UNITED STATES PATENT AND TRADEMARK OFFICE

BEFORE THE PATENT TRIAL AND APPEAL BOARD

BLUE COAT SYSTEMS, INC.,
Petitioner,

v.

FINJAN, INC.,
Patent Owner.

Patent No. 8,225,408

DECLARATION OF AZER BESTAVROS, PH.D.

TABLE OF CONTENTS

I, Azer Bestavros, declare as follows:

I. QUALIFICATIONS

1. I am a Professor of Computer Science at Boston University, whose faculty I joined in 1991. I chaired the Computer Science Department from 2000 to 2007, overseeing a period of significant growth, culminating in the Chronicle of Higher Education's ranking of the Department as seventh in the U.S. in terms of scholarly productivity.

2. I am the Founding Director of the BU Hariri Institute for Computing at Boston University, which was set up in 2010 to "create and sustain a community of scholars who believe in the transformative potential of computational perspectives in research and education." I am also the co-Chair of the Council on Educational Technology & Learning Innovation, which was set up in 2012 to develop Boston University's strategy as it relates to leveraging on-line technology in on-campus, residential programs.

3. In addition to my academic responsibilities at Boston University, over the years I have taken significant regional and national research leadership responsibilities. This includes: serving since 2010 as co-chair for Research, Education, and Outreach of the Massachusetts Green High-Performance Computing Center – a consortium of the five major research institutions in the Commonwealth of

Massachusetts (Boston University, Harvard University, MIT, Northeastern University, and the University of Massachusetts); serving since 2013 as board member of the Cloud Computing Caucus, a non-profit, non-partisan coalition of industry and key government stakeholders, focused on raising awareness and educating lawmakers and the public on issues associated with cloud computing; and serving from 2007 to 2012 as the elected chair of the IEEE Computer Society Technical Committee on the Internet.

4. I also organized Computer Science leadership workshops at the Computing Research Association (CRA) Snowbird conferences on models for university-led technology transfer and incubation in 2000, and on models for publications in CS in 2006. In addition, I organized a number of meetings to develop research agendas and recommendations to government agencies, including the PI meeting of the CRI program at NSF, and the HCCS committee of the National Coordination Office for Networking and Information Technology. Most recently, I served as chair of the 2014 Committee of Visitors tasked to review the performance of the CNS Division of the CISE Directorate of the National Science Foundation.

5. I am a senior member of the Association for Computer Machinery (ACM) and a senior member of the Computer Society of the Institute of Electrical and Electronics Engineers (IEEE), among other professional societies and organizations.

Within these organizations, I served as general chair, PC chair or PC member of most flagship technical conferences in networking, real-time systems, and databases, including ACM Sigmetrics, IEEE Infocom, ACM PODC, IEEE ICNP, ACM MMSys, IEEE HotWeb, IEEE RTSS, IEEE RTAS, ICDCS, ACM LCTES, IEEE ICDE, ACM Sigmod, and VLDB. I co-organized formative workshops that led to ACM SIGPLAN LCTES and ACM SIGCOMM IMC. I have also served on the editorial board of major journals and periodicals, currently including IEEE Internet Computing and Communications of the ACM.

6. Prior to joining the faculty at Boston University, from June 1988 to September 1991, I was a Research Fellow, Teaching Fellow, and Research Assistant at Harvard University. From September 1985 to June 1987, I was a Research Assistant, Teaching Assistant, and Instructor at Alexandria University (Egypt).

7. I obtained my Ph.D. in Computer Science in 1992 from Harvard University under Thomas E. Cheatham, one of the "roots" of the academic genealogy of applied computer scientists. I also hold a Master of Science degree in Computer Science from Harvard University, which I obtained in 1988; a Master of Science degree in Computer Science and Automatic Control from Alexandria University, which I obtained in 1987; and a Bachelor of Science degree in Computer Engineering from Alexandria University, which I obtained in 1984.

8. I have studied, taught, practiced, and conducted research in Computer Science and Computer Engineering for more than 30 years. My expertise is in the broad fields of computer networking, distributed systems, and real-time computing, with significant experience in Web content caching and distribution systems, scalable Internet services, cloud computing, Internet architecture and networking protocols, among others.

9. I have extensive consulting and industrial research experience, including past and current engagements with a number of technology firms, including BBN Technologies, Sycamore Networks, NetApp, Microsoft, Verizon Labs, Macromedia, Allaire, Bowne, SUTI Technologies, and AT&T Bell Labs. I have consulted and served on the technical advisory board of many companies, and I have been retained by a number of law firms as a consultant on intellectual property issues related to Internet technologies and applications.

10. My curricular development efforts include my CS-350 course, which I developed and have taught since 1998. Through a rigorous treatment of the invariant concepts underlying computing systems design, CS-350 familiarizes students with canonical problems that reoccur in software systems, including operating systems, networks, databases, and distributed systems, and provides students with a set of classical algorithms and basic performance evaluation techniques for tackling such

problems. More recently, I have spearheaded a team effort to develop a set of courses for non-majors that can be used to introduce elements of mathematical abstraction, quantitative and methodical thinking, as utilized in mathematics, statistics, and computer science, with an emphasis on their relevance in our daily lives as reflected in widely used Internet and Web technologies and applications. In addition to these courses, I have taught undergraduate courses and graduate seminars on large-scale Internet systems, sensor networks, computer architecture, and real-time systems, and have guest-lectured in Sociology on issues related to Technology, Society and Public Policy.

11. Over the years, my contributions in research, teaching, and service have been recognized by a number of awards, including multiple best-paper awards from IEEE and ACM conferences, multiple distinguished ACM and IEEE service awards, and being selected multiple times as a distinguished speaker of the IEEE Computer Society (last time in 2010). In 2010, I received the United Methodist Scholar Teacher Award in recognition of "outstanding dedication and contributions t3o the learning arts and to the institution" at Boston University, and the ACM Sigmetrics Inaugural Test of Time Award for research results "whose impact is still felt 10-15 years after its initial publication."

12. A copy of my Curriculum Vitae, attached as EX10XX, contains further details on my education, experience, publications, patents, and other qualifications to render an expert opinion in this matter.

II. SCOPE OF WORK

13. I understand that a petition is being filed with the United States Patent and Trademark Office for *Inter Partes* Review of U.S. Patent No. 8,225,408 to Rubin et al. ("the '408 Patent," attached as EX1001), entitled "Method and System for Adaptive Rule-Based Content Scanners."

14. I have been retained by Blue Coat Systems, Inc. ("Blue Coat") to offer an expert opinion on the patentability of the claims of the '408 patent, as well as several other patents assigned to Finjan. I receive $550 per hour for my services. No part of my compensation is dependent on my opinions or on the outcome of this proceeding. I have previously testified for Blue Coat as an expert on the issue of noninfringement in case 13-cv-03999-BLF, which involved different patents. I do not have any other current or past affiliation as an expert witness or consultant with Blue Coat.

15. I have been specifically asked to provide my opinions on claims 2, 8, 11, 24-29, and 31-34 of the '408 patent. In connection with this analysis, I have reviewed the '408 patent and its file history. I have also reviewed and considered various other documents in arriving at my opinions, and may cite to them in this declaration. For

convenience, the information considered in arriving at my opinions is listed in Appendix A.

III. OVERVIEW OF THE '408 PATENT

16. The '408 patent is directed to protecting computers against potentially malicious programs using programming language-specific sets of rules and a "parse tree" data structure. EX1001 at Title, Abstract.

17. The '408 patent describes scanning an incoming stream of computer code by creating tokens, generating a parse tree using patterns in those tokens, and identifying patterns of tokens in the parse tree as potential exploits. *See id*. Patterns are identified using "parser rules" and "analyzer rules" specific to one of multiple programming languages. Accordingly, the challenged claims recite "multi-lingual" methods that determine a specific computer language from a plurality of languages and use a "scanner" specific to that language to scan the incoming stream of computer code.

18. The '408 patent was filed in August 2004 and was subject to a first office action rejecting and/or objecting to all claims in July 2008. Over the next four years, the applicant amended the claims in response to eight separate rejections. In 2012, the applicant substantially re-wrote the claims, adding additional limitations to the independent claims, including (1) multi-language processing capability and

(2) temporal restrictions regarding when the claimed system receives a data stream, builds a parse tree, and detects viruses within the parse tree. *See* EX1004 at 40-53. The claims were allowed following those additions. *See* EX1004 at 69-71.

19. I understand that the priority date for a particular claim is based in part on when in a chain of related patents the written description that supports that claim first appeared. The '408 patent was filed on August 30, 2004, as a continuation-in-part of Application No. 09/539,667 (now U.S. Patent No.6,804,780), filed on March 30, 2000, which is itself a continuation of Application No. 08/964,388 (now U.S. Patent No. 6,092,194), filed on November 6, 1997.

20. Although filed as a continuation-in-part, the '408 patent shares almost nothing with the earlier-filed applications. For example, the '667 and '388 applications do not mention "tokens" or "parse trees," elements that appear throughout all claims of the '408 patent. *See* EX1005, EX1006. The earliest specification that a person of skill in the art would recognize as providing a description of the subject matter of those claims was the application filed August 30, 2004, that later issued as the '408 patent. *See* EX1001. As such, the challenged claims are entitled to a priority date no earlier than August 30, 2004, the '408 patent's own filing date.

IV. LEGAL STANDARD

21. I understand that a claimed invention is not patentable under 35 U.S.C. § 103, for obviousness if the differences between the invention and the prior art are such that the subject matter as a whole would have been obvious at the time the invention was made to a person having ordinary skill in the art to which the subject matter pertains.

22. It is further my understanding that a determination of obviousness requires inquiries into: (1) the scope and contents of the art when the invention was made; (2) the differences between the art and the claims at issue; (3) the level of ordinary skill in the pertinent art when the invention was made; and, to the extent they exist, (4) secondary indicia of obviousness.

23. I understand that a claim can be found to be obvious if all the claimed elements were known in the prior art and one skilled in the art could have combined the elements as claimed by known methods with no change in their respective functions, and the combination would have yielded nothing more than predictable results to one of ordinary skill in the art.

24. I understand that hindsight must not be used when comparing the prior art to the invention for obviousness. Thus, a conclusion of obviousness must be firmly based on knowledge and skill of a person of ordinary skill in the art at the time the invention was made without the use of post-filing knowledge.

25. I understand that in order for a claimed invention to be considered obvious, there must be some rational underpinning for combining cited references as proposed.

26. I understand that obviousness may also be shown by demonstrating that it would have been obvious to modify what is taught in a single piece of prior art to create the patented invention. Obviousness may be shown by showing that it would have been obvious to combine the teachings of more than one item of prior art. In determining whether a piece of prior art could have been combined with other prior art or with other information within the knowledge of one of ordinary skill in the art, the following are examples of approaches and rationales that may be considered:

(a) Combining prior art elements according to known methods to yield predictable results;

(b) Simple substitution of one known element for another to obtain predictable results;

(c) Use of a known technique to improve similar devices (methods, or products) in the same way;

(d) Applying a known technique to a known device (method, or product) ready for improvement to yield predictable results;

(e) Applying a technique or approach that would have been "obvious to try" (choosing from a finite number of identified, predictable solutions, with a reasonable expectation of success);

(f) Known work in one field of endeavor may prompt variations of it for use in either the same field or a different one based on design incentives or other market forces if the variations would have been predictable to one of ordinary skill in the art; or

(g) Some teaching, suggestion, or motivation in the prior art that would have led one of ordinary skill to modify the prior art reference or to combine prior art reference teachings to arrive at the claimed invention.

V. OVERVIEW OF THE PRIOR ART

27. In my opinion, and as explained in further detail below, claims 2, 8, 11, and 24-28, and 30-34 of the '408 patent fail to identify anything new or significantly different from what was already known to individuals of skill in the field prior to the filing of the application that led to the '408 patent, including prior to August 30, 2004.

28. Below is an overview of certain of the main prior art references that I rely on for my opinion that claims 2, 8, 11, and 24-28, and 30-34 of the '408 are unpatentable: Chandnani, Kolawa, Knuth, Huang, and Walls.

A. Chandnani

29. U.S. Patent Appl. Pub. No. 2002/0073330 ("Chandnani," EX1007), titled "Detection of Polymorphic Script Language Viruses by Data Driven Lexical

Analysis," was filed on July 14, 2001. I understand that Chandnani is prior art under 35 U.S.C. § 102(b) because it was published on June 13, 2002, more than one year before the filing date of the '408 patent.

30. Chandnani teaches detecting polymorphic script language viruses using data-driven lexical analysis. EX1007 at [0002]. Like the '408 patent, Chandnani scans for polymorphic viruses—those that have slightly different code but the same malicious functionality—by converting a data stream into a stream of tokens and then searching for patterns that indicate the presence of potentially malicious programs. *Id.* at [0014]-[0020], [0056]-[0065]. Also like the '408 patent, Chandnani scans a continuous stream of data, and continues to receive upstream data while analyzing downstream data. *See, e.g., id*. at [0057] (describing the data stream as a series of characters), [0060] (describing a two-stage detection process), Fig. 2.

31. Although Chandnani may not expressly describe how tokens are parsed and analyzed, Chandnani's disclosure of parsing a stream into tokens and then storing those tokens suggests and implicitly teaches using a parse tree because a person of ordinary skill in the art understood that the obvious place to store those tokens was in a parse tree. *See* EX1007 at [0040]-[0046]; below ¶¶ 102-109.

32. In addition, using a parse tree to store portions of an incoming data stream was obvious, as illustrated by prior art such as Kolawa. Use of a parse tree data

structure to represent and analyze computer code was well-known by 2004. Numerous prior art references describe the use of parse trees for these purposes, including the following:

- EX1008 (Kolawa) at 3:8-12 ("The parse tree is searched for a match between such a node in the parse tree having a node type that matches such a node type in the set of node types for the selected quality rule."), 5:62-64 ("The quality of the source code 10 is checked on an individual parse tree basis.")
- EX1012 at 5:19-22 ("[I]nterceptor determines if the data retrieval request corresponds to at least one of the rules of the security policy, and identifies, via a parse tree, selectivity operators indicative of the allowable data items to be retrieved.")
- EX1013 at 5 ("The parser output is a full parse tree (a collection of nodes, each representing a piece of the SQL such as an operator, function, or value), which reflects all the SQL grammar.") ("[T]he firewall compares this parse tree with the rules you've devised.")
- EX1014 at 5:3-5 ("[P]arser 20 processes the suspect string 26 and suspect filed [sic] 27 on a line-by-line basis and generates a hierarchical parse tree, as is known in the art.")

- EX1015 at 14:25-28 ("Parser 296 identifies non-terminals and valid strings and creates a parse tree.")
- EX1016 at 13:34-36 ("[S]erver 102 converts the source-code instructions of the submitted query into a parse tree.")

B. Kolawa

33. U.S. Patent No. 5,860,011 ("Kolawa," EX1008), titled "Method and System for Automatically Checking Computer Source Code Quality Based on Rules," was filed on February 19, 1996. I understand that Kolawa is prior art under 35 U.S.C. § 102(b) because it issued on January 12, 1999, more than one year before the August 30, 2004 filing date of the '408 patent.

