Model-Carrying Code: A Practical Approach for Safe Execution of Untrusted Applications*

R. Sekar V.N. Venkatakrishnan Samik Basu Sandeep Bhatkar Daniel C. DuVarney
Department of Computer Science
Stony Brook University, Stony Brook, NY 11794.
Email: {sekar, venkat, bsamik, sbhatkar, dand}@cs.sunysb.edu

ABSTRACT

This paper presents a new approach called *model-carrying code* (MCC) for safe execution of untrusted code. At the heart of MCC is the idea that untrusted code comes equipped with a concise high-level model of its security-relevant behavior. This model helps bridge the gap between high-level security policies and low-level binary code, thereby enabling analyses which would otherwise be impractical. For instance, users can use a fully automated verification procedure to determine if the code satisfies their security policies. Alternatively, an automated procedure can sift through a catalog of acceptable policies to identify one that is compatible with the model. Once a suitable policy is selected, MCC guarantees that the policy will not be violated by the code. Unlike previous approaches, the MCC framework enables code producers and consumers to collaborate in order to achieve safety. Moreover, it provides support for policy selection as well as enforcement. Finally, MCC makes no assumptions regarding the inherent risks associated with untrusted code. It simply provides the tools that enable a consumer to make informed decisions about the risk that he/she is willing to tolerate so as to benefit from the functionality offered by an untrusted application.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection, — *Invasive software* ; K.6.5 [**Computing Milieux**]: Management of computing and Information Systems, Security and Protection, Unauthorized access

General Terms

Security, Verification

Keywords

mobile code security, policy enforcement, sand-boxing, security policies

*This research is supported mainly by an ONR grant N000140110967, and in part by NSF grants CCR-0098154 and CCR-0208877, and an AFOSR grant F49620-01-1-0332.

SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.

1. INTRODUCTION

There has been significant growth in the use of software from sources that are not fully trusted — a trend that has accelerated since the advent of the Internet. Examples of untrusted or partially trusted software include: document handlers and viewers (e.g., Real Audio, ghostview), games, peer-to-peer applications (e.g., file sharing, instant messaging), freeware, shareware and trialware, and mobile code (applets, JavaScript, ActiveX).

Contemporary operating systems provide little support for coping with such untrusted applications. Although support for *code-signing* has been introduced into recent OSes, this technique is useful only for verifying that the code originated from a trusted producer. If the code originated from an untrusted or unknown producer, then code-signing provides no support for safe execution of such code. The users (henceforth called *code consumers*) are faced with the choice of either losing out on the potential benefits provided by such code by not running it, or exposing themselves to an unacceptable level of risk by running the code with all of the privileges available to the code consumer.

The lack of OS-level support for safe execution of untrusted code has motivated the development of a number of alternative approaches. These approaches can be divided into *execution monitoring* [14, 12, 31, 33, 18, 1, 19] and static analysis [29, 28, 7, 11, 34, 24]. With execution monitoring, policy violations are detected at runtime, at which point the consumer can be prompted to see if he/she is willing to grant additional access rights to the program, or instead wishes to simply terminate it. In the former case, the consumer is being asked to make decisions on granting additional access to a program without knowing whether these accesses will allow the program to execute successfully, or simply lead to another prompt for even more access. On the other hand, terminating the program causes inconvenience, since the user may have already spent a significant amount of time searching/acquiring the untrusted code, or in providing input to it. In addition, the consumer may have to perform "clean up" actions, such as deleting temporary files created by the program or rolling back changes to important data.

Static analysis-based techniques do not suffer from the inconvenience of runtime aborts. However, from a practical perspective, static analysis techniques are effective only when operating on the source code of programs. Typically, code consumers deal with binary code, which makes it difficult (if not impossible) for them to statically verify whether the code satisfies their policy. Although proof-carrying code (PCC) [29] can, in principle, allow such verification to be applied to binaries, practical difficulties have limited its application to primarily type and memory safety properties. Thus, for the vast majority of code distributed in binary form, and the vast majority of safety policies which concern resource accesses

