IDENTIFYING AND CONTROLLING UNDESIRABLE PROGRAM BEHAVIORS

Maria M. King*
MKing@Dockmaster.ncsc.mil

Abstract

This paper describes a new mechanism for comparing selected program properties against a pol;icy, or set of rules, that states allowable program behavior[2, 10]. The motivation for this work is the increased need to control undesirable behaviors of programs, such as those inherent in Trojan horses and computer viruses. This mechanism, called an Automatic Policy Checker (APC), is currently implemented under SunOS[1]. This paper will discuss the design and implementation of the APC and the application of the APC to the virus problem. Conclusions concerning anti-viral policy in light of the test results will also be presented.

Introduction

The motivation for this work is the increased need for computer security mechanisms to control undesirable activity of programs, such as those caused by computer viruses[1], Trojan horses and other types of malicious logic.

The major contribution of this work is an automatic tool, called an Automatic Policy Checker (APC), for comparing certain types of program behaviors against a policy that states allowable program behaviors. An important feature of the APC is that it does not implement any specific policy, clearly separating the policy from the mechanism which enforces the policy[8]. Existing mechanisms either rely on the user to specify their own policy[7] or embed an ad hoc policy in the mechanism[5]. The APC allows experiments with policies intended to prohibit a variety of undesirable program behaviors. The APC does not rely on any new architectural support, has minimal effect on performance, and does not require user knowledge of threat. Furthermore, if the APC is used in conjunction with a filter mechanism as described in [2, 6], reliance on some number of humans to act in a trustworthy manner, which is often required in many computer security mechanisms, is no longer needed.

This paper first describes a formal language based on regular expressions that was developed for stating policies and certain types of program behaviors. A high-level overview of the design of the APC is described here while [10] provides a more detailed discussion. The APC has been applied to the computer virus problem. A study of anti-viral policies based on the viral property of *file modification* was conducted and is described in the section on policies. Experiments were run and the empirical data is discussed and results presented.

High-Level Overview

The idea is to explicitly state a system's policy regarding allowable program activity. Subsequently, the APC is used to compare a selected program property against the policy, prior to installation. The APC determines whether a program's specified actions fall within the perimeter of a particular policy.

Definition 1 *A policy is a set of rules that formally states allowable program behavior, in a particular system.*

*Formerly Maria M. Pozzo.
[1]SunOS is a trademark of Sun Microsystems, Incorporated.

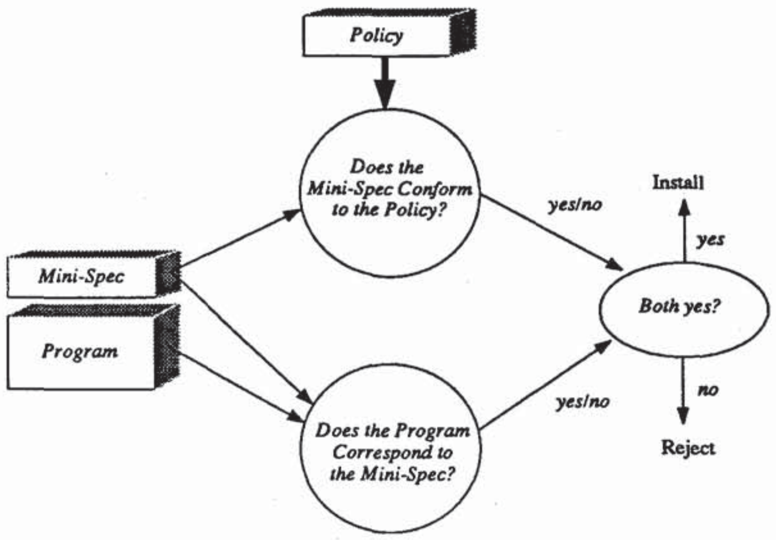
Policy
Does the Mini-Spec Conform to the Policy?
Mini-Spec
Program
Does the Program Correspond to the Mini-Spec?
yes/no
yes/no
Both yes?
yes
Install
no
Reject

Figure 1: High-Level Overview

The term *specification* when applied to programs is usually taken to mean a general statement of *all* of the functional and/or other relevant properties of a program. To distinguish this form of specification from the more general use, the term *mini-spec* is used.

Definition 2 *A mini-spec formally states a selected subset of the functional properties of a program's behavior.*

This paper discusses the question: "Does the mini-spec conform to the policy?" Of equal concern is the correspondence between the mini-spec and the program it specifies. The scheme described in [2] proposes the use of a *filter* that will analyze a binary program and ensure that it conforms to what is stated in the mini-spec (see Figure 1). Traditionally, such an analysis has proven to be difficult. However, the assumption in [2] is that such programs should take full advantage of good software engineering techniques and need not contain the types of actions that are difficult to analyze, such as dynamic code generation, complicated computations for generating object names, and operating system manipulations. The basic premise is that reasonably engineered programs will be analyzable[2]. A reasonably engineered program is one that at least uses a structured methodology, is modular, and is written in a higher-level language. Current research described in [6] has implemented a filter program such as the one proposed in [2]. The filter approach appears promising.