34. Kolawa teaches a method and system for rule-based evaluation of source code quality. EX1008 at 1:19-22. In particular, Kolawa discloses using a "conventional" lexical analyzer that scans code, groups it into tokens, and organizes the tokens using a parse tree:

> The source code 10 is read as input to a lexical analyzer/parser 11 which is conventional in the art. The lexical analyzer scans the source code 10 and groups the instructions into tokens. The parser performs the hierarchical analysis which groups the tokens into grammatical phrases that are represented by a parse tree 12.

Id. at 3:66-4:4; *see also id.* at Fig. 1. Kolawa then searches the parse tree to identify problematic code based on a set of rules. *See id.* at 4:48-59. Kolawa reports rule violations as error messages that describe the corresponding quality concern. *Id.* at 4:59-60. Kolawa discloses an embodiment that supports two different programming languages and notes that support for additional languages is also possible. *Id.* at 3:53-56.

C. Knuth

35. "On the Translation of Languages from Left to Right" ("Knuth," EX1009) was published in Information and Control in 1965. I understand that Knuth is prior art under 35 U.S.C. § 102(b) because it was published more than one year before the August 30, 2004 filing date of the '408 patent.

36. Knuth is a foundational paper describing the parsing of programming languages from left-to-right. EX1009 at Abstract. Knuth provides examples of parsing code and building parse trees using a shift-and-reduce process. EX1009 at 618-625, Tables I and II. In one example, detailed in Table I, Knuth describes the shift and reduce process: "'Shift' means 'perform the shift left operation' mentioned in step 2; 'reduce p' means 'perform the transformation (21) with production p.'" EX1009 at 620. Knuth also describes the basic parsing steps of recursively matching

patterns in strings and generating parent nodes attached to those patterns, thereby generating a parse tree. EX1009 at 609-610.

D. Huang

37. U.S. Patent No. 6,968,539 ("Huang," EX1010), titled "Methods and Apparatus for a Web Application Processing System," was filed on August 4, 2000. I understand that Huang is prior art under 35 U.S.C. § 102(e) because it was filed before the August 30, 2004 filing date of the '408 patent.

38. Huang teaches a method and system for installing and processing web applications written as web pages that have access to the full range of operating system resource, including resources not typically accessible through a web browser. EX1010 at Abstract, 5:7-20. Huang teaches that scripting languages such as JavaScript are commonly used in web content such as HTML documents, and that they can be provided as program code embedded in an HTML document. EX1010 at 8:57-64

39. Huang further teaches a method and system for parsing, for example, the HTML code of the web applications to determine whether it contains references to Uniform Resource Locators (URLs) of web objects that may not be allowed by the web application's security setting. *Id.* at 10:25-36. Huang teaches that if a violation is

detected, for example, if the HTML code includes a link to an URL that is not allowed by the security setting, an exception is generated. EX1010 at 10:31-40.:

E. Walls

40. U.S. Patent No. 7,284,274 ("Walls," EX1011), titled "System and Method for Identifying and Eliminating Vulnerabilities in Computer Software Applications," was filed on January 18, 2002. I understand that Walls is prior art under 35 U.S.C. § 102(e) because it was filed before the August 30, 2004 filing date of the '408 patent.

41. Walls, like Kolawa, teaches a methodology for identifying potential source code vulnerabilities. EX1011 at Abstract. Walls, like Kolawa, generates a parse tree of the code being analyzed and then searches the parse tree for matches that indicate potential vulnerabilities. *Id.* at 7:25-31, 8:31-36. Walls uses a "pipelined" approach to analyze code in stages, such that different parts of a single code stream can be parsed and analyzed at the same time. *Id.* at 7:3-6. One advantage of this technique is "the advantage of pipelining the process where multiple components can be analyzed simultaneously." *Id.* at 7:7-11.

F. Chandnani, Kolawa, Knuth, Huang, and Walls Are All Analogous Art

42. I understand that to combine prior art references when evaluating validity, those references must generally be "analogous." To be analogous, the art

must be in the same field of endeavor as the '408 patent and/or must be pertinent to the problems to which the '408 patent is directed.

43. This requirement is met by each of the references that are used in combination in this declaration. Each reference is in the same field of endeavor as the '408 patent—a field that includes rule-driven "content scanners" for analyzing program code. *See* EX1001 at Title, Abstract.

44. Although the '408 patent focuses on detecting potentially malicious code, a POSA would have understood that scanning for malicious code involves the same or similar techniques as scanning for related code quality and security issues. Most of the written description in the '408 patent focuses on the structure and function of the patent's rule-based scanner, rather than on what the scanner is trying to detect. *See, e.g.*, EX1001 at 6:14-16, Figs. 1-4.

45. The rule-based nature of the '408 patent's scanner means that the underlying structure of the purported invention would not change based on what type of code is being scanned. Instead, because rules can be established to search for arbitrary patterns, the only change necessary to convert from scanning for exploits to scanning for code vulnerabilities (or other code quality issues) would be the inclusion of rules designed specifically to search for tokens and patterns of tokens indicative of code quality problems. *See* EX1001 at 6:17-20 ("An ARB scanner system is

preferably designed as a generic architecture that is language independent, and is customized for a specific language through use of a set of language-specific rules."); 6:35-37 ("It may thus be appreciated that the present invention provides a flexible content scanning method and system, which can be adapted to any language syntax.").

46. Detection of exploits is closely intertwined with detection of code weaknesses because malware often takes advantage of and attacks vulnerabilities and other weaknesses in software code:

> For an experienced hacker or rogue insider, manipulating software to this end is made especially easy due to the variety of information and tools available on-line. An attacker's biggest challenge is simply finding the vulnerabilities in the context of a large business application.

EX1017 at 1:43-48.

47. In some respects, the only difference between a code quality problem and a vulnerability to a malicious virus is the intent of the person who creates or exploits the problem. A quality problem, such as a pattern of code that creates a security hole, might be innocently created by one programmer. That same security hole might also be used by a hacker to propagate a virus.

48. Prior art references confirm the link between code quality and malicious software attacks. For example, the ARCHER reference shows that coding errors "can be exploited by malicious attackers to compromise a system." EX1018 at 1. Similarly,

the Chess patent notes that "security vulnerabilities are subtle, logical errors that can span thousands of lines of code" and that an "attacker's biggest challenge is simply finding the vulnerabilities in the context of a large business application." EX1017 at 2:25-28, 1:46-48.

49. As discussed in more detail below, each reference combined in this declaration is directed to scanning and analyzing programming code and to scanning for potential exploits and/or other security concerns. For example, Chandnani is directed to analyzing code to detect potential viruses. *See id.*; EX1007. Kolawa discloses rule-based systems for detecting potential problems in source code. *See* EX1008 at 2:34-36 ("automatically checking source code quality based on rules"). Knuth is a foundational paper describing the parsing of programming languages from left-to-right. EX1009 at Abstract. Huang is directed to analyzing the code of web application to ensure, among other things, that no security rules are violated. *See* EX1010. Walls scans for security vulnerabilities in programming code. *See* EX1011.

50. Chandnani is directed to rule-driven code scanning. Chandnani uses the term "data driven" in its title and through its specification. *See* EX1007. A person of ordinary skill in the art would have understood "data driven" to be synonymous with "rule-based," not least because Chandnani equates the two. *Id.* at [0069] ("a rule-based approach may be used for script language detection"). Like the '408 patent,

Chandnani can search for different patterns by modifying the data that defines those patterns. *Id*. at [0055]. Also like the '408 patent, Chandnani parses suspect code into tokens and can detect token patterns that correspond to potential exploits even if the byte-for-byte coding of those tokens differs from one iteration to another.

51. Kolawa is also directed to rule-driven code scanning. Kolawa's disclosure focuses on rule-based systems for analyzing code to identify potential problems in the code. EX1008 at 2:34-37 ("automatically checking source code quality based on rules"), Title ("Method and System for Automatically Checking Computer Source Code Quality Based on Rules"). Kolawa also describes its systems as a scanner. *Id.* at 3:66-4:2. Although Kolawa's rule-based scanner is intended to detect "program errors and bugs of all kinds" (as opposed to the "potential exploits" described in the '408 patent), both references detect potentially harmful patterns in code, notwithstanding the intent of the code's author. *Id.* at 1:26-29.

52. A person of ordinary skill in the art would understand that Kolawa is directed to the same general problem as the '408 patent. For example, the '408 patent states that a goal of the purported invention was the ability to perform a "thorough diagnosis" of code in order to recognize patterns of problematic code, even where that code might be written in different ways. EX1001 at 1:42-55. Similarly, Kolawa discloses the use of rules to detect patterns in code that might not be apparent when

examining only the syntax of that code. *See* EX1008 at 2:35-43, 1:35-37 (noting that more complex, quality-related concerns are amenable to being expressed as rules). Kolawa describes how rules that indicate code quality problems operate on patterns of nodes in a parse tree: "Each rule operates on nodes in the parse tree 12 to identify a pattern of nodes unique to the particular rule." *Id.* at 5:48-49.

53. Knuth is analogous art to the '408 patent. Knuth is a foundational paper describing the parsing of programming languages from left-to-right. EX1009 at Abstract. For example, Knuth provides examples of parsing code and building parse trees using a shift-and-reduce process. EX1009 at 618-625, Tables I and II. Knuth also describes the basic parsing steps of recursively matching patterns in strings and generating parent nodes attached to those patterns, thereby generating a parse tree. EX1009 at 609-610.

54. Like the '408 patent, Huang is directed to, among other things, parsing programming code, such as JavaScript and HTML, to detect potential exploits based on security settings. EX1010 at 10:25-36. Huang discloses the use of a web manager that executes a web application by first reading the language code of the web pages and determining the language type of the code. *Id.* at 9:39-46.

55. Walls is analogous art to the '408 patent. Walls is directed to detecting quality problems in programming code. EX1011 at 5:19-21 ("[A] need exists for

certification processes that certify the actual quality of the software."). And Walls expressly describes the close relationship between software quality and malware attacks. *Id.* at 1:48-51 (malicious attacks "are often made possibly by flaws in the software"). Walls also teaches data (rule) driven code analysis through the use of a "knowledge database [that] stores information regarding the various fault classes to be scanned for." *Id.* at 7:31-33.

VI. LEVEL OF ORDINARY SKILL AND RELEVANT TIME

56. I have been advised that "a person of ordinary skill in the art" is a hypothetical person to whom one could assign a routine task with reasonable confidence that the task would be successfully carried out. I have been advised that the relevant timeframe is prior to the relevant priority date, which I understand is August 30, 2004.

57. The relevant technology field for the '408 patent is security programs, including content scanners for program code. By virtue of my education, experience, and training, I am familiar with the level of skill in the art of the '408 patent prior to August 30, 2004.

58. In my opinion, a person of ordinary skill in the relevant field prior to August 30, 2004, would include someone who had, through education or practical

experience, the equivalent of a bachelor's degree in computer science or a related field and at least an additional three to four years of work in the field of computer security.

59. A person of ordinary skill in the relevant field would have been aware of and would have been working with trends from the early-to-mid of the 1990s through the early 2000s, including trends towards scanning and analyzing code as it is received over a network and the incorporation of different security features within malware and vulnerability detection platforms.

60. I understand that the person of ordinary skill in the art is presumed to be aware of the pertinent art.

VII. CLAIM CONSTRUCTION

61. I have been advised that, in the present proceeding, the claims of the '408 patent are to be given their broadest reasonable interpretation in view of the specification. I also understand that, at the same time, absent some reason to the contrary, claim terms are typically given their ordinary and accustomed meaning as would be understood by one of ordinary skill in the art. I have followed these principles in my analysis throughout this declaration. I discuss some terms below and what I understand as constructions of these terms.

"parse tree"

62. I understand that in an earlier instituted *inter partes* reviews of the '408 patent, the term "parse tree" has been construed as "a hierarchical structure of

interconnected nodes built from scanned content." This construction corresponds with my understanding of the meaning of this claim term, and accordingly I adopt this interpretation of the term for the purposes of my analysis.

"dynamically building . . . while said receiving receives the incoming stream"

63. I understand that in an earlier instituted *inter partes* reviews of the '408 patent, the term "dynamically building . . . while said receiving receives the incoming stream" has been construed as "a time period for dynamically building overlaps with a time period during which the incoming stream is being received." This construction corresponds with my understanding of the meaning of this claim term, and accordingly I adopt this interpretation of the term for the purposes of my analysis.

"dynamically detecting . . . while said dynamically building builds the parse tree"

64. I understand that in an earlier instituted *inter partes* reviews of the '408 patent, the term "dynamically detecting . . . while said dynamically building builds the parse tree" has been construed as "a time period for dynamically detection overlap with a time period during which the parse tree is built." This construction corresponds with my understanding of the meaning of this claim term, and accordingly I adopt this interpretation of the term for the purposes of my analysis.

"instantiating . . . a scanner for the specific programming language"

65. I understand that in an earlier instituted *inter partes* reviews of the '408 patent, the term "instantiating . . . a scanner for the specific programming language" has been construed as "substituting specific data, instructions, or both into a generic program unit to make it usable for scanning the specific programming language." This construction corresponds with my understanding of the meaning of this claim term, and accordingly I adopt this interpretation of the term for the purposes of my analysis.

VIII. GROUND 1: Claims 2, 11, 24-28, and 30-34 are rendered obvious by Chandnani in view of Kolawa and Knuth

66. As explained in detail below, it is my opinion that each and every element of claims 2, 11, 24-28, and 30-34 of the '408 patent can be found in the prior art, including the references identified below.

67. Each section of claims 2, 11, 24-28, and 30-34 of the '408 patent is presented below in bold text followed by my analysis of that part of the claim. The analysis below identifies exemplary disclosure of the cited references relative to the corresponding claim elements, and it is not meant to be exclusive.

68. Claim 2 depends from claim 1 and claim 11 depends from claim 9, so I begin my analyses with claims 1 and 9.

69. Chandnani describes computer based systems and methods for detecting polymorphic script language viruses using data-driven lexical analysis. EX1007 at [0002]. For example, in Chandnani the "data detection engine" scans incoming code

for polymorphic viruses by searching for patterns that indicate the presence of potentially malicious programs. EX1007 at [0014]-[0020], [0057]-[0065], FIGS. 1 and 2.

70. Chandnani further explains that the system receives the code, in the form of a data stream, over a network, stating that "a subject file may be downloaded to the computer system or computer through network 78" and "the script language virus detection methodologies may be performed on a file (or a data stream received by the computer through a network) before the file is stored/copied/executed/opened on the computer." EX1007 at Abstract, [0032]-[0034], [0067]. Annotated Figure 2 is a graphical representation of the processes described in Chandnani and shows where the data detection engine 53, receives the data stream.