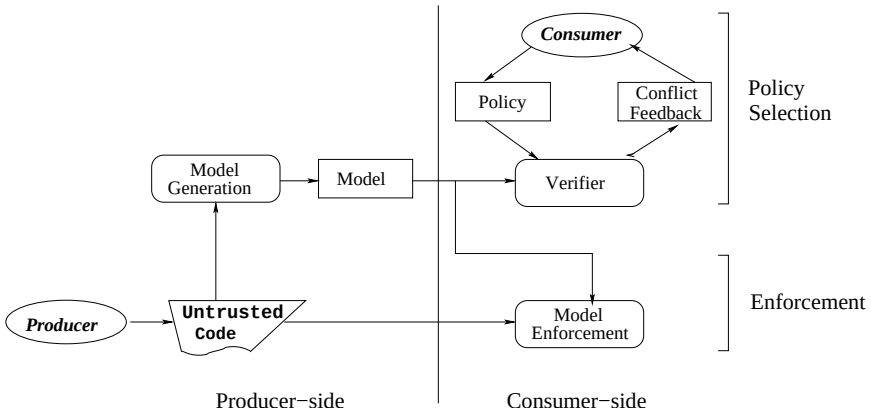


Figure 1: The Model-Carrying Code Framework

made by untrusted programs, static analysis-based approaches do not provide a practical solution for safe execution.

The new approach presented in this paper, called model-carrying code (MCC), combines the convenience of static analysis-based approaches such as PCC (i.e., the reduction or elimination of runtime aborts) with the practicality of execution-monitoring (i.e., the ability to enforce a rich class of consumer-specified security policies). It is inspired by the PCC approach, and shares with it the high-level idea that untrusted code is accompanied by additional information that aids in verifying its safety. With MCC, this additional information takes the form of a *model* that captures the security-relevant behavior of code, rather than a proof. Models enable code producers to communicate the security needs of their code to the consumer. The code consumers can then check their policies against the model associated with untrusted code to determine if this code will violate their policy. Since MCC models are significantly simpler than programs, such checking can be fully automated.

Models serve the important purpose of decoupling the concerns of code producers and consumers. Code producers need not guess security policies of different consumers, nor do they need to expend resources on the generation of proofs in response to requests from consumers. Similarly, code consumers no longer need to tackle the difficult problem of correctly guessing the security needs of an untrusted application. Moreover, they need not reveal their security policies to code producers that they do not trust. Thus, MCC provides a framework for code producers and consumers to collaborate to achieve safe execution of untrusted code. This contrasts with previous execution-monitoring approaches [19, 14, 18, 12] that place the burden of security entirely on the code consumer, and PCC, which places the burden entirely on the code producer.

MCC enables code consumers to try out different security policies of interest to them *prior to the execution of untrusted code*, and select one that can statically be proved to be consistent with the model associated with a piece of untrusted code. This contrasts with purely execution monitoring-based approaches, wherein the consumer needs to deal with repeated runtime aborts (and associated clean-up efforts) to try out different policies; and with PCC, where the only policies that can be statically checked are those for which proofs have been furnished by the code producer.

When a consumer's policy is violated by a model, MCC provides a concise summary of all violations, rather than providing them one by one. By capturing all policy violations in one shot, MCC helps avoid repeated policy violation prompts that are associated with execution monitoring-based approaches. Moreover, this summary is of considerable help in navigating the policy space and identifying the refinement that is most suitable for a given piece of code. Thus, MCC provides support not only for policy enforcement, but also *policy selection* — a problem that has not been addressed by previous research in this area.

1.1 Overview of Approach

The key idea in our approach (see Figure 1) is the introduction of program behavioral *models* that help bridge the semantic gap between (very low-level) binary code and high-level security policies. These models successfully capture security-related properties of the code, but do not capture aspects of the code that pertain only to its functional correctness. The model is stated in terms of the security-relevant operations made by the code, the arguments of these operations, and the sequencing relationships among them. In our current implementation, these operations correspond to system calls, but alternatives such as function calls are also possible.

While models can be created manually, doing so would be a time-consuming process that would affect the usability of the approach. Therefore, we have developed a *model extraction* approach that can automatically generate the required models. Since the model extraction takes place at the producer end, it can operate on source code rather than binary code. It can also benefit from the test suites developed by the code producer to test his/her source code.

