own values. An interrupt 0x80 can be issued by the emulator and the kernel processes the system call. Once control returns to the emulator, the emulator can update its registers and restore the original values in the application's registers.

If the instrumented application does not crash after the forced return, the system has successfully found a vaccine for the specific fault, which may be used on the actual application running on the server. At 370, the system can then update the application based at least in part on the emulation.

In accordance with some embodiments, artificial diversity features may be provided to mitigate the security risks of software monoculture.

FIG. 6 is a simplified flowchart illustrating the various steps performed in using an application community to monitor an application for faults and repair the application in accordance with some embodiments. This is a generalized flow chart. It will be understood that the steps shown in FIG. 6 may be performed in any suitable order, some may be deleted, and others added.

Generally, the system may divide an application's code into portions of code at 610. Each portion or slice of the application's code may, for example, be assigned to one of the members of the application community (e.g., workstation, server, etc.). Each member of the application community may monitor the portion of the code for various types of failures at 620. As described previously, failures include programming errors, exceptions, software faults (e.g., illegal memory accesses, division by zero, buffer overflow attacks, TOCTTOU violations, etc.), threats (e.g., computer viruses, worms, trojans, hackers, key recovery attacks, malicious executables,

US 8,601,322 B2

17

probes, etc.), and any other suitable fault that may cause abnormal application termination or adversely affect the one or more applications.

For example, the system may divide the portions of code based on the size of the application and the number of members in the application community (i.e., size of the application/members in the application community). Alternatively, the system may divide the portions of code based on the amount of available memory in each of the members of the application community. Any suitable approach for determining how to divide up the application's code may also be used. Some suitable approaches are described hereinafter.

For example, the system may examine the total work in the application community, W , by examining the cost of executing discrete slices of the application's code. Assuming a set of functions, F , that comprise an application's callgraph, the i^{th} member of F is denoted as f_i . The cost of executing each f_i is a function of the amount of computation present in f_i (i.e., x_i) and the amount of risk in f_i (i.e., v_i). The calculation of x_i can be driven by at least two metrics: o_i , the number of machine instructions executed as part of f_i , and t_i , the amount of time spent executing f_i . Both o_i and t_i may vary as a function of time or application workload according to the application's internal logic. For example, an application may perform logging or cleanup duties after the application passes a threshold number of requests.

In some embodiments, a cost function may be provided in two phases. The first phase calculates the cost due to the amount of computation for each f_i . The second phase normalizes this cost and applies the risk factor v_i to determine the final cost of each f_i and the total amount of work in the system. For example, let

$$T = \sum_{i=1}^N x_i$$

If $C(f_i, x_i) = x_i/T * 100$, each cost may be normalized by grouping a subset of F to represent one unit of work.

In some embodiments, the system may account for the measure of a function's vulnerability. For example, the system treats v_i as a discrete variable with a value of α , where α takes on a range of values according to the amount of risk such that:

$$v_i = \begin{cases} \alpha & \text{(if } f_i \text{ is vulnerable)} \\ 1 & \text{(if } f_i \text{ is not vulnerable)} \end{cases}$$

Given v_i for each function, the system may determine the total amount of work in the system and the total number of members needed for monitoring:

$$W = N_{\text{vuln}} = \sum_{i=1}^n v_i * r_i$$

After the system (e.g., a controller) or after each application community member has calculated the amount of work in the system, work units can be distributed. In one example, a central controller or one of the workstations may assign each node approximately W/N work units. In another suitable example, each member of the application community may determine its own work set. Each member may iterate through the list of work units flipping a coin that is weighted with the value $v_i * r_i$. Therefore, if the result of the flip is "true," then the member adds that work unit to its work set.

18

Alternatively, the system may generate a list having $n * W$ slots. Each function can be represented by a number of entries on the list (e.g., $v_i * r_i$). Every member of the application community can iterate through the list, for example, by randomly selecting true or false. If true, the application community member monitors the function of the application for a given time slice. Because heavily weighted functions have more entries in the list, a greater number of users may be assigned to cover the application. The member may stop when its total work reaches W/N . Such an approach offers statistical coverage of the application.