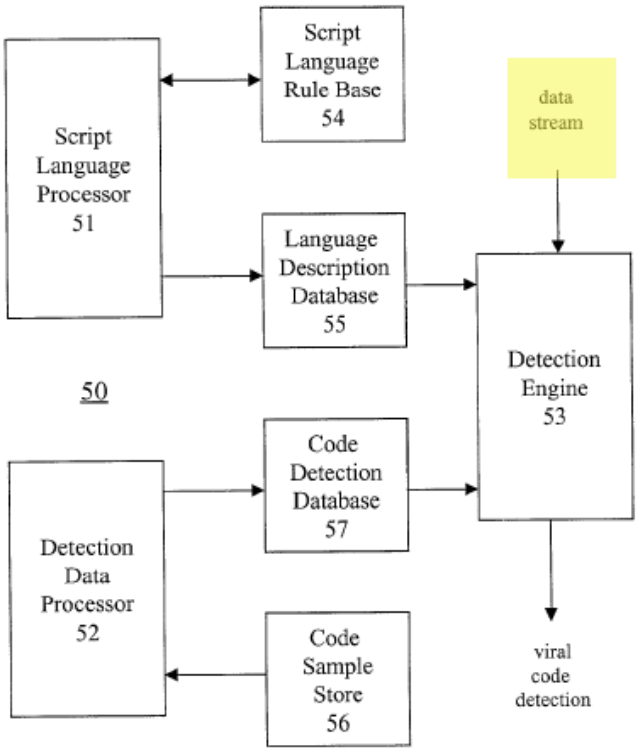


FIG. 2

EX1007 at Figure 2, Annotated

71. Chandnani also discloses that the data detection engine determines the specific programing language in which the code was written by using language check data stored in the language description database. EX1007 at [0034], [0062], Figure 2. Chandnani then uses language definition data to dynamically generate a stream of tokens from the code in the data stream. EX1007 at [0020], [0061], Figure 2.

72. Chandnani goes on to describe how the detection engine dynamically analyzes the stream of tokens for patterns that indicate that the code contains virus or

other malicious actions. EX1007 at [0065]. Chandnani dynamically analyzes the steam of tokens by comparing the patterns of tokens in the data stream to viral code detection data, which can include samples of viral code converted into token patterns stored in a Code Detection Database. EX1007 at [0065], Figure 2. If the token patterns created from the data stream match a token pattern in the viral detection data, then Chandnani signals that viral code was detected. EX1007 at [0065], Figure 2.

73. Although Chandnani may not expressly describe how tokens are parsed and analyzed, Chandnani suggests the use of a parse tree for storing tokens, stating that language description data includes rules "sufficient for the detection engine 53 to lexically analyze and parse a data stream," and the parsing process includes "an output token which indicates that the corresponding pattern has been matched." EX1007 at [0039]-[0046].

74. Furthermore, Kolawa describes how tokens are parsed and analyzed to create a parse tree. Kolawa teaches a method and system for rule-based evaluation of source code quality. EX1008 at 1:19-22. In particular, Kolawa discloses using a "conventional" lexical analyzer that scans code, groups it into tokens, and organizes the tokens using a parse tree:

> The source code 10 is read as input to a lexical analyzer/parser 11 which is conventional in the art. The lexical analyzer scans the source code 10 and groups the instructions into tokens. The parser performs the

> hierarchical analysis which groups the tokens into grammatical phrases that are represented by a parse tree 12.

EX1008 at 3:66-4:4; *see also* EX1008 at Fig. 1. Kolawa describes how the parse tree is searched to identify problematic code based on a set of rules. EX1008 at 4:48-59. Kolawa reports rule violations as error messages that describe the corresponding quality concern. EX1008 at 4:59-60.

Claim 1, preamble: A computer processor-based multi-lingual method for scanning incoming program code, comprising:

75. In my opinion, Chandnani discloses a method for scanning incoming program code. For example, Chandnani states that "[t]he data stream corresponding to a file to scan is tokenized by lexical analysis. The data stream is fed to a lexical analyzer (not shown) in the detection engine which generates a stream of tokens." EX1007 at [0062]. Chanandnai goes on to state that "[t]he detection engine lexically analyzes a data stream using the language description data and the detection data to detect the viral code." EX1007 at [0016].

76. Chandnani also discloses that the method is multi-lingual. For example, Chandnani teaches that "language description data corresponding to one or more script languages is prepared by script language processor 51" and that the "the definitions of target script languages may include language definition rules and possibly language check rules." EX1007 at [0032], [0035]. In my opinion, a person of

skill in the art would understand these statements to clearly describe a system and method that is multi-lingual.

77. Chandnani also discloses that the method is computer processor-based, stating that "the computer system 70 comprises a processor 71." Moreover, even without such a particular statement, a person of skill in the art would understand that a system or method for detection of polymorphic script language viruses by data driven lexical analysis includes a processor for carrying out the task.

Claim 1.1: **receiving, by a computer, an incoming stream of program code**

78. In my opinion, Chandnani discloses receiving, by a computer, an incoming stream of program code. For example, Chandnani describes how a potentially infected file is received as a data stream via a network. Chandnani states that it "provides tools (in the form of apparatus, systems and methods) for detecting script language viruses by performing a lexical analysis of a data stream on a computing device/system" and that the "subject file may be . . . received via a network, such as the Internet." EX1007 at [0029].

Claim 1.2: determining, by the computer, any specific one of a plurality of programming languages in which the incoming stream is written

79. In my opinion, Chandnani discloses determining, by the computer, any specific one of a plurality of programming languages in which the incoming stream is written. For example, Chandnani describes how the target script language of the data stream is determined, stating that "[b]efore the analysis is commenced, target script languages, including their constituent parts, which may be used by the script language viruses, are identified/defined." EX1007 at [0034], *see also* EX1007 at [0062].

Claim 1.3: instantiating, by the computer, a scanner for the specific programming language, in response to said determining

80. In my opinion, Chandnani discloses instantiating, by the computer, a scanner for the specific programming language, in response to said determining. As discussed above, these claim terms are interpreted to mean "substituting specific data, instructions, or both, by the computer, into a generic program unit to make it usable for scanning the specific programming language, in response to said determining." In particular, Chandnani describes how the data stream is analyzed to select which language the data stream is written in, stating that, "[t]he data stream is analyzed using the language check data to select the language definition data to use for the detection process." EX1007 at [0062]. Then the language definition data to use for the detection process, including language definition rules, are selected and used to instantiate the detection engine for the specific language. Chandnani describes

language definition rules as "rules for a target script language describe the constructs of the target script language and any relations between the constructs." EX1007 at [0035]. In other words, the language definition data and rules are specific data and instructions that define how to interpret the target language when it is scanned by the detection engine.

81. In my opinion, the detection engine is a generic detection engine that becomes suitable for scanning a particular programming language after the language definition data is supplied to the lexical analyzer within the detection engine. The lexical analyzer is contained within the detection engine and is the part of the system that scans the data stream and generates a stream of tokens. Chandnani [0062]. This part of the system becomes usable for scanning a specific programming language once the "the selected language definition data and the data stream are supplied to the lexical analyzer." EX1007 at [0062].

Claim 1.4: the scanner comprising parser rules and analyzer rules for the specific programming language

82. As I explain in more detail below, in my opinion, Chandnani discloses parser rules and analyzer rules for the specific programming language of the incoming data stream. Chandnani's scanner, the detection engine, includes "language definition rules and the language check rules (if defined) sufficient for the detection engine 53 to lexically analyze and parse a data stream." EX1007 at [0046]. Chandnani's detection

engine also includes "detection data to detect viral code." Chandnani describes how the detection data is built using samples of viral code and that the "detection data may include multiple layers of tests. Each of the tests may be specified as a token pattern match methodology." EX1007 at [0015], [0051].

Claim 1.5: wherein the parser rules define certain patterns in terms of tokens, tokens being lexical constructs for the specific programming language

83. In my opinion, Chandnani discloses wherein the parser rules define certain patterns in terms of tokens, tokens being lexical constructs for the specific programming language. As discussed below, in my opinion, Chandnani's "language definition rules" supplied to the detection engine are parser rules that define certain patterns in terms of tokens, tokens being lexical constructs for the specific programming language.

84. Chandnani's language definition rules define tokens that are lexical constructs that form the vocabulary of the programming language of the incoming data stream. For example, Chandnani states that "[l]anguage definition rules for a target script language describe the constructs of the target script language" and that they are "sufficient for the detection engine 53 to lexically analyze and parse a data stream." EX1007 at [0035], [0046].

85. Chandnani explains that the language definition rules define lexical constructs in terms of tokens: "[t]he data stream may be converted to a stream of tokens using lexical analysis. The tokens may correspond to respective language constructs." EX1007 at [0020]. Chandnani goes on to state that the rules also define the "relations between the constructs." EX1007 at [0035]. The language definition rules' combination of rules defining both the lexical constructs, and defining the relationships between the constructs, shows that Chandnani discloses parser rules that define patterns in the data stream in terms of tokens, the tokens being lexical constructs for the specific programming language.

Claim 1.6: wherein the analyzer rules identify certain combinations of tokens and patterns as being indicators of potential exploits, exploits being portions of program code that are malicious

86. In my opinion, Chandnani discloses the analyzer rules identify certain combinations of tokens and patterns as being indicators of potential exploits, exploits being portions of program code that are malicious. As discussed below, in my opinion, Chandnani's "viral code detection data" supplied to the detection engine are analyzer rules identify certain combinations of tokens and patterns as being indicators of potential exploits, exploits being portions of program code that are malicious. In particular, Chandnani's viral code detection data is created using a detection regimen

that includes "layers of token pattern matching and/or CRC signature checking." EX1007 at [0050], [0016]; *see also* EX1007 at Fig. 3 ("Prepare detection data for viral code"), [0069] (describing Chandnani's detection methodology as "a rule-based approach"). Chandnani describes how the detection data is built using samples of viral code and that the "detection data may include multiple layers of tests. Each of the tests may be specified as a token pattern match methodology." EX1007 at [0015], [0051]. The "token pattern match methodology" described in Chandnani define rules for identifying characteristics of potentially malicious program code. *See id.* at [0051].

87. Chandnani's detection engine uses these pattern-matching rules to identify potential exploits, stating that "[t]he detection engine lexically analyzes a data stream using . . . the detection data to detect the viral code." EX1007 at [0016]. Chandnani goes on to describe, in detail, how the Code Detection Database and its stored detection data are used in conjunction with the Detection Engine to detect malicious or viral code. EX1007 at Fig. 2, [0065].

Claim 1.7: identifying, by the computer, individual tokens within the incoming stream

88. In my opinion, Chandnani discloses identifying, by the computer, individual tokens within the incoming stream. For example, Chandnani states that "[t]he data stream corresponding to a file to scan is tokenized by lexical analysis. The

data stream is fed to a lexical analyzer (not shown) in the detection engine which generates a stream of tokens." EX1007 at [0062].

Claim 1.8: dynamically building, by the computer while said receiving receives the incoming stream, a parse tree whose nodes represent tokens and patterns in accordance with the parser rules

89. In my opinion, Chandnani in view of Kolawa discloses dynamically building, by the computer while said receiving receives the incoming stream, a parse tree whose nodes represent tokens and patterns in accordance with the parser rules.

90. As discussed above, a "parse tree" is interpreted to mean a "hierarchical structure of interconnected nodes built from scanned content." As also discussed above, "dynamically building . . . while said receiving receives the incoming stream," is interpreted to mean "a time period for building overlaps with a time period during which the incoming stream is being received." Thus, claim 1, element 8 is "dynamically building, by the computer, a parse tree whose nodes represent tokens and patterns in accordance with parser rules, a time period for dynamically building overlapping with a time period during which the incoming stream is being received."

91. For clarity and ease of discussion, I have separated my discussion of this claim element into two parts: discussion related to building the parse tree and discussion related to the dynamically building.

Parse Trees

92. Chandnani teaches parsing a data stream into tokens. EX1007 at 3:65-67 ("The data stream may be converted to a stream of tokens using lexical analysis."), 6:10-23 (storing tokens created as a result of parsing an IF-THEN statement). As discussed above with respect to the state of the art at the time of the filing of the '408 patent, parsing a data stream in the manner taught by Chandnani included the building of a parse tree. *See* above § V.A. Parsing code into a parse tree was common in the art, not only at the time of filing the '408 patent, but for decades before that. *See, e.g.*, EX1009. This is the standard process by which a person of skill in the art parses program code. *See* above § V.A.

93. Chandnani's disclosure of parsing a stream into tokens and then storing those tokens suggests and implicitly teaches using a parse tree, because a person of skill in the art understood that the obvious place to organize those tokens was in a parse tree, including the type of parse tree taught by Kolawa and other prior art references. There are many reasons for concluding that Chandnani's disclosure of parsing a data stream into tokens includes teaching the use of parse tree.

94. First, Chandnani expressly describes identifying expressions in a programming language based on grammar rules for that language, and then storing

those expressions. For example, Chandnani describes the use of a grammar rule for parsing an IF-THEN conditional statement:

> (1) search for the keyword "IF";
> (2) search for the first instance of the keyword "THEN" after the instance of "IF" found in (1);
> (3) store the expression between the keyword "IF" found in (1) and the keyword "THEN" found in (2), as an expression to be parsed;
> (4) search for a statement terminator after the keyword "THEN" found in (2); and
> (5) store the expression between the keyword "THEN" found in (2) and the statement terminator found in (4), as an expression to be parsed."

EX1007 at [0040] – [0045].

95. A person of skill in the art would have understood that if an expression, such as the IF-THEN statement described above, were parsed, the resulting tokens would be organized into a parse tree. As discussed above, parse trees are the data structure used to describe the relationships between programming expressions that are parsed using grammar rules as evidenced by numerous references. *See* EX1008, EX1012, EX1014, EX1015, EX1016, EX1017, EX1019.

96. Second, based on Chandnani's disclosure of a lexical analyzer that parses code into tokens, a person of skill in the art would have known that the tokens had to be organized somehow. Recognizing that an individual token could form part of a

larger grammatical construct, the person of skill in the art would have known that the Chandnani system necessarily had to organize the already identified tokens while awaiting and receiving the next tokens in the data stream. In the example discussed in the previous section, Chandnani's system had to place the "IF" expression in a data structure while waiting for the "THEN" expression that followed. Computer languages in use at the times Chandnani and the '408 patent were filed (2001 and 2004, respectively) include constructs that can span hundreds of intermediate constructs between tokens that signal the beginning and end of a particular expression. Because of this embedding of constructs in other constructs, whatever data structure was selected for use in Chandnani's system had to be able to represent such a hierarchy of tokens. A parse tree would have been perfect for this type of hierarchy.

97. A parse tree also would have been perfect for accomplishing the types of operations on tokens and token patterns that Chandnani describes. For example, Chandnani describes searching tokens to find patterns. EX1007 at [0052] – [0054]. This pattern-match search would have required looking for structural as well as textual patterns (for example, patterns that not only matched the searched-for pattern character by character, but also matched in a meaningful way, such as a pattern that corresponds to the sub-structure of a related pattern). *See* EX1007 at [0052] – [0054], [0009] – [0013]. Because parse trees depict both tokens and their structural

relationship to one another, parse trees enable the type of searching described in both Chandnani and the '408 patent.

98. Furthermore, the tools a person of skill in the art would have used to build the Chandnani system would themselves have used parse trees. A person of skill in the art typically constructs software systems like Chandnani's system using existing components and programming patterns as building blocks. One such component would have been the lexical analyzer/parser taught by Chandnani. A person of skill in the art likely would have used an off-the-shelf lexical analyzer/parser, particularly given that they were well known and often free or inexpensive to obtain. As early as sophomore level computer science classes, students are taught that such parsers generally organized tokens and token patterns in a parse tree, which also confirms the obviousness of using a parse tree to organize the tokens generated by Chandnani's system. *See* above § V.A; *see also* EX1020 at 6 ("Parse tree are particularly easy to construct.").