The code consumer receives both the model and the program from the producer. The consumer wants to be assured that the code will satisfy a security policy selected by him/her. The use of a security behavior model enables us to decompose this assurance argument into two parts:

- *policy satisfaction:* check whether the model satisfies the policy, i.e., the behaviors captured by the model are a subset of the behaviors allowed by the policy. This can be expressed symbolically as

  $$\mathcal{B}[M] \subseteq \mathcal{B}[P]$$

  where P denotes a policy, M denotes a model, and $\mathcal{B}$ is a function that maps a policy (or a model) to the set of all behaviors satisfied by the policy/model.
- *model safety:* check if the model captures a safe approximation of program behavior — more precisely, that any behavior exhibited by the program is captured by the model:

  $$\mathcal{B}[A] \subseteq \mathcal{B}[M]$$

  Here, A denotes an application, and $\mathcal{B}$ and M have the same meaning as before.

Together, these two imply that $\mathcal{B}[A] \subseteq \mathcal{B}[P]$, i.e., the application A satisfies the security policy P.

Note that model safety is a necessary step whenever the code consumer does not trust the model provided by the code producer.

In particular, the producer may provide an incorrect model either due to malice, or errors/omissions in the model extractor (e.g., failure to account for all possible program behaviors).

In principle, policy satisfaction as well as model safety can be established using static analysis or verification techniques. In practice, however, we resort to runtime enforcement for ensuring model safety due to the difficulties in verifying properties of low-level (binary) code.

The policy selection component in Figure 1 is concerned with policy satisfaction, whereas the enforcement component is concerned with model safety. The policy selection component uses automated verification (actually, model-checking [8]). Since models are much simpler than programs, complete automation of this verification step is possible. If the model is *not* consistent with the policy, the verifier generates a compact and user-friendly summary of all consistency violations. The consumer can either discard the code at this point, or refine the policy in such a way that would permit execution of the code without posing undue security risks.

The policy selection step requires that a consumer be knowledgeable about security issues. Consumers that do not possess this level of knowledge can rely on their system administrator to pre-specify the policy to be used with an untrusted application at its installation time, or provide a set of choices that the user can select prior to execution of the code.

If the refined policy is consistent with the model, then the model and the code are forwarded to the enforcement module. Our current implementation of enforcement is based on system call interception. If the enforcement component detects a deviation from the model, then the execution of the untrusted code is terminated. An alternative to model enforcement is to directly enforce the consumer's security policy, as discussed further in Section 5.

Although execution monitoring, by itself, has the drawbacks mentioned earlier, its use in MCC does not entail the same drawbacks. Note that any enforcement violation in MCC indicates that either the code producer intentionally supplied an incorrect model, or that the model extractor was faulty. The predominance of benign code over malicious code on the Internet indicates that most code producers are not malicious, and hence they will not intentionally provide incorrect models. We expect violations due to model extractor errors to be unlikely as well. Thus, in the vast majority of cases, MCC enables code consumers to use untrusted code *safely* (i.e., their security policy will not be violated) and *conveniently* (i.e., there will be no runtime aborts). In a small minority of cases, where a runtime violation is experienced, safety is still retained, but convenience may be lost.

Although this paper focuses on untrusted programs executing on a UNIX operating system, techniques from our approach could be easily adapted for different execution environments such as Java or Microsoft's Common Language Runtime (CLR) environment (part of its .NET initiative). As a first step in this direction, we have done some preliminary work in defining security policies in terms of security-relevant method calls in Java, and implementing policy enforcement via bytecode rewriting [36].

1.2 Organization of the Paper

We begin our description of the MCC approach with an overview of the MCC policy language in Section 2. Next, in Section 3, we describe our approach for extracting models from programs. The policy selection component is described in Section 4, while enforcement is described in Section 5. Our implementation of the above four components is described in Section 6, together with performance results that establish the practicality of MCC. Finally, concluding remarks appear in Section 7.

2. SECURITY POLICIES

Enforcement in MCC relies on execution monitoring, and hence only *enforceable security policies* [31] are of interest. Such policies are limited to *safety properties.* With other kinds of properties such as those involving information flow [42] and covert channels [23], enforcement is either impossible or impractical, and hence they are not considered in this paper.

Common examples of enforceable policies include access-control and resource-usage policies. Java 2 security model [19] supports standard access-control policies, but can handle applications that consist of code from multiple producers. Naccio [14] supports specification of both access control and resource usage policies. The security automaton formalism [31] can support safety properties that involve sequencing relationships between operations. However, this formalism (and the associated language PoET/PSLang [13]) does not provide the ability to remember argument values such as file descriptors for subsequent comparisons with arguments of other operations. We have shown in [36, 33] that this ability to remember arguments enhances the expressive power of the policy language significantly. Accordingly, our policy language is based on *extended finite state automata* (EFSA) that extend standard FSA by incorporating a finite set of state variables to remember argument values. For instance, we can associate a `write` operation with the file name involved in writing by recording the return value from an `open` operation (a file descriptor), and comparing it with the argument of the `write` operation. Below, we describe this policy language and illustrate it through examples.