In some embodiments, a distributed bidding approach may be used to distribute the workload of monitoring and repairing an application. Each node in the callgraph G has a weight $v_i * r_i$. Some subset of the nodes in F is assigned to each application community member such that each member does no more work than W/N work. The threshold can be relaxed to be within some range ϵ of W/N , where ϵ is a measure of system fairness. Upon calculating the globally fair amount of work W/N , each application community member may adjust its workload by bargaining with other members using a distributed bidding approach.

Two considerations impact the assignment of work units to application community members. First, the system can allocate work units with higher weights, as these work units likely have a heavier weight due to a high v_i . Even if the weight is derived solely from the performance cost, assigning more members to the work units with higher weights is beneficial because these members can round-robin the monitoring task so that any one member does not have to assume the full cost. Second, in some situations, $v_i * r_i$ may be greater than the average amount of work, W/N . Achieving fairness means that $v_i * r_i$ defines the quantity of application community members that is assigned to it and the sum of these quantities defines the minimum number of members in the application community.

In some embodiments, each application community member calculates a table. An example of such a table is shown in FIG. 7. Upon generating the table, application community members may place bids to adjust each of their respective workloads. For example, the system may use tokens for bidding. Tokens may map directly to the number of time quanta that an application community member is responsible for monitoring a work unit or a function of an application. The system ensures that each node does not accumulate more than the total number of tokens allowed by the choice of ϵ .

If an application community member monitors more than its share, then the system has increased coverage and can ensure that faults are detected as quickly as possible. As shown in 630 and 640, each application community member may predict that a fault may occur in the assigned portion of code or may detect that a fault has occurred causing the application to abort, where the assigned portion of the code was the source of the fault. As faults are detected, applications members may each proactively monitor assigned portions of code containing the fault to prevent the application from further failures. As discussed previously, the application community member may isolate the portion of the code that caused the fault and use the emulator to test vaccines or fixes. At 650, the application community member that detects or predicts the fault may notify the other application community members. Other application members that have succumbed to the fault may be restarted with the protection mechanisms or fixes generated by the application member that detected the fault.

Assuming a uniform random distribution of new faults across the application community members, the probability of a fault happening at a member, k , is: $P(\text{fault}) = 1/N$. Thus,

US 8,601,322 B2

19

the probability of k detecting a new fault is the probability that the fault happens at k and that k detects the fault: $P(\text{fault at } k \wedge \text{detection}) = 1/N * k_i$, where k_i is the percentage of coverage at k. The probability of the application community detecting the fault is:

$$P(\text{AC detect}) = \sum_{i=1}^N \frac{1}{N} * k_i$$

As each k_i goes to 100%, the above-equation becomes

$$\sum_{i=1}^N \frac{1}{N}$$

or N/N , a probability of 1 that the fault is detected when it first occurs.

It will also be understood that various embodiments may be presented in terms of program procedures executed on a computer or network of computers.

A procedure is here, and generally, conceived to be a self-consistent sequence of steps leading to a desired result. These steps are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared and otherwise manipulated. It proves convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like. However, all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities.

Further, the manipulations performed are often referred to in terms, such as adding or comparing, which are commonly associated with mental operations performed by a human operator. No such capability of a human operator is necessary, or desirable in many cases, in any of the operations described herein in connection with various embodiments; the operations are machine operations. Useful machines for performing the operation of various embodiments include general purpose digital computers or similar devices.

Some embodiments also provide apparatuses for performing these operations. These apparatuses may be specially constructed for the required purpose or it may comprise a general purpose computer as selectively activated or reconfigured by a computer program stored in the computer. The procedures presented herein are not inherently related to a particular computer or other apparatus. Various general purpose machines may be used with programs written in accordance with the teachings herein, or it may prove more convenient to construct more specialized apparatus to perform the described method. The required structure for a variety of these machines will appear from the description given.

Some embodiments may include a general purpose computer, or a specially programmed special purpose computer. The user may interact with the system via e.g., a personal computer or over PDA, e.g., the Internet an Intranet, etc. Either of these may be implemented as a distributed computer system rather than a single computer. Similarly, the communications link may be a dedicated link, a modem over a POTS line, the Internet and/or any other method of communicating between computers and/or users. Moreover, the processing could be controlled by a software program on one or more

20

computer systems or processors, or could even be partially or wholly implemented in hardware.