99. A person of skill in the art, in view of Chandnani's disclosure of parsing a data stream into tokens that represent programming constructs would have chosen the parse tree data structure for organizing the tokens. A parse tree would have also been my first choice and in my opinion, the ideal data structure for this purpose. Many other content-based security scanners, such as those taught by Deb and Scandura,

referred to the use of LEX/YACC and other parsers to perform grammar-based parsing that stored tokens in a parse tree. EX1015 at 10:4-29 (discussing LEX/YACC tools), 3:33-37 ("The method initiates with receiving a message. Then, a grammar associated with the message is identified. Next, the message is converted into a token stream. Then, a parse tree defined by tokens of the token stream is created."), 14:25-40; EX1021 at 2:9-17; EX1022 at 6.

100. In addition, using a parse tree with the Chandnani system would have been obvious because doing so would have been nothing more than the use of a known technique to improve a similar system. To the extent it is argued that Chandnani's system did not use a parse tree, use of a parse tree would have improved the system for the reasons already discussed. There were comparable prior art systems that explicitly used a parse tree data structure (such as Kolawa), and a person of skill in the art could have incorporated a parse tree data structure as a storage technique to improve the Chandnani system in the same way the same technique was used in the Kolawa system.

101. Kolawa taught that parsing groups "tokens into grammatical phrases that are represented by a parse tree" was a technique that was "conventional in the art." EX1008 at 3:65-4:5, Figs. 2, 3. A person of skill in the art would immediately understand that grouping grammatical phrases that are represented by a parse tree, is a

description of the process of building a parse tree. The Kolawa and Chandnani systems were comparable in that both were directed to analyzing code using rule-based pattern matching. A person of skill in the art could have used known parse tree techniques in Chandnani in the same way they were used in Kolawa. Parsers like LEX/YACC were designed to receive code streams as input, and simply using the code stream that already existed in Chandnani as an input to such parsers would have predictably resulted in outputting a parse-tree representation of the incoming code. Most commercially available parsing software available in 2004 stored tokens and patterns in parse trees by default. EX1015 at 10:4-29 (discussing LEX/YACC tools); EX1022 at 6. Numerous prior art references describe the use of parse trees for code analysis. *See* above § V.A.

102. As explained above, using a parse tree in combination with Chandnani would have been obvious to a person of skill in the art, at least because Chandnani suggests such a data structure. Similarly, it would have been obvious to a person of skill in the art to combine Chandnani with the parse tree teachings of other prior art references, such as Kolawa.

103. Combining Kolawa's parse-tree teachings with Chandnani would have further been obvious as a combination of prior art elements according to known methods to yield predictable results. Chandnani and Kolawa together include all

elements of claim 1, including a parse tree, and the only difference between the purported invention of claim 1 in the '408 patent and the prior art was, at most, the lack of an express disclosure of the combination in a single reference. As discussed below, a person of skill in the art would have combined a parse tree with the other elements in the '408 patent claims using known methods, and each element would have performed the same function in the resulting combination as it performed separately. The resulting combination would have been predictable to a person of skill in the art because parse trees had been used successfully in numerous similar applications for decades, as explained in Section V.A.

104. Combining Kolawa's parse-tree teachings with another system like Chandnani would have been simple and predictable, because readily available parser systems were available as standalone components designed to be integrated into other systems. For example, the YACC parser that I and my undergraduate students often use had programmatic hooks to allow for easy integration of the YACC tool with other code. *See* EX1020 at 1-2. When combining Kolawa with Chandnani, each element performs the same function as it does separately. The tokenizer and parser of Chandnani's detection engine accepts the character stream within the data stream as input and continues to create tokens and patterns of tokens from that input and the

parse tree of Kolawa is the data structure in which the tokens and patterns of tokens representing code are organized. EX1007 at [0046]; EX1008 at 3:66-4:13.

105. Because parse trees and tokenizers/parsers were often used together, a person of skill in the art had known methods for combining the two and would have immediately recognized and predicted the resulting combination. A person of skill in the art knew that the most typical data structure for storing such data was a parse tree (that is, an abstract syntax tree). *See* EX1016 at 13:34-36 ("[Q]uery server 102 converts the sourcecode instructions of the submitted query into a parse tree (also known as a syntax tree)."); *see also* EX1012 at 14:4-10 ("[T]he parse tree 58 is a typical processing construct for implementing SQL based access, as is known to those of skill in the art. Accordingly, the exemplary implementation operates on the parse tree 158 representation, adding nodes . . . .").

106. A person of skill in the art would have predictably and successfully used Kolawa's parse tree in combination with Chandnani because Kolawa taught the use of a parse tree for the same purpose for which Chandnani identified and stored tokens: to group tokens and search for patterns of tokens that represent problematic software code. Kolawa explains how it groups tokens in a parse tree:

> The source code 10 is read as input to a lexical analyzer/parser 11 which is conventional in the art. The lexical analyzer scans the source code 10 and groups the instructions into tokens. The parser performs the

> hierarchical analysis which groups the tokens into grammatical phrases that are represented by a parse tree 12.

EX1008 at 3:66-4:4; *see also* EX1008 at Figs. 1, 3.

107. The Kolawa parse tree enabled the software to easily identify patterns of nodes, which is also the purpose of Chandnani's detection engine:

> Each rule operates on nodes in the parse tree 12 to identify a pattern of nodes unique to the particular rule. Examples of source code 10 written in the C++ programming language and corresponding parse trees 12 (or parse tree segments) for each rule are respectively set forth in Appendices A and B.

EX1008 at 5:48-52; *see also* Fig. 2.

108. A person of skill in the art would have been motivated to combine Chandnani with the parse-tree teachings of Kolawa (and other prior art references) for a number of reasons. As a general matter, storing code in tree form makes it easier to manipulate relevant information in that code than it would be if that code were stored only in text form. EX1021 at 3:28-30. Code stored in the tree can be easily moved, combined, and reorganized by manipulation of pointers that connect the tree nodes. Trees are also useful because other ways of looking at code often fail to adequately address hierarchical structural characteristics of the code or to enable the detection of complex structure problems in the code—a key requirement for virus detection. EX1021 at 1:58-63. Tree-based representations "reduce[] the effort required to create

systems for reverse engineering source code," and virus scanning involves reverse engineering the code to understand what it does. EX1021. at Abstract.

109. Using parse trees also would have reduced the costs of building a virus-detection program because available, open-source parsing utilities already used parse trees. *See, e.g.*, EX1020 at 5 ("Parse trees are particularly easy to construct."). These publicly available software utilities had the additional benefit of already having been tested by the many users who relied on them. Such utilities were available for a wide variety of languages, in part because some of the parsers were used to write the compilers and interpreters for those languages (enabling the general use of the languages in the first place).

Dynamically building

110. In my opinion, Chandnani discloses a time period for building overlaps with a time period during which the incoming stream is being received. For example, Chandnani teaches parsing a data stream in which parsing operations (such as tokenizing and storing code) occur during a time period that overlaps with the time period during which the data stream is received. This is consistent with the interpretation of dynamically building discussed above in Section VII.

111. In particular, Chandnani generates tokens by examining each character in the data stream, checking for a match against a state transition table and, depending on

the result of that comparison, outputting a token, all before moving on to the next character in the data stream. For example, Chandnani states that the data stream includes characters, letters, symbols, etc to be scanned. EX1007 at [0057]. Chandnani then describes how it processes the characters in the data steam into tokens, character by character:

> The lexical analyzer . . . retrieves the next character from the data stream and checks if the character matches any of the entries in a current state transition table retrieved from the language definition data corresponding to the current state. If there is a match, the lexical analyzer moves to the next state of the matched transition entry. If there is no match between the character being processed and the state transition table entries for the current state, the lexical analyzer returns to static state 0 and retrieves the next character from the data stream. The next state of the matched transition entry may be a final state with an output token, as described above. When a final state, which has an output token rather than a next state, is encountered, a pattern has been matched and the token is output.

EX1007 at [0063]. The above quote from Chandnani precisely describes a dynamic process of parsing a data stream into tokens as the data stream is being received because the characters in the data stream are retrieved from the data stream one at a time and processed into tokens.

112. In addition, most parsers available in 2004 worked in a similar fashion, by receiving a character, making a determination of whether or not to create a token, and then receiving another character. *See, e.g.*, EX1020 at 6.

113. Furthermore, Chandnani and Kolawa teach parsing a data stream into a parse tree before the entire data stream is received by the computer. For example, Chandnani's "virus detection methodologies may be performed on a . . . data stream received by the computer through a network[] before the file is stored/copied/executed/opened on the computer." EX1007 at [0067].

114. Thus, the Chandnani and Kolawa combination teaches the "dynamically building" limitation. *See* EX1007 at [0067].

Claim 1.9: dynamically detecting, by the computer while said dynamically building builds the parse tree, combinations of nodes in the parse tree which are indicators of potential exploits, based on the analyzer rules

115. In my opinion, Chandnani discloses dynamically detecting, by the computer while said dynamically building builds the parse tree, combinations of nodes in the parse tree which are indicators of potential exploits, based on the analyzer rules. As discussed above in Section VII, "dynamically detecting . . . while said dynamically building builds the parse tree," means "a time period for detecting overlaps with a time period during which the parse tree is being built." Thus, claim

element 1.9 is "dynamically detecting, by the computer, combinations of nodes in the parse tree which are indicators of potential exploits, based on the analyzer rules, a time period for dynamically detecting overlapping with a time period during which the parse tree is being built."

116. For clarity and ease of discussion, I have separated my discussion of this claim element into two parts: discussion related to detecting and discussion related to the time period for detecting overlapping with a time period for building.

Detecting combinations of nodes in the parse tree which are indicators of potential exploits.

117. In my opinion Chandnani discloses identifying combinations of tokens that indicate potential exploits, based on the analyzer rules. For example, Chandnani explains that "[t]he detection data processor prepares detection data for viral code corresponding to a script language virus. The detection engine lexically analyzes a data stream using the language description data and the detection data to detect the viral code." EX1007 at [0016]. As explained above with respect to claim 1, element 6, the detection data for viral code includes tests that may be specified as a token pattern match for viral code. EX1007 at [0050]-[0051]. As I also explained above, a person of skill in the art, based on the disclosures of Chandnani and Kolawa, would parse the data stream into a parse tree. Thus, during the detection process, patterns in the stream of tokens in the parse tree are compared to the patterns of the detection data

for viral code, if there is a match, then viral code was detected. EX1007 at [0062], [0065].

118. Kolawa also discloses that its patterns of tokens, organized in the form a parse tree (EX1008 at 3:66-4:4), are compared with patterns of nodes representing code that may have quality concerns, such as being easily exploitable. EX1008 at 4:52-56.

119. Moreover, even searches for individual nodes in a system built in view of Chandnani and Kolawa are pattern-based searches, because some of the nodes represent recognized patterns of multiple tokens. Therefore, searches for these nodes are also searches for the patterns these nodes represent. In Chandnani, for example, tokens are generated "when the pattern represented by the grammar rule is matched," and one such token represents a coding pattern called an "IF THEN" pattern. EX1007 at [0039]-[0045]. In Kolawa, rules that detect violations "can be the existence of a particular type of node, but can also include a particular sequence or ordering of node types." EX1008 at 8:11-12.

Dynamically Detecting

120. In my opinion, Chandnani discloses a time period for detecting overlaps with a time period during which the parse tree is being built. As discussed above with respect to claim 1, element 8, Chandnani in view of Kolawa teaches building a parse

tree. In particular, the combination works on a data stream of code, parsing it, and then outputting a stream of tokens as a parse tree. *See* analysis with respect to claim 1, element 8, and EX1007 at [0062].

121. Chandnani then goes on to describe how the detection engine operates on the stream of tokens to check from viral code. In particular, the data stream, having been converted into a stream of tokens, is processed using the detection data to check for viral code. EX1007 at [0060], [0062], [0063]. After each token is output, the patterns in the token stream are checked against the patterns in the detection data, stating that "a pattern match or CRC check on the generated token stream is attempted." EX1007 at [0064]; *see also* EX1007 at [0065], Fig. 6.

122. Thus, Chandnani teaches that the detection stage operates on a stream of tokens in the same way the tokenizer operates on the incoming stream of computer code. *See* EX1007 at [0065] ("If the check is a pattern match, the token stream is analyzed lexically using the pattern match detection data and language description data (step 44)."). Thus, the tokenizer, which identifies and organizes tokens, and the analyzer, which searches for tokens and patterns that indicate potential exploits, operate on the incoming data stream at the same time, one on the raw data and the other on the generated stream of tokens.

Claim 1.10: indicating, by the computer, the presence of potential exploits within the incoming stream, based on said dynamically detecting.

123. In my opinion, Chandnani discloses indicating, by the computer, the presence of potential exploits within the incoming stream, based on said dynamically detecting. For example, Chandnani explains in reference to Figure 7, that pattern and CRC matches are preformed and "[i]f [a match] is successful, detection of viral code is signaled (step 46)." EX1007 at [0065]. Chandnani's signal is indicating the presence of potential exploits.

Claim 2: The method of claim 1 wherein said dynamically building a parse tree is based upon a shift-and-reduce algorithm.

124. I explained above how Chandnani and Kolawa disclose every element of claim 1.

125. In my opinion, Knuth discloses the additional element of claim 2 of wherein said dynamically building a parse tree is based upon a shift-and-reduce algorithm.

126. While Chandnani and Kolawa may not expressly disclose building a parse tree based on a shift-and-reduce algorithm, Kolawa does describe the process of building its parse tree as one that is "conventional in the art." EX1008 at 3:66-4:4.

127. Knuth is a foundational paper describing the parsing of programming languages from left-to-right. EX1009 at Abstract. Knuth provides examples of parsing code and building parse trees using a shift-and-reduce process. EX1009 at 618-625, Tables I and II. In one example, detailed in Table I, Knuth describes the shift and reduce process: "'Shift' means 'perform the shift left operation' mentioned in step 2; 'reduce p' means 'perform the transformation (21) with production p.'" EX1009 at 620.

128. A person of skill in the art would have been motivated to combine Chandnani and Kolawa with Knuth at least because Kolawa states that the "source code 10 is read as input to a lexical analyzer/parser 11 which is *conventional in the art*." EX1008 at 3:66-4:4. Knuth, being a foundational paper that describes the shift and reduce algorithm, is a prime example of a parsing algorithm that is conventional in the art. Moreover, as previously discussed, YACC is a conventional program for parsing and compiling source code and YACC uses a shift and reduce algorithm to build a parse tree. *See, e.g.*, EX1023 at 3:49-4:3 (describing the LALR parser as using a shift and reduce algorithm).

Claim 9, preamble: A computer system for multi-lingual content scanning.

129. With respect to this claim element, the preamble of claim 1 includes substantively similar language, so my analysis with respect to that element similarly applies to this claim element.

Claim 9.1: a non-transitory computer-readable storage medium storing computer-executable program code that is executed by a computer to scan incoming program code.

130. In my opinion, Chandnani discloses a non-transitory computer-readable storage medium storing computer-executable program code that is executed by a computer to scan incoming program code. Chandnani discloses that the "apparatus and methods described above (including the associated data and rules) may be embodied in a computer program (or some unit of code) stored on/in computer readable medium, such as memory, hard drive or removable storage media." EX1007 at [0066].

Claim 9.2: a receiver, stored on the medium and executed by the computer, for receiving an incoming stream of program code.