An alternative method for verifying that the program conforms to the mini-spec is source code to specification correlation. The code-to-spec correlation process would have to be altered slightly since it is a one-to-one mapping between each line of code and each line of the specification. The mini-spec only states a *subset* of the program's behavior and such a mapping does not exist. However, verifying the source code against the mini-spec, as opposed to the binary, requires the existence of a trusted means for generating the binary from the source code. Without a trusted means, it would be possible to change the binary during the compilation stage.

The scope of this work is the specification of the mini-spec, development of policy, and the conformance of the mini-spec to a policy. It is assumed that mechanisms exist for verifying a program against its mini-spec, as described above. It is further assumed that once a program is verified against its mini-spec, whether by a filter program or some other means, the program and the associated mini-spec must be sealed or encapsulated in some way to prevent tampering. These issues are well understood and will not be addressed here. The APC accepts a program/mini-spec pair that has been verified and properly sealed. The next section discusses the language used for stating mini-specs and policies.

A Regular Expression Based Specification Language

This section discusses the formal language that was developed for writing mini-specs and policies. The language is based on regular expression notation. The reasons for choosing regular expressions are presented in the next section. The syntax and use of the language is provided in Section , and the limitations of the language are discussed in Section .

Why Regular Expressions?

At the level of an applications program, a system resource might correspond to a file, device, block of memory, an so on. An applications program requests system services through system calls in which a system resource is referenced by a human-readable name. A name translation mechanism converts the human-readable name to the actual page(s) on disk, memory location, etc. The name translation mechanism assumes that the supplier of the name being translated has appropriate access, leaving all access decisions to the access control mechanism, if one exists. The problem is that conventional access control mechanisms are concerned with the access between users and resources, no check is made concerning the access between programs and resources. The example provided in [5] shows how the Fortran compiler only needs access to xyz.for and xyz.obj but can easily gain access to login.com if allowed by the access control mechanism.

The APC controls the access between programs and system resources. The policy is a set of rules which states allowable program behavior. There is one rule for each type of operation under control. Each rule is a set of human-readable names of system resources accessible to that operation. For example, the "modification rule" might be a set of names of directories where modification is permitted on the system. A mini-spec is also a set of rules, one for each type of operation that must be controlled in the particular system. Thus, a program's "modification rule" would be the set of human-readable names of system resources that the program might attempt to modify.

The notion of regular expressions has long been used in the design of lexical analyzers for grouping variable names and other tokens[4]. Other uses for regular expressions include text editors, pattern matching programs, and various file-searching programs. Regular expressions are well-suited for representing a set of strings such as the set of resource names, attribute names, or system call names that can be manipulated by a program.

For ease of discussion, the remainder of this paper will discuss policies and mini-specs that have only one rule, i.e., control a single operation. It is a simple matter to extend these ideas to multiple rules.

Discussion

An *alphabet*, Σ, is a finite set of symbols. A *(formal) language*, denoted L, is a set of strings of symbols from a particular alphabet. The language Σ^* is the set of all strings over a particular alphabet Σ; thus all languages L over Σ are a subset of Σ^*. A regular expression, r, is a way of describing these languages. The notation $L(r)$ denotes the language described by r.

Let r_i be the regular expression that denotes the mini-spec for a particular operation of program i. The set of strings denoted by r_i is a finite-state language over some alphabet Σ. The language specified by r_i is denoted as

$$L(r_i) \tag{1}$$

Let p be the regular expression that denotes the policy, and $L(p)$ is the language denoted by p. Determining if the mini-spec for a given program is acceptable according to the policy of a specific system then becomes a matter of determining if the language represented by the program's mini-spec is a subset of the language denoted by the policy, for each individual rule. More formally, if

$$L(r_i) \subseteq L(p) \tag{2}$$

for each corresponding rule in the policy, then the mini-spec is acceptable according to the system's policy.

Theoretically, the answer to equation 2 is straightforward. Ultimately, we want to be able to compare the two regular expressions without having to elucidate each element in the languages denoted by the expressions. To show that this can be done, consider the following properties of regular expressions.

1. First, the languages denoted by regular expressions are precisely those languages accepted by finite automata; so $L(r_i)$ and $L(p)$ are accepted by deterministic finite automata $M(r_i)$ and $M(p)$, respectively[4, 9]. The class of languages denoted by regular expressions is closed under complementation, i.e., the complement of a language denoted by a regular expression is also a language that can be denoted by a regular expression. To show this, let $M = (Q, \Sigma, \delta, q_0, F)$[2] be a deterministic finite automaton (DFA). Let L be the language over Σ accepted by M; so $L \subseteq \Sigma^*$. Then, the complementary language, $\Sigma^* - L$, is accepted by the DFA $M' = (Q, \Sigma, \delta, q_0, Q - F)$. In other words, M and M' are the same except that the final states are opposite.
2. Second, by definition the languages denoted by regular expressions are closed under union. Therefore, given that the class of languages denoted by regular expressions are closed under complementation and union, it is simple to show that they are also closed under intersection. Let L_1 and L_2 be languages over the alphabet Σ. Then $L_1 \wedge L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$.