Although a single computer may be used, systems according to one or more embodiments are optionally suitably equipped with a multitude or combination of processors or storage devices. For example, the computer may be replaced by, or combined with, any suitable processing system operative in accordance with the concepts of various embodiments, including sophisticated calculators, hand held, laptop/notebook, mini, mainframe and super computers, as well as processing system network combinations of the same. Further, portions of the system may be provided in any appropriate electronic format, including, for example, provided over a communication line as electronic signals, provided on CD and/or DVD, provided on optical disk memory, etc.

Any presently available or future developed computer software language and/or hardware components can be employed in such embodiments. For example, at least some of the functionality mentioned above could be implemented using Visual Basic, C, C++ or any assembly language appropriate in view of the processor being used. It could also be written in an object oriented and/or interpretive environment such as Java and transported to multiple destinations to various users.

Other embodiments, extensions, and modifications of the ideas presented above are comprehended and within the reach of one skilled in the field upon reviewing the present disclosure. Accordingly, the scope of the present invention in its various aspects is not to be limited by the examples and embodiments presented above. The individual aspects of the present invention, and the entirety of the invention are to be regarded so as to allow for modifications and future developments within the scope of the present disclosure. For example, the set of features, or a subset of the features, described above may be used in any suitable combination. The present invention is limited only by the claims that follow.

What is claimed is:

1. A method for detecting anomalous program executions, comprising:
 - executing at least a portion of a program in an emulator;
 - comparing a function call made in the emulator to a model of function calls for the at least a portion of the program, wherein the model is a combined model created from at least two models created at different times; and
 - identifying the function call as anomalous based on the comparison.
2. A method for detecting anomalous program executions, comprising:
 - executing at least a portion of a program in an emulator;
 - comparing a function call made in the emulator to a model of function calls for the at least a portion of the program, wherein the model is a combined model created from at least two models created using different computers; and
 - identifying the function call as anomalous based on the comparison.
3. The method of claim 1, further comprising modifying the function call so that the function call becomes non-anomalous.
4. The method of claim 1, further comprising generating a virtualized error in response to the function call being identified as being anomalous.
5. The method of claim 1, wherein the comparing compares the function call name and arguments to the model.
6. The method of claim 1, wherein the model reflects normal activity of the at least a portion of the program.
7. The method of claim 1, wherein the model reflects attacks against the at least a portion of the program.

US 8,601,322 B2

21

22

8. The method of claim 1, further comprising randomly selecting at least a portion of the model to be used in the comparison from a plurality of different models relating to the program.

9. The method of claim 1, further comprising notifying another computer of the anomalous function call upon the function call being identified as anomalous.

10. A non-transitory computer-readable medium containing computer-executable instructions that, when executed by a processor, cause the processor to perform a method for detecting anomalous program executions, comprising:

- executing at least a portion of a program in an emulator;
- comparing a function call made in the emulator to a model of function calls for the at least a portion of the program, wherein the model is a combined model created from at least two models created at different times; and
- identifying the function call as anomalous based on the comparison.

11. A non-transitory computer-readable medium containing computer-executable instructions that, when executed by a processor, cause the processor to perform a method for detecting anomalous program executions, comprising:

- executing at least a portion of a program in an emulator;
- comparing a function call made in the emulator to a model of function calls for the at least a portion of the program, wherein the model is a combined model created from at least two models created using different computers; and
- identifying the function call as anomalous based on the comparison.

12. The non-transitory computer-readable medium of claim 10, wherein the method further comprises modifying the function call so that the function call becomes non-anomalous.

13. The non-transitory computer-readable medium of claim 10, wherein the method further comprises generating a virtualized error in response to the function call being identified as being anomalous.

14. The non-transitory computer-readable medium of claim 10, wherein the comparing compares the function call name and arguments to the model.

15. The non-transitory computer-readable medium of claim 10, wherein the model reflects normal activity of the at least a portion of the program.