131. In my opinion, Chandnani discloses a receiver, stored on the medium and executed by the computer, for receiving an incoming stream of program code. Chandnani's detection engine satisfies this limitation. Chandnani teaches that its detection engine receives and analyzes a "data stream received by the computer through a network." EX1007 at [0067]; *see also* EX1007 at Figs. 1-2, [0029], [0031]-

[0032], [0057], [0062]. Chandnani also discloses "using a processor for[] receiving a data stream." EX1007 at claim 22. Thus, since Chandnani teaches that its virus detection apparatus and methods "may be embodied in a computer program (or some unit of code) stored on/in computer readable medium" EX1007 at [0066], a person of skill in the art would have understood Chandnani's detection engine to include a unit of code (that is, a set of programming instructions) for receiving the data stream. Thus, Chandnani teaches or renders obvious a "receiver" for receiving an incoming stream of computer code.

Claim 9.3: a multi-lingual language detector, stored on the medium and executed by the computer, operatively coupled to said receiver for detecting any specific one of a plurality of programming languages in which the incoming stream is written.

132. In my opinion, Chandnani discloses a multi-lingual language detector, stored on the medium and executed by the computer, operatively coupled with said receiver for detecting any specific one of the plurality of programming languages in which the incoming stream is written.

133. In Chandnani the data stream is received by the computer through a network (EX1007 at [0067]) and this data stream is then processed by a language detector. Chandnani states that "[t]he detection engine 53 retrieves the language check data from language description module 55 (step 31) and uses the language

check data to lexically analyze the data stream to determine the appropriate script language (step 33)." EX1007 at [0061]. Thus, Chandnani discloses a multi-lingual language detector for detecting any specific one of the plurality of programming languages in which the incoming stream is written.

134. Chandnani also states that the "apparatus and methods described above (including the associated data and rules) may be embodied in a computer program (or some unit of code) stored on/in computer readable medium, such as memory, hard drive or removable storage media." EX1007 at [0066]. A person of skill in the art would understand that the detection engine of Chandnani and the functions it carries out are embodied, at least in part, by stored program code that is executed by the computer.

135. Finally, the detection engine is operatively coupled to the receiver because the data stream passes though the receiver and then to the language detector.

136. Therefore Chandnani discloses a multi-lingual language detector, stored on the medium and executed by the computer, operatively coupled with said receiver for detecting any specific one of the plurality of programming languages in which the incoming stream is written. *See also* EX1007 at [0019]-[0020], [0035], and [0062] and Figs. 2 and 6.

Claim 9.4: a scanner instantiator, stored on the medium and executed by the computer, operatively coupled to said receiver and said multi-lingual language detector for instantiating a scanner for the specific programming language, in response to said determining.

137. With respect to the claim language directed to "instantiator . . . for instantiating a scanner for the specific programming language, in response to said determining," claim 1, element 3 includes substantively similar language, so my analysis with respect to that element similarly applies to this claim element.

138. Moreover, in Chandnani the data stream is received by the language detector through a network and this data stream is then processed by a language detector and a scanner (EX1007 at [0032] and [0067]) which is instantiated with language data. Thus, the instantiator is operatively coupled to the receiver and multi-lingual language detector because it communicates with the language detector which communicates with the receiver and multi-lingual language detector.

139. Chandnani also states that the "apparatus and methods described above (including the associated data and rules) may be embodied in a computer program (or some unit of code) stored on/in computer readable medium, such as memory, hard drive or removable storage media." EX1007 at [0066]. A person of skill in the art would understand that the detection engine of Chandnani and the functions it carries

out are embodied, at least in part, by stored program code that is executed by the computer.

140. Therefore, Chandnani discloses a scanner instantiator, stored on the medium and executed by the computer, operatively coupled to said receiver and said multi-lingual language detector for instantiating a scanner for the specific programming language, in response to said determining. *See also* EX1007 at [0062], [0061], and [0064] and Figs. 2, 6, and 7.

Claim 9.5: the scanner comprising: a rules accessor for accessing parser rules and analyzer rules for the specific programming language

141. With respect to parser rules and analyzer rules for the specific programming language, claim 1, element 4 includes substantively similar language, so my analysis with respect to that element similarly applies to this claim element.

142. Regarding a rules accessor, because claim 9 specifies that the scanner instantiator is stored on the computer-readable storage medium and executed by the computer, a person of skill in the art would have understood that the claimed "rules accessor" is simply a software module (that is, set of programming instructions) that performs the function specified in the claim itself: "accessing parser rules and analyzer rules for the specific programming language."

143. The scanner instantiated by Chandnani's detection engine includes a "rules accessor" for accessing (or retrieving) the language description data and virus detection data. Chandnani discloses that "detection engine 53 retrieves the language check data from language description module 55 (step 31) and uses the language check data to lexically analyze the data stream to determine the appropriate script language (step 33)." EX1007 at [0061], Figs. 2, 6. Then, "[t]he language definition data for the script language determined in step 53 is retrieved from language description module 55 (step 35)." EX1007 at [0061]. Later, the detection engine "retrieves the entries of detection data" used to identify potential exploits. EX1007 at [0064]; *see also* EX1007 at [0058], Figs. 2, 7.

144. Therefore, Chandnani discloses the scanner comprising: a rules accessor for accessing parser rules and analyzer rules for the specific programming language.

Claim 9.6: wherein the parser rules define certain patterns in terms of tokens, tokens being lexical constructs for the specific programming language

145. With respect to this claim element, claim 1, element 5 includes substantively similar language, so my analysis with respect to that element similarly applies to this claim element.

Claim 9.7: and wherein the analyzer rules identify certain combinations of tokens and patterns as being indicators of potential exploits, exploits being portions of program code that are malicious

146. With respect to this claim element, claim 1, element 6 includes substantively similar language, so my analysis with respect to that element similarly applies to this claim element.

Claim 9.8: a tokenizer, for identifying individual tokens within the incoming [data stream]

147. With respect to identifying individual tokens within the incoming data stream, claim 1, element 7 includes substantively similar language, so my analysis with respect to that element similarly applies to this claim element.

148. Furthermore, the lexical analyzer of Chandnani is a tokenizer. Chandnani states that the lexical analyzer generates a stream of tokens from the data stream. See EX1007 at [0062]. *See also* EX1007 at [0020], and Figs. 6 and 7.

Claim 9.9: a parser, for dynamically building while said receiver is receiving the incoming stream, a parse tree whose nodes represent tokens and patterns in accordance with the parser rules accessed by said rules accessor

149. With respect to this claim element, claim 1, element 8 includes substantively similar language, so my analysis with respect to that element similarly applies to this claim element.

150. Chandnani also discloses that a lexical analyzer that parses a data stream using grammar rules. For example, Chandnani states that "[t]he language description data for a target script language is a representation of the language definition rules and the language check rules (if defined) sufficient for the detection engine 53 to lexically analyze and parse a data stream." EX1007 at [0046]; *see also* EX1007 at [0038]-[0045].

Claim 9.10: an analyzer, for dynamically detecting, while said parser is dynamically building the parse tree, combinations of nodes in the parse tree which are indicators of potential exploits, based on the analyzer rules

151. With respect to this claim element, claim 1, element 8 includes substantively similar language, so my analysis with respect to that element similarly applies to this claim element.

152. Chandnani also states that the "apparatus and methods described above (including the associated data and rules) may be embodied in a computer program (or some unit of code) stored on/in computer readable medium, such as memory, hard drive or removable storage media." EX1007 at [0066]. A person of skill in the art would understand that the detection engine of Chandnani and the functions it carries out are embodied, at least in part, by stored program code in the form of an analyzer

that is executed by the computer. *See also* analysis and opinion regarding claim 29, element 9.

Claim 9.11: a notifier, stored on the medium and executed by the computer, operatively coupled to said scanner instantiator for indicating the presence of potential exploits within the incoming stream, based on results of said analyzer

153. With respect to this claim element, claim 1, element 10 includes substantively similar language, so my analysis with respect to that element similarly applies to this claim element.

154. In addition, Chandnani teaches that its detection engine includes units of code that satisfy the "notifier" and "scanner instantiator" limitations. *See* EX1007 at [0066]. Since Chandnani's notifier indicates the presences of potential exploits based on a successful pattern match recognized by the scanner, it would have been obvious to a person of skill in the art that the successful pattern match must be communicated to the notifier by the scanner. Because the scanner instantiator is in communication with the notifier, Chandnani teaches the "operatively coupled" requirement. *See also* EX1007 at [0065] and Fig. 7.

Claim 11: The system of claim 9 wherein said parser dynamically builds the parse tree using a shift-and-reduce algorithm.

155. With respect to this claim, claim 2 includes substantively similar language, so my analysis with respect to claim 2 similarly applies to this claim.

156. Claims 24-28 depend from claim 23 and claims 30-34 depend from claim 29, so I begin my analyses with claims 23 and 29.

Claim 23, preamble: A computer processor-based multi-lingual method for scanning content incoming program code:

157. With respect to this claim element, the preamble of claim 1 includes substantively similar language, so my analysis with respect to that element similarly applies to this claim element.

Claim 23.1: for each of a plurality of programming languages, expressing exploits in terms of patterns of tokens and rules

158. In my opinion, Chandnani discloses for each of a plurality of programming languages, expressing exploits in terms of patterns of tokens and rules. Chandnani teaches creating "language description data corresponding to one or more script languages" and then preparing "viral code detection data . . . for one or more script language viruses." EX1007 at [0032]; *see also* EX1007 at Figs. 3-7. The viral code detection data includes "layers of token pattern matching and/or CRC signature checking." EX1007 at [0050]; *see also* EX1007 at [0016]. These "token pattern match methodologies" define rules for identifying characteristics of computer viruses, or potentially malicious program code. EX1007 at [0052]-[0054]. Chandnani describes one such exploit as a "token pattern match" identified as pattern "p1." EX1007 at [0052]-[0054].

159. Thus, Chandnani discloses expressing exploits in terms of language-specific patterns of tokens and rules for each of a plurality of programming languages. *See also* EX1007 at [0050]-[0051], Figs. 3-7

Claim 23.2: wherein exploits are portions of program code that are malicious

160. With respect to this claim element, claim 1, element 6 includes substantively similar language, so my analysis with respect to that element similarly applies to this claim element.

Claim 23.3: wherein tokens are lexical constructs of a specific programming language

161. With respect to this claim element, claim 1, element 5 includes substantively similar language, so my analysis with respect to that element similarly applies to this claim element.

Claim 23.4: wherein rules designate certain patterns of tokens as forming programmatical constructs

162. In my opinion, Chandnani discloses wherein rules designate certain patterns of tokens as forming programmatical constructs. Chandnani teaches "language definition rules" for a target script language that describe the constructs of the target script language and any relations between the constructs. Chandnani states that "[l]anguage definition rules for a target script language describe the constructs of the target script language and any relations between the constructs." EX1007 at

[0035]. Thus, "relations" define patterns between constructs, which are tokens. See EX1007 at [0035]; *see also* EX1007 at [0039].

163. Chandnani also teaches "grammar rules," such as "IF-THEN" rules, that designate token patterns forming programming constructs. *See* EX1007 at [0040]-[0045].

Claim 23.5: receiving, by a computer, an incoming stream of program code

164. With respect to this claim element, claim 1, element 1 includes substantively similar language, so my analysis with respect to that element similarly applies to this claim element.

Claim 23.6: determining, by the computer, any specific one of the plurality of programming languages in which the incoming stream is written

165. With respect to this claim element, claim 1, element 2 includes substantively similar language, so my analysis with respect to that element similarly applies to this claim element.

Claim 23.7: dynamically building, while said receiving receives the incoming stream, a parse tree whose nodes represent tokens and rules vis-à-vis the specific programming language

166. In my opinion, Chandnani discloses dynamically building, while said receiving receives the incoming stream, a parse tree whose nodes represent tokens and rules vis-à-vis the specific programming language.

167. Claim 1, element 8 includes substantively similar language. While claim 1, element 8 recites "parser rules" rather than "specific programming language," the "parser rules" as discussed with respect to claim 1 are language specific because "parser rules define certain patterns in terms of tokens, tokens being lexical constructs for the specific programming language." EX1001 at claim 1, 19:52-61.

168. Therefore, the plain meaning of both limitations is "nodes represent tokens and token patterns in relation to the specific programming language," and my analysis with respect to claim 1, element 8 applies similarly to this claim element.

Claim 23.8: dynamically detecting, while said dynamically building builds the parse tree, patterns of nodes in the parse tree which are indicators of potential exploits, based on said expressing vis-à-vis the specific programming language

169. In my opinion, Chandnani discloses dynamically detecting, while said dynamically building builds the parse tree, patterns of nodes in the parse tree which are indicators of potential exploits, based on said expressing vis-à-vis the specific programming language.

170. Claim 1, element 9 includes substantively similar language. The analyzer rules in claim 1 are described as rules that "identify certain combinations of tokens and patterns as being indicators of potential exploits." EX1001 at claim 1, 19:52-61. As such, "detecting . . . based on expressing" in this claim element is equivalent to detecting based on analyzer rules in claim 1 and my opinion with respect to claim 1, element 9 applies similarly to this claim element.

Claim 23.9: indicating, by the computer, the presence of potential exploits within the incoming stream, based on said dynamically detecting

171. With respect to this claim element, claim 1, element 10 includes substantively similar language, so my analysis with respect to that element similarly applies to this claim element.

Claim 24: The method of claim 23 wherein said dynamically building comprises positioning nodes of the parse tree corresponding to rules as parent nodes, the children of which correspond to the tokens within the patterns that correspond to the rules.

172. As discussed above, Chandnani and Kolawa disclose all the elements of claim 23. While Chandnani and Kolawa may not expressly describe the creation and positioning of parent and child nodes within the parse tree or expressly describe assigning values to nodes or storing an indicator in a node, this process was well understood in the art and was fundamental to building a functional parse tree.

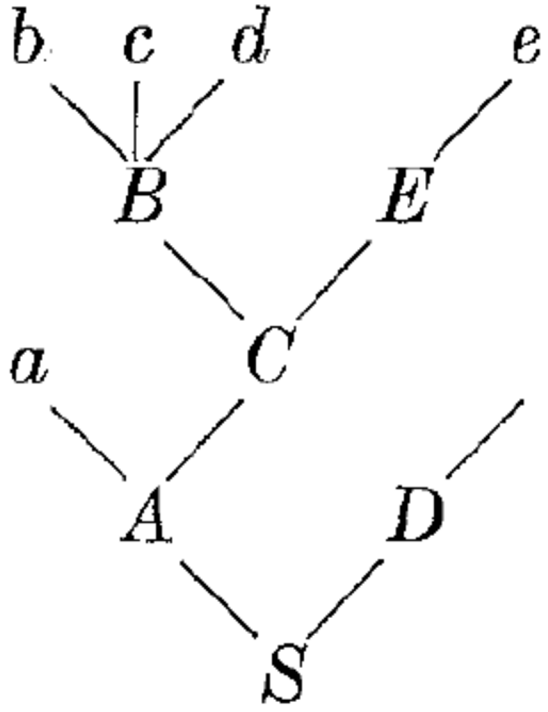
173. Claim 24 includes additional details on the process of building a parse tree. The additional details include "positioning nodes of the parse tree corresponding to rules as parent nodes" and "children" that "correspond to the tokens within the patterns that correspond to the rules." Although recited as additional limitations on claim 23, claim 24 does not actually add anything. Instead, it just recites parts of the well-known process of building a parse tree. Such details are explicitly disclosed in Knuth.

174. Knuth describes methods of translating languages from left to right. EX1009 at Title. Knuth describes that languages translatable from left to right are "particularly important in the case of computer programming" because such languages serve as models for real computer programming languages. EX1009 at 607. Knuth describes algorithms for generating parse trees from sequences of tokens (corresponding, for example, to characters in strings) based on a set of rules defining a grammar. *See, e.g.*, EX1009 at Abstract, 608-10, Tables I and II.