Returning to equation 2, to answer the question, consider the following equation:

$$(\Sigma^* - L(p)) \wedge L_i(r) = \emptyset \tag{3}$$

Consider the language that is the complement of the language denoted by the policy. If the language denoted by the program's mini-spec, $L(r_i)$, has anything in common with the complementary language of the policy, $\Sigma^* - L(p)$, then clearly, $L(r_i)$ is not a subset of $L(p)$.

Although it can be shown theoretically that two regular expressions can be directly compared to determine if one is a subset of the other, algorithmically the problem is considered PSPACE-complete[3]. Solutions to many PSPACE-complete problems exist, and in fact, these algorithms work well when certain constraints are applied. The APC currently implements one such algorithm. The primary constraint is that the regular expressions that denote the mini-spec and the policy, must be simple enough to be processed during a reasonable processing cycle. For regular expressions that do not meet this constraint, two alternatives are available. A detailed discussion of the algorithm, and these alternatives is provided in [10].

Language Syntax and Usage

Table 1 identifies the basic operators of the language. The precedence is listed from highest to lowest with the loop operator having the highest precedence. Parenthesis are used to override the normal precedence order as the example in Table 1 shows. The first four operators listed, loop, concatenation, union, and parenthesis for grouping, are standard regular expression operators. Note, however, that the loop operator indicates $0 \leq i$ where i is limited by the maximum string length on a particular machine. Thus, the expression a^* denotes a finite language, which differs from the standard definition.

Nonterminal definitions provide user-friendliness by allowing a user to define commonly used expressions. Nonterminal definition names are 1-8 characters in length, all small letters; the definition itself is written in the operators of the language. Nonterminal definitions can be referenced via the angle brackets (< >) operator and can be embedded. The depth of macro definitions is machine dependent but it is wise to keep a limit on it. Nonterminal definitions are

[2]Where Q is the set of all states in the DFA, Σ is the input alphabet, δ represents the transition function, q_0 is the initial state, F is the set of final states, and $q_0, F \subseteq Q$.[4, 9]

Table 1: Syntax of Language

SYMBOL	MEANING	EXAMPLE
$\star$	loop - $0 \leq i$	$a^* \Rightarrow \{\epsilon, a, aa, aaa, \ldots\}$
	concatenate	$ab \Rightarrow \{ab\}$
$\mid$	union	$a \mid b \Rightarrow a \cup b$; $\{a, b\}$
( )	grouping	$(a \mid b)^* \Rightarrow \{a, b, aa, ab, ba, bb, \ldots\}$
		$a \mid b^* \Rightarrow \{a, b, bb, bbb, \ldots\}$
::=	nonterminal definition	id ::= $(a \mid b)^*$
< >	nonterminal reference (1)	<id>$\Rightarrow$ $(a \mid b)^*$
[ ]	series (2)	$[a \mid b \mid c \ldots]$
{cwd}	current working directory	{cwd}/$(a \mid b)$ $\Rightarrow$ {cwd/a,cwd/b}
{home}	home directory	{home}$(a \mid b)$ $\Rightarrow$ {home/a,home/b}
files	define expression	files ::= <id>
Notes: (1) Nonterminals are 1-8 characters, all small letters. (2) Series can be used with nonterminal definitions.		

stored in files; example nonterminal definition files, called **sysdefs** and **unixdefs**, are shown in Figure 2. A file of nonterminal definitions can be referenced via the "#include" mechanism of Unix. The square brackets operator ([ ]) is used to define a long series such as all the lowercase letters or all the digits. This operator is an implementation enhancement; parenthesis or nothing can be used to represent the same thing, i.e., $(a \mid b \mid c) \equiv a \mid b \mid c \equiv [a \mid b \mid c]$. An improvement to the current language would be to allow $[a - z]$ to indicate all the lowercase letters.

The current working directory operator {cwd} and the home directory operator {home} can be used in systems that have knowledge about filesystem location, such as Unix or Multics. In a Unix system, for example, all directories in the system would include {cwd}/, {home}/, and all other directory locations.

Policies and mini-specs are stored in files. Figure 2 shows the mini-spec for the modification operation for the `calendar` program. The last line of a mini-spec or policy file must begin with the "files" operator followed by the defines or goes into (::=) symbol as shown in the example in Figure 2. The example shows that the `calendar` program can create files in the current working directory of the form "cal" followed by a string as defined in the **unixdefs** nonterminal file. The grammar for the language just described is provided in [10].

Writing Policies and Mini-Specs for Real Programs

A mini-spec is written either during program development by a user wishing to submit a program for installation or it can be written for programs that already exist. Detailed information must be available in order to write a mini-spec for an existing program. This information might include source code, detailed design documentation, programmers notes, and test results.

Writing a policy requires knowledge about the particular threat, the system vulnerabilities, and the desired environment. Although some users may have the sophistication for writing a policy, in most cases the policy should be written by a security officer or other security personnel. Section discusses the application of the APC to the virus problem, the development of anti-viral policy, and presents results of using the APC to test for undesirable program behavior (in this case viral behavior) in 125 Unix programs.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE

fastcase® Smarter legal research.