16. The non-transitory computer-readable medium of claim 10, wherein the model reflects attacks against the at least a portion of the program.

17. The non-transitory computer-readable medium of claim 10, wherein the method further comprises randomly

selecting at least a portion of the model to be used in the comparison from a plurality of different models relating to the program.

18. The non-transitory computer-readable medium of claim 10, wherein the method further comprises notifying another computer of the anomalous function call upon the function call being identified as anomalous.

19. A system for detecting anomalous program executions, comprising:

- a processor that:
 - executes at least a portion of a program in an emulator;
 - compares a function call made in the emulator to a model of function calls for the at least a portion of the program, wherein the model is a combined model created from at least two models created at different times; and
 - identifies the function call as anomalous based on the comparison.

20. The system of claim 19, wherein the method further comprising modifying the function call so that the function call becomes non-anomalous.

21. The system of claim 19, wherein the method further comprising generating a virtualized error in response to the function call being identified as being anomalous.

22. The system of claim 19, wherein the comparing compares the function call name and arguments to the model.

23. The system of claim 19, wherein the model reflects normal activity of the at least a portion of the program.

24. The system of claim 19, wherein the model reflects attacks against the at least a portion of the program.

25. The system of claim 19, wherein the method further comprises randomly selecting at least a portion of the model to be used in the comparison from a plurality of different models relating to the program.

26. The system of claim 19, wherein the method further comprises notifying another computer of the anomalous function call upon the function call being identified as anomalous.

27. A system for detecting anomalous program executions, comprising:

- a processor that:
 - executes at least a portion of a program in an emulator;
 - compares a function call made in the emulator to a model of function calls for the at least a portion of the program, wherein the model is a combined model created from at least two models created using different computers; and
 - identifies the function call as anomalous based on the comparison.

* * * * *

FORM 30. Certificate of Service

UNITED STATES COURT OF APPEALS
FOR THE FEDERAL CIRCUIT

CERTIFICATE OF SERVICE

I certify that I served a copy on counsel of record on
by:

- US mail
- Fax
- Hand
- Electronic Means
(by email or CM/ECF)

Name of Counsel

Signature of Counsel

Law Firm

Address

City, State, ZIP

Telephone Number

FAX Number

E-mail Address

NOTE: For attorneys filing documents electronically, the name of the filer under whose log-in and password a document is submitted must be preceded by an "/s/" and typed in the space where the signature would otherwise appear. Graphic and other electronic signatures are discouraged.

FORM 19. Certificate of Compliance With Rule 32(a)

**CERTIFICATE OF COMPLIANCE WITH TYPE-VOLUME LIMITATION,
TYPEFACE REQUIREMENTS, AND TYPE STYLE REQUIREMENTS**

1. This brief complies with the type-volume limitation of [Federal Rule of Appellate Procedure 32\(a\)\(7\)\(B\)](#) or [Federal Rule of Appellate Procedure 28.1\(e\)](#).

- The brief contains [13,978] words, excluding the parts of the brief exempted by [Federal Rule of Appellate Procedure 32\(a\)\(7\)\(B\)\(iii\)](#), or
- The brief uses a monospaced typeface and contains [*state the number of*] lines of text, excluding the parts of the brief exempted by [Federal Rule of Appellate Procedure 32\(a\)\(7\)\(B\)\(iii\)](#).

2. This brief complies with the typeface requirements of [Federal Rule of Appellate Procedure 32\(a\)\(5\)](#) or [Federal Rule of Appellate Procedure 28.1\(e\)](#) and the type style requirements of [Federal Rule of Appellate Procedure 32\(a\)\(6\)](#).

- The brief has been prepared in a proportionally spaced typeface using [*Microsoft Word 2010*] in [*14 Point Times New Roman*], or
- The brief has been prepared in a monospaced typeface using [*state name and version of word processing program*] with [*state number of characters per inch and name of type style*].

/s/ Gavin Snyder

(Signature of Attorney)

Gavin Snyder

(Name of Attorney)

Attorney for Appellant

(State whether representing appellant, appellee, etc.)

1/20/2015

(Date)