175. Knuth discloses positioning nodes of the parse tree corresponding to rules as parent nodes, the children of which correspond to tokens within the patterns that correspond to the rules. For example, Knuth describes a grammar defined by the rules "S → AD, A→ aC, B→ bcd, C →BE, D →ε, E → e." EX1009 at 609. Knuth describes generating a "derivation tree," corresponding to a parse tree, with parent

nodes corresponding to rules (referred to as "intermediates" and represented with upper case letters) and children corresponding to tokens (referred to as "terminals" and represented with lower case letters). EX1009 at 608. The tree has a root S, referred to as the "principal intermediate character." *Id.* The grammar rules correspond to rules for generating a parse tree: the arrow for each rule connects a parent node on the left to a pattern of child nodes on the right. *See* EX1009 at 632-33 (discussing a "parsing process" that represents matched rules with corresponding nodes in a "derivation tree"). For example, the rule "B→ bcd" indicates that a string of tokens with the ordered pattern "bcd" can be connected as children of a parent node "B." *See* EX1009 at 609.

176. Knuth describes generating a parse tree for this grammar to parse the string "abcde." *Id.* The parent nodes of the tree, reproduced below, correspond to rules (as identified by labels corresponding to the left-hand sides of the rules), and the children of each parent node correspond to the tokens within the patterns that correspond to the rules (found on the right-hand sides of the rules):



Id. at 609, Fig. 3. The structure of the parse tree shows that parent nodes correspond to rules, with their children as corresponding patterns. For example, the upper-left parent node "B" corresponds to the rule "B→ bcd" (*i.e.*, B is on the left side of the rule) and its children (tokens "b," "c," and "d") match the pattern "bcd" on the right side of the rule. Similarly, the children of "A" are the token "a" and the subtree with parent "C;" this corresponds to the rule "A→ aC." It is simple to confirm that each node of the tree corresponds to a rule in the grammar. *See* EX1009 at 609. Knuth further teaches methods of building parse trees for languages using a general algorithm (the shift-and-reduce algorithm) that applies to a wide array of languages. *See* EX1009 at 618-625, Tables I and II; *see also* EX1024 at 4:28-41; EX1023 at 3:34-60.

177. Building a parse tree according to the method of Chandnani and Kolawa, and having the structure taught by Knuth, would involve "positioning nodes of the parse tree corresponding to rules as parent nodes, the children of which correspond to the tokens within the patterns that correspond to the rules," as recited in claim 24, because Knuth teaches that parse trees are built according to this structure.

178. Therefore, in my opinion, Chandnani in view of Kolawa and Knuth discloses said dynamically building comprises positioning nodes of the parse tree corresponding to rules as parent nodes, the children of which correspond to the tokens within the patterns that correspond to the rules.

Claim 25: The method of claim 24 wherein said dynamically building comprises adding a new parent node to the parse tree when a rule is matched.

179. In my opinion, discloses said dynamically building comprises adding a new parent node to the parse tree when a rule is matched. Similar to my analysis with respect to claim 24, claim 25 also merely recites part of the standard process for building a parse tree, which includes adding a new parent nodes to a parse tree when the parsers matches a corresponding rule.

180. Knuth describes a method of building a parse tree that includes adding a new parent node to the parse tree when a rule is matched. Knuth discloses searching a parse tree for a "handle," which is defined as "the leftmost set of adjacent leaves

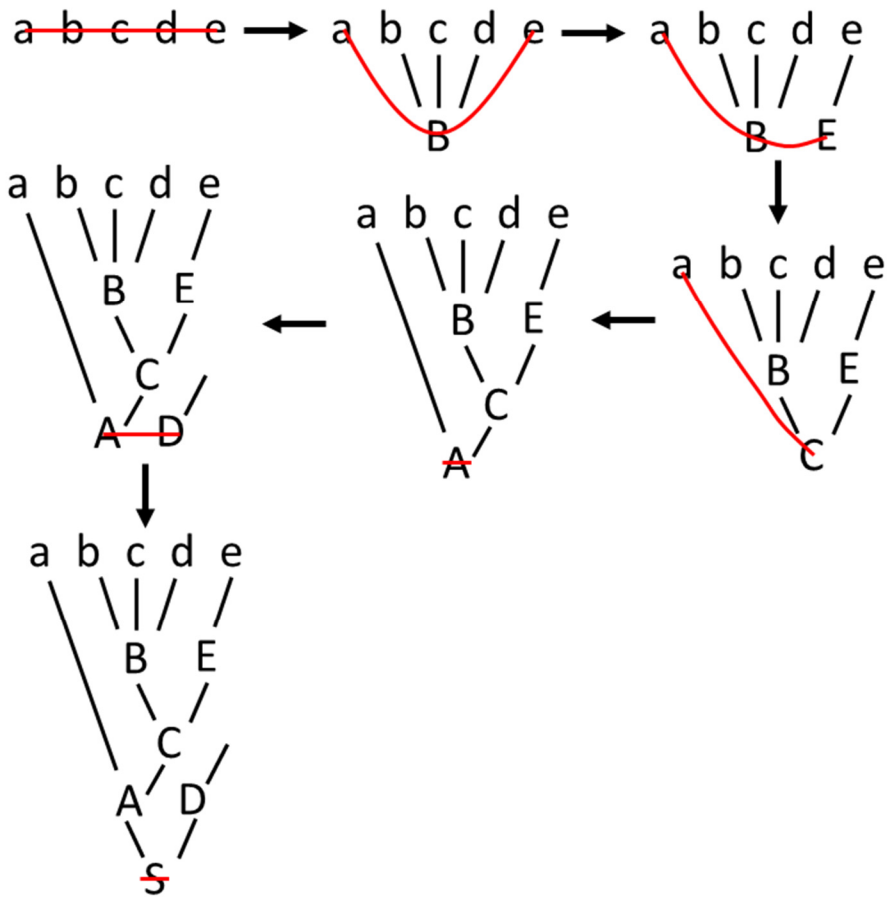
forming a complete branch." EX1009 at 609. In other words, the handle is a pattern that matches a rule—based on the rules of the grammar, a handle is any sequence on the right-hand side of a rule. *See* EX1009 at 609 (describing looking for a handle that matches a rule, such that a parent node can be attached to the sequence of the handle). Knuth describes identifying the handle and "pruning off" the handle by replacing it with the corresponding rule. *Id.* Knuth points out that in the string "abcde," "the handle is bcd," matching the rule "B →bcd." EX1009 at 610. Knuth describes that the first step of parsing the tree is identifying the handle and replacing it in the string with a parent node "B," giving "aBe," where the B is a parent node in the parse tree connected to the matched pattern "bcd." EX1009 at 610. Accordingly, Knuth discloses adding a new parent node to the parse tree when a rule is matched.

Claim 26: The method of claim 25 wherein said dynamically detecting detects patterns of nodes in the parse tree whenever said adding adds a new parent node to the parse tree.

181. In my opinion, Knuth discloses said dynamically detecting detects patterns of nodes in the parse tree whenever said adding adds a new parent node to the parse tree. Knuth describes generating a parse tree by repeatedly identifying a handle, replacing the handle with a parent node matching the handle's pattern, and then searching for a new handle. EX1009 at 609-610. In one example, involving parsing the string "abcde," Knuth states "the process of pruning the handle at each step

corresponds exactly to derivation (5) in reverse." EX1009 at 610. Derivation (5) is "S → AD → A→ aC→ aBE→ aBe→ abcde." EX1009 at 609. Thus, Knuth teaches that the parsing sequence for the exemplary string is abcde → aBe→ aBE→ AC → A→ AD→ S. In other words, "abcde" has "bcd" matched and replaced with "B" to give "aBe," which then has matched "e" matched and replaced with "E," giving "aBE," and so forth until reaching the root S.

182. The manner in which this builds a parse tree matching the parse tree illustrated in Fig. 3 of Knuth from the string "abcde" is illustrated in the following graphic, in which each intermediate tree has a red line drawn through a corresponding intermediate string, which is the string to be parsed in the subsequent step of parsing:



The parser detects a pattern in each intermediate string, adds a corresponding parent node to generate a new tree with a new intermediate string. In the first step, the line passes through “abcde,” indicating that all 5 tokens are to be scanned for patterns. the pattern “bcd,” corresponding to rule “B→bcd” is matched, and the node B is created and matched to the pattern. The remaining string still in need of parsing is “aBe,” indicated by the red line through the token nodes “a” and “e” and the rule node “B.” The “bcd” pattern is already matched to a rule, so it is not part of the remaining string to parse. The parser then repeatedly detects patterns in each new intermediate string,

causing the tree to be built and the red line to move up along the remaining unmatched nodes until reaching the pattern "AD" matching the rule of the root "S," thereby producing a complete parse tree.

183. The overall pattern described by Knuth is to detect a pattern, add a parent node, and then detect a pattern again until the parse tree is complete. Accordingly, Knuth discloses detecting patterns of nodes in the parse tree whenever said adding adds a new parent node to the parse tree, as recited in claims 26 and 32.

184. Unsurprisingly, this matches the behavior described in Chandnani, in which the detection engine operates on the stream of tokens to check for viral code. In particular, the data stream, having been converted into a stream of tokens, is processed using the detection data to check for viral code. EX1007 at [0060], [0062], [0063]. Chandnani teaches that after each token is output, the patterns in the token stream are checked against the patterns in the detection data, stating that "a pattern match or CRC check on the generated token stream is attempted." EX1007 at [0064]; *see also* EX1007 at [0065], Fig. 6.

185. Therefore, Chandnani in view of Kolawa and Knuth discloses detecting patterns of nodes in the parse tree whenever said adding adds a new parent node to the parse tree.

Claim 27: The method of claim 26 wherein tokens and rules have names associated therewith, and wherein said dynamically building comprises assigning values to nodes in the parse tree, the value of a node corresponding to a token being the name of the corresponding token, and the value of a node corresponding to a rule being the name of the corresponding rule.

186. In my opinion, Knuth discloses wherein tokens and rules have names associated therewith, and wherein said dynamically building comprises assigning values to nodes in the parse tree, the value of a node corresponding to a token being the name of the corresponding token, and the value of a node corresponding to a rule being the name of the corresponding rule.

187. In particular, Knuth discloses a parse tree in which rule nodes are given the names A, B, C, D, E, and S; and tokens are given the names a, b, c, d, and e. EX1009 at 609, Fig. 3; *see also id.* at 608 ("we will use upper case letters A, B, C , . . . to stand for intermediates, and lower case letters a, b, c, . . . to stand for terminals"). For example, the leftmost token node of the original string "abcde" is given the name "a," illustrated by the correspondingly named node in Fig. 3. Likewise, when parsing, the first parent node generated by matching a pattern to a rule is the rule node named "B," corresponding to the rule "B→ bcd." *Id.* Knuth also gives further examples of parse trees with named parent and child nodes corresponding to rule names and token names. *See, e.g.,* EX1009 at 633 (showing various nodes, including rule nodes and

tokens, with corresponding names assigned). Accordingly, Knuth discloses assigning values to nodes in the parse tree corresponding to the respective names of tokens and rule nodes, as recited in claims 27 and 33.

188. Kolawa also discloses assigning values to nodes corresponding to the name of rules and tokens. For example, Kolawa discloses:

> The lexical analyzer scans the source code 10 and groups the instructions into tokens. . . . Each instruction is represented in the parse tree 12 by at least one node with interconnecting paths representing dependencies between each token. A root node indicates the entry point into the parse tree. Each node also is of a particular type, such as PLUS__EXPR which is an addition expression node.

EX1008 at 3:66-4:11. The "types" assigned to nodes correspond to names, and those names indicate the token or rule to which the node corresponds. For example, the "PLUS__EXPR" node would correspond to a rule for addition expression. *Id.* In fact, this matches an example by Knuth of a grammatical rule for algebraic expressions which could be expressed in the form "S→(S+S)." EX1009 at 619. Indeed, a person of ordinary skill would find it obvious to name nodes according to their function, in the manned recited in claim 27, as each node must have its function encoded in the node in order to carry it out.

189. Accordingly, Chandnani in view of Kolawa and Knuth discloses "wherein tokens and rules have names associated therewith, and wherein said

dynamically building comprises assigning values to nodes in the parse tree, the value of a node corresponding to a token being the name of the corresponding token, and the value of a node corresponding to a rule being the name of the corresponding rule," as recited in claim 27.

Claim 28: The method of claim 27 wherein said dynamically building comprises storing an indicator for a matched rule in the new parent node of the parse tree when the rule is matched.

190. In my opinion, Knuth discloses wherein said dynamically building comprises storing an indicator for a matched rule in the new parent node of the parse tree when the rule is matched. Knuth discloses that when a new parent node matching a rule is added to a parse tree, a symbol corresponding to the rule is stored in the node. EX1009 at 609, 610. For example, when the rule "B→bcd" is matched to the string of tokens "bcd," a parent node is created with a name "B" and an ordered set of connections to the tokens "b," "c," and "d." *Id.* It would have been obvious that this symbol is stored in the node, along with the stored connections to corresponding child nodes, thereby identifying the rule that was matched by the parser. For example, the node could have its name stored as one or more characters, and its matched pattern indicated by stored pointers to the child nodes. So a node with the name "B" and links to three child nodes with names "b," "c," and "d" would indicate, by that name and those connections, that it was a rule node matching the rule "B→bcd."

191. This teaching is also reflected in Kolawa, which discloses that "[e]ach node also is of a particular type, such as PLUS__EXPR which is an addition expression node." EX1008 at 4:9-11. The type encoded by Kolawa in a rule node is an indicator of the rule being matched; the type of the "PLUS__EXPR" node would indicate that it matched an addition expression rule.

192. Accordingly, Chandnani in view of Kolawa and Knuth discloses "wherein said dynamically building comprises storing an indicator for a matched rule in the new parent node of the parse tree when the rule is matched," as recited in claim 28.

Claim 29, preamble: A computer system for multi-lingual content scanning.

193. With respect to this claim element, the preamble of claim 9 includes substantively similar language, so my analysis with respect to that element similarly applies to this claim element.

Claim 29.1: a non-transitory computer-readable storage medium storing computer-executable program code that is executed by a computer to scan incoming program code.

194. With respect to this claim element, claim 9, element 1 includes substantively similar language, so my analysis with respect to that element similarly applies to this claim element.

Claim 29.2 an accessor, stored on the medium and executed by the computer, for accessing descriptions of exploits in terms of patterns of tokens and rules.

195. With respect to this claim element, claim 1, element 6 and claim 23, element 1 include substantively similar language regarding "pattern match rules that describe exploits in terms of patterns of tokens and rules," so my analysis with respect to those elements similarly applies to this claim element.

196. Chandnani also teaches an accessor for accessing (retrieving) exploit descriptions. For example, Chandnani's detection engine "retrieves the entries of detection data" used to identify potential exploits. EX1007 at [0064]. A person of skill in the art would have understood that the function of retrieving entries is carried out, at least in part by, program code. The program code retrieves, or in other words, accesses the descriptions of exploits.

Claim 29.3: wherein exploits are portions of program code that are malicious.

197. With respect to this claim element, claim 1, element 6 includes substantively similar language, so my analysis with respect to that element similarly applies to this claim element.

Claim 29.4: wherein tokens are lexical constructs of any one of a plurality of programming languages.

198. With respect to this claim element, claim 1, element 6 includes substantively similar language, so my analysis with respect to that element similarly applies to this claim element.

Claim 29.5: wherein rules designate certain patterns of tokens as forming programmatical constructs.

199. With respect to this claim element, claim 23, element 4 includes substantively similar language, so my analysis with respect to that element similarly applies to this claim element.

Claim 29.6: a receiver, stored on the medium and executed by the computer, for receiving an incoming stream of program code.

200. With respect to this claim element, claim 9, element 2 includes substantively similar language, so my analysis with respect to that element similarly applies to this claim element.

Claim 29.7: a multi-lingual language detector, stored on the medium and executed by the computer, operatively coupled with said receiver for detecting any specific one of the plurality of programming languages in which the incoming stream is written.

201. With respect to this claim element, claim 9, element 3 includes substantively similar language, so my analysis with respect to that element similarly applies to this claim element.

Claim 29.8: a parser, stored on the medium and executed by the computer, operatively coupled with said accessor, with said receiver and with said language detector for dynamically building, while said receiver is receiving the incoming stream, a parse tree whose nodes represent tokens and rules vis-à-vis the specific programming language.

202. In my opinion, Chandnani discloses a parser, stored on the medium and executed by the computer, operatively coupled with said accessor, with said receiver and with said language detector for dynamically building, while said receiver is receiving the incoming stream, a parse tree whose nodes represent tokens and rules vis-à-vis the specific programming language. Chandnani discloses a lexical analyzer, also known as a parser, that parses a data stream using grammar rules. For example, Chandnani states that "[t]he language description data for a target script language is a representation of the language definition rules and the language check rules (if defined) sufficient for the detection engine 53 to lexically analyze and parse a data stream." EX1007 at [0046]; *see also* EX1007 at [0038]-[0045].

203. Claim 23, element 7 includes substantively similar language regarding parse tree nodes and tokens, so my analysis with respect to that element similarly applies to this claim element.

204. Claim 1, element 8, includes substantively similar language regarding dynamically building, so my analysis with respect to that element similarly applies to this claim element.

205. Finally, the parser is operatively coupled with the accessor, receiver, and the language detector, all of which are part of Chandnani's detection engine, because the parser is in communication with the accessor, receiver, and language detector.

Claim 29.9: an analyzer, stored on the medium and executed by the computer, operatively coupled with said parser, with said accessor and with said language detector, for dynamically detecting, while said parser is dynamically building the parse tree, patterns of nodes in the parse tree which are indicators of potential exploits, based on the descriptions of exploits vis-à-vis the specific programming language.

206. In my opinion, Chandnani discloses an analyzer, stored on the medium and executed by the computer, operatively coupled with said parser, with said accessor and with said language detector, for dynamically detecting, while said parser is dynamically building the parse tree, patterns of nodes in the parse tree which are indicators of potential exploits, based on the descriptions of exploits vis-à-vis the specific programming language.

207. As described above with respect to claim 1, element 9, Chandnani's detection engine in combination with the parse tree teachings of Kolawa teaches the

dynamic detection of script viruses (exploits) by identifying combinations of tokens stored as notes in a parse tree based on pattern matching rules.

208. As described above with respect to claim 23, element 8, Chandnani also teaches detecting potential exploits "based on the descriptions of exploits vis-à-vis the specific programming language."

209. Chandnani's analyzer is operatively coupled with the parser, the accessor, and the language detector, all of which are part of Chandnani's detection engine, because the analyzer is in communication of the parser, accessor, and language detector.

Claim 29.10: a notifier, stored on the medium and executed by the computer, operatively coupled with said analyzer, for indicating the presence of potential exploits within the incoming stream, based on results of said analyzer.

210. In my opinion, Chandnani discloses a notifier, stored on the medium and executed by the computer, operatively coupled with said analyzer, for indicating the presence of potential exploits within the incoming stream, based on results of said analyzer.

211. As described above with respect to claim 1, element 10, Chandnani's detection engine includes a notifier that indicates the presence of potential exploits within the incoming stream, based on the results of the analyzer.

212. Chandnani also teaches that its detection engine includes units of code that satisfy the "notifier" and "scanner instantiator" limitations. EX1007 at [0066]. Since Chandnani's notifier indicates the presences of potential exploits based on a successful pattern match recognized by the scanner, it would have been obvious to a person of skill in the art that the successful pattern match must be communicated to the notifier by the scanner. Because the scanner instantiator is in communication with the notifier, Chandnani teaches the "operatively coupled" requirement. *See also* EX1007 at [0065] and Fig. 7.

Claim 30: The system of claim 29 wherein said parser positions nodes of the parse tree corresponding to rules as parent nodes, the children of which correspond to tokens within the patterns that correspond to the rules.

213. With respect to this claim, claim 24 includes substantively similar language, so my analysis with respect to that claim similarly applies to this claim.

Claim 31: The system of claim 30 wherein said parser adds a new parent node to the parse tree when a rule is matched.

214. With respect to this claim, claim 25 includes substantively similar language, so my analysis with respect to that claim similarly applies to this claim.

Claim 32: The medium [sic] of claim 31 wherein said analyzer dynamically detects patterns of nodes in the parse tree when said parser adds a new parent node to the parse tree.

215. With respect to this claim, claim 26 includes substantively similar language, so my analysis with respect to that claim similarly applies to this claim.

Claim 33: The system of claim 32 wherein tokens and rules have names associated therewith, and wherein said parser assigns values to nodes in the parse tree, the value of a node corresponding to a token being the name of the corresponding token, and the value of a node corresponding to a rule being the name of the corresponding rule.

216. With respect to this claim, claim 27 includes substantively similar language, so my analysis with respect to that claim similarly applies to this claim.

Claim 34: The system of claim 33 wherein said parser stores an indicator for a matched rule in the new parent node of the parse tree when the rule is matched.

217. With respect to this claim, claim 28 includes substantively similar language, so my analysis with respect to that claim similarly applies to this claim.

IX. GROUND 2: Claim 8 is rendered obvious by Chandnani in view of Kolawa and Huang

218. As explained in detail below, it is my opinion that each and every element of claim 8 of the '408 patent can be found in the prior art, including the references identified below.

219. Each section of claim 8 of the ’408 patent is presented below in bold text followed by my analysis of that part of the claim. The analysis below identifies exemplary disclosure of the cited references relative to the corresponding claim elements, and it is not meant to be exclusive.

220. Claim 8 of the ’408 patent depends from claim 1. My analysis above in Ground 1 with respect to claim 1explains how Chandnani and Kolawa disclose every element of claim 1.

Claim 8.1: The method of claim 1 wherein the incoming stream of program code includes embedded program code

221. Chandnani describes that the analysis methods it discloses are useful for multiple scripting languages, describing the use of “language description data corresponding to one or more script languages.” EX1007 at [0032]; *see also* EX1007 at [0037] (describing language description data for “respective target *languages*.” (emphasis added)). Chandnani discloses the detection of specific programming languages, such as JavaScript and VBScript. *Id.* at [0012]. Accordingly, Chandnani discloses methods useful for handling content in multiple languages.

222. To the extent that, Chandnani and Kolawa do not explicitly disclose that the incoming stream of program code includes embedded program code, Huang provides such disclosure. Huang teaches a method and system for processing web applications written in the form of web pages using, for example, the programming

language HTML. EX1010 at Abstract, 5:7-20. Huang further teaches a method and system for parsing, for example, the HTML code of web applications to determine whether it contains links to Uniform Resource Locators (URLs) that may not be allowed by the web application's security setting. EX1010 at 10:31-36. Huang teaches that if a violation is detected—for example, the HTML code includes a link to a URL that is not allowed by the security setting—an exception is generated. EX1010 at 10:37-40.

223. Huang teaches that scripting languages such as JavaScript are commonly used in web content such as HTML documents, and that they can be provided as program code embedded in an HTML document:

> Those skilled in the art will appreciate that currently the most commonly used script language in web pages is JavaScript. Script in a web page provides a way to embed logic that creates dynamic visual displays or conducts immediate computations when its web page is processed. Traditional script language used in web pages is limited to the browser functions and HTML elements.

EX1010 at 8:57-64; *see also id.* at 1:57-67 (describing browser support for embedded Javascript).

224. As Huang teaches that web content such as HTML documents commonly use embedded code, including scripting languages such as JavaScript, and that such code could include security violations, it would be obvious to use the parsing methods

of Chandnani on such code, as Chandnani describes detection of viral code in files received via the Internet. *See* EX1007 at [0057].

Claim 8.2: identifying, by the computer, another one of the plurality of programming languages in which the embedded program code is written, the other programming language being different that the specific programming language in which the incoming stream is written;

225. Huang discloses the use of a web manager that parses a web page with embedded code and determines a language type for each part of the web page's code:

> the Web application manager executes a Web application by first reading the language code in the Web pages of this application (step 401), and does not terminate (block 403) until all code has been processed (step 402). For each unit of code read, the Web application manager determines the language type of this code (step 404).

EX1010 at 9:39-46; *see also id.* at Fig. 4. In the case of an incoming stream of a web page comprising JavaScript embedded in HTML, the embedded language (JavaScript) would be different than the specific programming language in which the incoming stream is written (HTML). Based on the detected language of the embedded script, the script is then parsed and analyzed for security violations. EX1010 at 11:12-24, Fig. 7.

226. Chandnani also discloses identifying, by the computer, a programming language in which program code is written. Chandnani discloses language rules that can detect programming languages, stating that "[t]he script language processor

prepares language description data corresponding to at least one script language" and that "[d]efinitions of target script *languages* . . . can be rule-based in form." EX1007 at [0016], [0035] (emphasis added).

227. Chandnani additionally describes how "the data stream, in one embodiment in which the target *script languages* are defined by pattern matching rules and patterns are associated with output tokens (described above, may be converted to a stream of tokens." EX1007 at [0060] (emphasis added). In this way, Chandnani teaches that the code is processed into tokens by the detection engine based on the detected language.

228. In light of the teaching of Huang that, when parsing a web page with embedded code, the parser should determine the language type of each code unit for corresponding parsing, it would be obvious when parsing such a file including an embedded code segment to identify the language of the embedded code from among the target script languages of Chandnani, in order to parse the embedded code.

Claim 8.3: repeating said instantiating, said identifying, said dynamically building, said dynamically detecting and said indicating for the embedded program code, based on the parser rules and the analyzer rules for the other programming language.

229. As discussed above, Huang teaches that for each unit of code with a respective detected language, the language is determined and the code is parsed using that language. EX1010 at 9:39-46, 11:12-24, Figs. 4 and 7. This includes embedded code such as JavaScript embedded in HTML. EX1010 at 1:57-67, 11:40-55. The parsing includes code analysis for detection of security violations. EX1010 at 9:39-46.

230. It would be obvious to use the parsing methods disclosed by Chandnani for the detection of viral code in embedded code, as taught by Huang, because Chandnani teaches the parsing and analysis of code from the Internet, which would include HTML pages with embedded JavaScript. *See* EX1007 at [0057], EX1010 at 8:57-64. For embedded code in a different language from the specific programming language in which the incoming stream is written (such as JavaScript in HTML), the analysis of the embedded JavaScript would include repeating said instantiating, said identifying, said dynamically building, said dynamically detecting and said indicating for the embedded program code, based on the parser rules and the analyzer rules for the other programming language (the language of the embedded code).

231. Converting the data stream into a stream of tokens involves instantiating the detection engine with the language rules for each of the identified target languages, otherwise the detection engine would not be able to properly convert a data stream including multiple languages into a stream of tokens in a parse tree data

structure. For embedded code, this would require repeating the instantiating in the corresponding language of the embedded code. Furthermore, a person of ordinary skill in the art would understand that the dynamically building step is repeated each time a token is created and added to the parse tree, as described above with respect to the dynamically building process with respect to claim 1, element 8, above.

232. As discussed above with respect to claim 1, element 9, Chandnani's dynamically detecting process operates and is repeated on a stream of tokens organized in a parse tree each time a token is generated. Furthermore, Figure 7 and the accompanying text in Chandnani show and describe how the detecting pattern is repeated for each detecting data entry. EX1007 at [0065] (stating that "if the pattern match step 44 . . . is not successful, then the method returns to step 42 to select another detection data entry"), Figure 7.

233. Accordingly, it would be obvious to use the methods of Chandnani and Kolawa to parse and analyze embedded code having written in a different language from the specific programming language in which the incoming stream is written, as taught by Huang, which would include identifying the programming language in which the embedded program code is written, and repeating the instantiating, identifying, dynamically building, dynamically detecting, and indicating steps for the embedded code's language.

X. GROUNDS 3 and 4: Claims 2, 8, 11, 24-28, and 30-34 are rendered obvious by the above-identified grounds further in view of Walls

234. As explained in detail below, it is my opinion that each and every element of claims 2, 8, 11, 24-28, and 30-34 of the '408 application can be found in the prior art, including the references identified below.

235. Grounds 4, 5, and 6 are mirror images of Grounds 1, 2, and 3 with the only addition being the Walls reference. That is:

Ground 4: Claims 2, 11, 24-28, and 30-34 are rendered obvious by Chandnani in view of Kolawa, Knuth, and Walls

Ground 5: Claim 8 is rendered obvious by Chandnani in view of Kolawa, Huang, and Walls

236. While it is my opinion that Chandnani and Kolawa disclose to a person of skill in the art the dynamically building and dynamically detecting elements of the claims, Walls provides additional disclosure for these elements because it details a particular method in which a time period for one process overlaps with a time period of another process.

237. Walls is directed to concurrently receiving, parsing, and analyzing pipelined stages. More specifically, Walls "provides a pipelined approach for certifying software wherein distinct components are assembled into a pipeline such that the results of one component are used as input for the next component." EX1011 at 7:3-9.

238. "Pipelining" is a common form of parallel processing that was known before 2004 as a way of increasing throughput by working on multiple stages of a process at the same time. Dictionary definitions describe pipelining in a manner akin to the operation of an assembly line that builds multiple vehicles concurrently, rather than waiting until one vehicle has passed completely through the line before starting to build the next one. EX1025 at 4 (defining "pipelining" as a "method of fetching and decoding instructions (preprocessing) in which, at any given time, several program instructions are in various stages of being fetched or decoded"); EX1026 at 4 (defining "pipeline processing" as a "category of techniques that provide simultaneous, or parallel, processing within the computer").

239. One of the most common uses of pipelining in computer technologies is in instruction execution:

> Pipelining is an implementation technique in which multiple instructions are overlapped in execution. Today, pipelining is key to making processors fast. . . . [T]he work to be done in a pipeline for an instruction is broken into small pieces. . . . Once again, pipelining does not reduce the time it takes to complete an individual instruction; it increases the number of simultaneously executing instructions.

EX1027 at 3; *see also* EX1028 at 1:22-28.

240. Using the pipelining approach described above, Walls builds an "abstract syntax tree" (that is, a type of expanded parse tree) from an already-received code

stream to feed its first pipeline stage (annotated "B" in the figure below) even as other upstream portions of code (annotated "A" below) are waiting to be received:

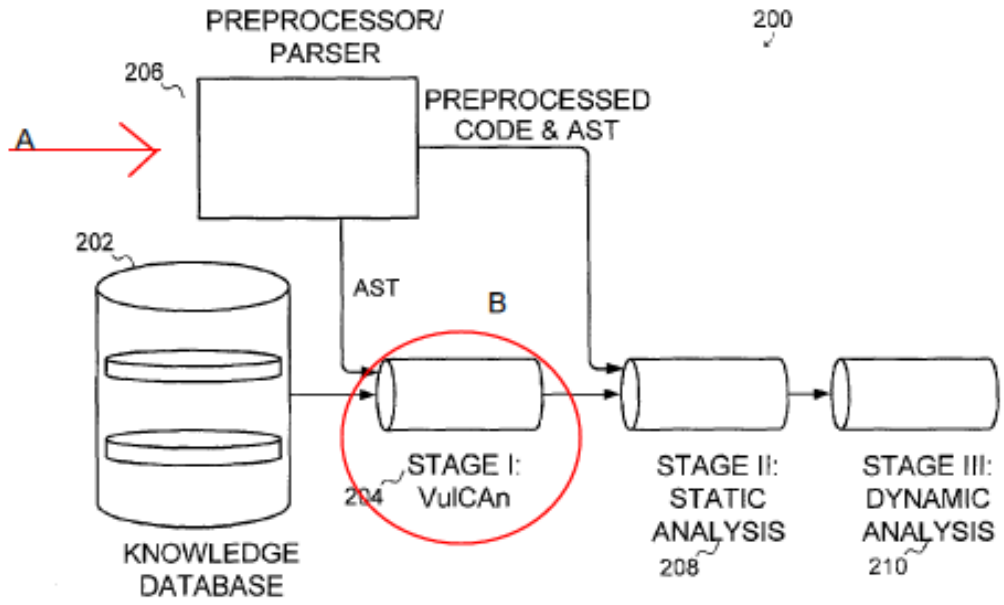


FIG. 2

EX1011 at Fig. 2 (annotations added); *see also* EX1011 at 7:25-31.

241. A person of skill in the art would have understood that there would be additional code information waiting to be parsed at "A" in order to keep the pipeline filled. Otherwise, the pipelined architecture would not achieve the goal of both pipelining and Walls: increased parallelism. *See* EX1011 at 7:8-10. If there was only one segment of code to analyze, the code would advance down the pipeline sequentially, with each stage beginning and ending its processing before sending the output to the next stage. This sequential processing would result in no parallelism at

all. The point of pipelining is to have a long, continuous stream of input entering the first pipeline stage such that, on average, all stages have work to perform on different parts of the stream. *See, e.g.*, EX1025 at 4; EX1026 at 4; EX1027 at 3; EX1028 at 1:22-28.

242. It would have been obvious to combine Walls with other references (including Chandnani and Kolawa) for a number of reasons, including because of the practical requirements of building a stream-oriented virus-checking program. Such systems are limited in the amount of latency (delay) that they can introduce into the overall communication link of which they are a part. If too much latency is introduced into the process, then users will start to complain. These programs typically process multiple data streams when implemented as firewalls (because there are multiple users behind the firewall, each with its own incoming data). And even when acting on behalf of a single user, a single web page request typically results in many different streams for different segments of the web page that are all running in parallel, such as parallel requests for each image on the page. *See, e.g.*, EX1015 at 19:66-67, Fig. 8A. Either scenario results in a large number of incoming streams that will arrive at or near the same time.

243. These practical requirements mean that a virus-detection system must respond to each incoming transaction in a timely fashion. Using a non-parallel

processing mode of operation would result in slower processing and could produce delays as each individual stream is separately processed and holds up all other waiting streams. *See* EX1029 at 2. For example, for a particular data stream, a parsing process may take 30 seconds of processing time and a detecting process may take 31 second of processing time. In a non-parallel processing mode of operation, the total time to complete the parsing and detecting processes on the data stream is 61 seconds of real world time. With parallel processing and pipelining, as the parsing process finishes processing a portion of code and, for example, outputs a token, the detecting process receives the processed portion of code and begins its analysis. In an ideal parallel processing system, the overall real world time to complete the parsing and detecting process would be about 31 seconds. For this reason, parallel processing is highly preferable, and the pipelining architecture described in Walls would have been a known, well-established solution to provide such parallel processing. Thus, it would have been obvious to combine Walls with the other references.

244. Combining Walls with other security scanning references also would have been obvious as a combination of prior art elements according to known methods to yield predictable results. Walls in combination with Chandnani and Kolawa discloses all elements of claim 1, including the "dynamically building" limitation, and the only difference between the purported invention claimed in the '408 patent and the

prior art was, at most, the lack of an express disclosure of the combination in a single reference. A person of skill in the art easily could have combined Walls' pipelined architecture with other elements in the '408 patent claims using known methods, and each element would have performed the same function in the resulting combination as it performed separately. The resulting combination would have been predictable to a person of skill in the art because pipelining techniques had been used successfully in numerous similar applications. *See* EX1027 at 3; EX1028 at 1:22-28.

245. At the time the '408 patent was filed, pipelining techniques were relatively easy to integrate into most programs because the tools for doing so, such as multi-threaded programming features and inter-process communications facilities, were well known, documented, and tested. *See, e.g.*, EX1030 at 1. A person of skill in the art also would have been able to predict the successful operation of the resulting combination because pipelining techniques had been used in the industry for many years in commercial products. *See, e.g.*, EX1031 at 1. The resulting combination would have been predictably successful in the combination of Ground 2 because both Walls and Kolawa used the same type of data structure (trees), and thus the temporal behavior of the references would have supported the same type of parallelization.

246. A person of skill in the art would have been motivated to combine the pipelining approach of Walls with the Chandnani and Kolawa combination also

because Walls notes the benefit of performing multiple operations in parallel: "[T]there is the advantage of pipelining the process where multiple components can be analyzed simultaneously." EX1011 at 7:7-11. Applying the pipelining approach of Walls to the combined teachings of Chandnani and Kolawa would increase the overall processing speed of the scanner and reduce the overall delay experienced by end users protected by the scanner. The pipelining approach also would make better use of multi-processor computing platforms that were widely available as of the filing date of the '408 patent.

247. My analysis of the disclosure of the "dynamically building" element of the claims in Chandnani and Kowala can be found above with respect to claim 1, element 8. My analysis of the disclosure of the "dynamically detecting" element of the claims in Chandnani and Kowala can be found above with respect to claim 1, element 9. To the extent that it is argued that Chandnani and Kowala do not disclose these claim elements, I provide the following discussion regarding the disclosure of Walls.

Dynamically building

248. As discussed above with respect to claim 1, element 8, it is my opinion that Chandnani and Kowala disclose the "dynamically building" element found in the claims.

249. Walls discloses a particular implementation for dynamically building that uses concurrent pipelined stages for receiving and parsing a data stream. As discussed above, Walls is directed to concurrently receiving, parsing, and analyzing pipelined stages. More specifically, Walls "provides a pipelined approach for certifying software wherein distinct components are assembled into a pipeline such that the results of one component are used as input for the next component." EX1011 at 7:3-9.

250. "Pipelining" is a common form of parallel processing that was known before 2004 as a way of increasing throughput by working on multiple stages of a process at the same time. Dictionary definitions describe pipelining in a manner akin to the operation of an assembly line that builds multiple vehicles concurrently, rather than waiting until one vehicle has passed completely through the line before starting to build the next one. EX1025 at 4 (defining "pipelining" as a "method of fetching and decoding instructions (preprocessing) in which, at any given time, several program instructions are in various stages of being fetched or decoded"); EX1026 at 4 (defining "pipeline processing" as a "category of techniques that provide simultaneous, or parallel, processing within the computer"). As Patternson explains and as understood by a person of skill in the art,

> Pipelining is an implementation technique in which multiple instructions are overlapped in execution. Today, pipelining is key to making processors fast. . . . [T]he work to be done in a pipeline for an instruction

> is broken into small pieces. . . . Once again, pipelining does not reduce the time it takes to complete an individual instruction; it increases the number of simultaneously executing instructions.

EX1027 at 3; *see also* EX1028 at 1:22-28.

251. Using the pipelining approach described above, Walls builds an "abstract syntax tree" (i.e., a type of expanded parse tree) from an already-received code stream to feed its first pipeline stage (annotated "B" in the figure below) even as other upstream portions of code (annotated "A" below) are waiting to be received:

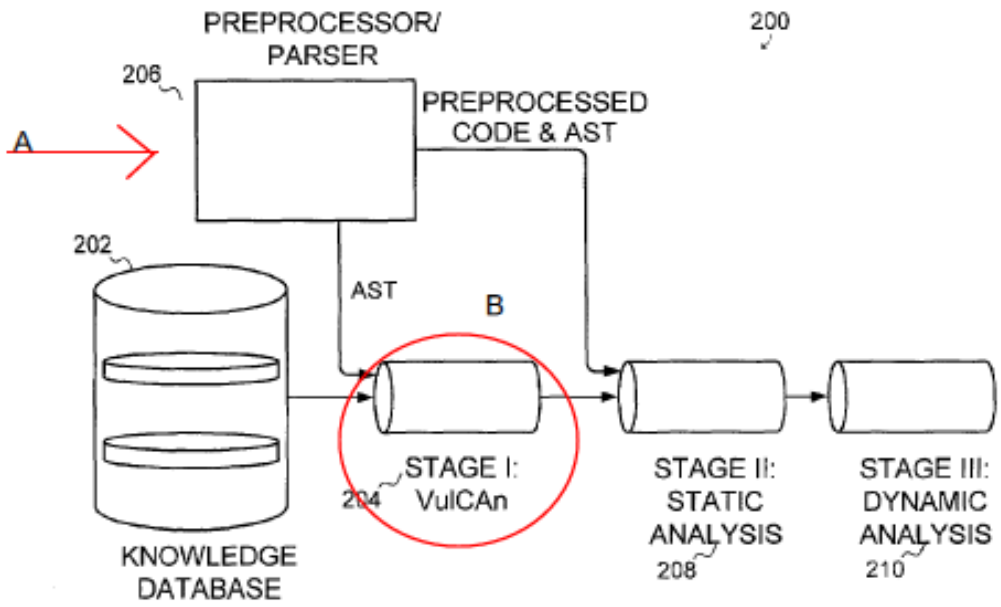


FIG. 2

EX1011 at Fig. 2 (annotations added); *see also* EX1011 at 7:25-31.

252. A person of skill in the art would have understood that when the pipelined approach is applied to Chandnani and Kowala, the data stream is still being received while the parser is generating tokens and building a parse tree.

253. Accordingly, Chandnani, Kowala, and Walls disclose the dynamically building element of the claims.

Dynamically detecting

254. As discussed above with respect to claim 1, element 9, it is my opinion that Chandnani and Kowala disclose the "dynamically detecting" element found in the claims.

255. Walls discloses a particular implementation for dynamically detecting that uses concurrent pipelined stages for receiving and parsing a data stream. As discussed above, Walls is directed to concurrently receiving, parsing, and analyzing pipelined stages. More specifically, Walls "provides a pipelined approach for certifying software wherein distinct components are assembled into a pipeline such that the results of one component are used as input for the next component." EX1011 at 7:3-9.

256. In my opinion, also would have been obvious to combine the pipelining teachings of Walls with the Chandnani and Kowala to implement concurrent pipelined stages for parse-tree building and exploit detection. These teachings are applicable to

parsing and detecting for the same reasons they are applicable to data stream receiving and parsing, as described above. In particular, Walls teaches that its "VulCAn" and "Static Analysis" stages operate on a data stream that is fed by an earlier tree-building stage. EX1011 at 8:23-37, 9:18-19. And because the various stages in Walls' pipeline operate on different parts of the data stream simultaneously (to achieve Walls' goal of increased throughput), a person of skill in the art would have understood that Walls teaches analyzing code during a time period that overlaps with the time period during which the incoming stream is received.

257. Accordingly, Chandnani, Kowala, and Walls disclose the dynamically detecting element of the claims.

XI. CONCLUDING STATEMENTS

258. In signing this declaration, I understand that the declaration will be filed as evidence in a contested case before the Patent Trial and Appeal Board of the United States Patent and Trademark Office. I acknowledge that I may be subject to cross-examination in this case and that cross-examination will take place within the United States. If cross-examination is required of me, I will appear for cross-examination within the United States during the time allotted for cross-examination.

259. I declare that all statements made herein of my knowledge are true, and that all statements made on information and belief are believed to be true, and that

these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code.

Dated: July 15, 2016 By: / Azer Bestavros, PH.D. /

Azer Bestavros, Ph. D.

XII. Appendix – List of Exhibits

Exhibit No.	Description
1001	U.S. Patent No. 8,225,408 (“the ’408 patent”)
1003	Curriculum Vitae of Dr. Azer Bestavros
1004	Excerpt of the File History of U.S. Patent No. 8,225,408 (“408 File History”)
1005	U.S. Patent Application No. 09/539,667
1006	U.S. Patent Application No. 08/964,388
1007	U.S. Patent Appl. Pub. No. 2002/0073330 (“Chandnani”)
1008	U.S. Patent No. 5,860,011 (“Kolawa”)
1009	Knuth, D.E., On the translation of languages from left to right, Information and Control 8, 607-639 (1965) ("Knuth")
1010	U.S. Patent No. 6,968,539 (“Huang”)
1011	U.S. Patent No. 7,284,274 (“Walls”)
1012	U.S. Patent No. 7,437,362 (“Ben-Natan Patent”)
1013	Ron Ben-Natan, “Protecting Your Payload,” SQL Server Magazine, Vol. 5, No. 8 (August 2003) (“Ben-Natan Article”)
1014	U.S. Patent No. 7,210,041 (“Gryaznov”)
1015	U.S. Patent No. 7,546,234 (“Deb”)
1016	U.S. Patent No. 7,185,003 (“Bayliss”)
1017	U.S. Patent No. 7,207,065 (“Chess”)
1018	Yichen Xie, et al., “ARCHER: Using Symbolic, Path-Sensitive Analysis to Detect Memory Access Errors,” Proc. of the 10th ACM SIGSOFT International Symposium on Foundations of Software

	Engineering (Sept. 2003) (“ARCHER”)
1019	U.S. Patent No. 6,697,950 (“Ko”)
1020	Stephen C. Johnson, “YACC: Yet Another Compiler Computer,” Bell Laboratories, Murray Hill, NJ (1978) (“YACC”)
1021	U.S. Patent No. 6,061,513 (“Scandura”)
1022	James F. Power and Brian A. Malloy, “Program Annotation in XML: A Parse Tree-Based Approach,” 9th IEEE Working Conference on Reverse Engineering (Nov. 1, 2002) (“Power”)
1023	U.S. Patent No. 5,822,592 ("Zhu")
1024	U.S. Patent No. 5,276,880 ("Platoff")
1025	Microsoft Press, Computer Dictionary, 3rd ed. (1997) (Excerpt)
1026	Computer Desktop Encyclopedia, 2nd ed. (1999)
1027	David Patterson and John Hennessy, “Computer Organization & Design, The Hardware / Software Interface” (1994)
1028	U.S. Patent No. 5,996,059 (“Porten”)
1029	John Lockwood, “Internet Worm and Virus Protection for Very High-Speed Networks” (August 1998)
1030	Sebastian Gerlach and Roger D. Hersch, “DPS – Dynamic Parallel Schedules,” IEEE Press (2003)
1031	B. Ramakrishna Rau and Joseph A. Fisher, “Instruction-Level Parallel Processing: History, Overview, and Perspective,” The Journal of Supercomputing (1993)