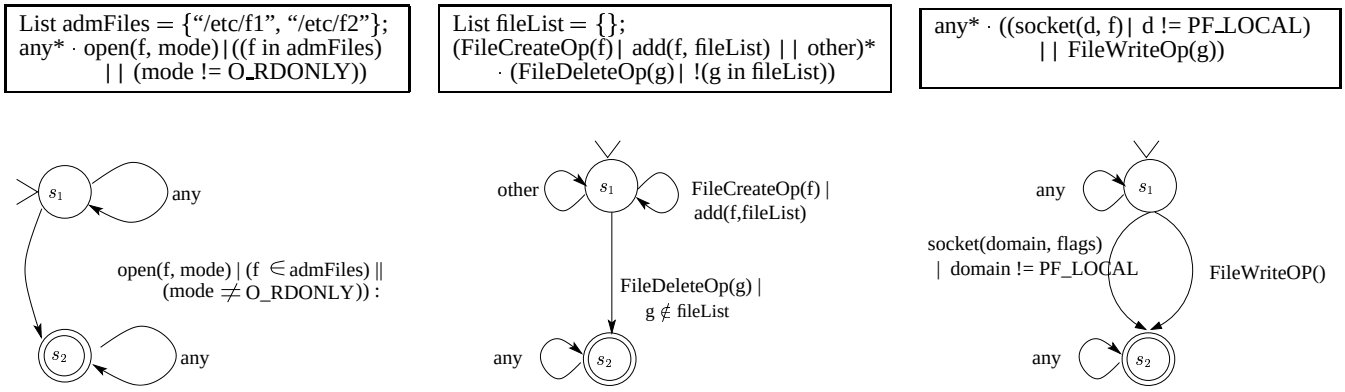
2.1 Security Policy Language

We model behaviors in terms of externally observable *events.* In modern operating systems, security-related actions of programs must be ultimately effected via system calls. For this reason, system calls constitute the event alphabet in our policies. Naturally, it is possible to define behaviors in terms of operations other than system calls, such as arbitrary function calls. Higher level policies can often be stated more easily and accurately in terms of function calls. For instance, a policy that permits a program P to make name server queries can be stated as "program P is allowed to use the function `gethostbyname`" rather than the more complicated (and less precise) version "program P is allowed to connect to IP address xyz on port 53." On the downside, enforcement of such policies will require secure interception of arbitrary function calls, which is not possible in general for binary code.

We use the term *history* to refer to a sequence of events. A history includes events as well as their arguments. A *trace* is a history observed during a single execution of a program. The behavior of a program A, denoted $\mathcal{B}(A)$, is defined to be the set of all traces that may be produced during any execution of A.

Policies capture properties of traces. They are expressed using EFSA. Like security automata, EFSA express negations of policies, i.e., they accept traces that violate the intended policy. The state of an EFSA is characterized by its *control state* (the same notion as the "state" of an FSA), plus the values of (a finite set of) state variables. State variables can take values from possibly infinite domains, such as integers and strings. Each transition in the EFSA is associated with an event, an enabling condition involving the event arguments and state variables, and a set of assignments to state variables. For a transition to be taken, the associated event must occur and the enabling condition must hold. When the transition is taken, the assignments associated with the transition are performed.

EFSA-based policies are expressed in our Behavior Monitoring Specification Language (BMSL). BMSL permits EFSA to be described by defining states, start and final states, and transition rules.



(a) Access control policy (b) History-sensitive policy (c) Sensitive file read policy

Figure 2: Examples of REE policies and their equivalent EFSA representation

BMSL also permits a dual representation of EFSA using *Regular Expressions over Events (REE)* [33]. Just as EFSA extend FSA with state variables, REEs extend regular expressions with state variables. For simple policies, REEs tend to be much more concise and "text-friendly" than EFSA. Hence in practice, we write most of our policies using REEs. The BMSL compiler can translate policies into an EFSA form that is used by the verifier. The EFSA form may also be used for policy enforcement, as we have done in the past for the purposes of intrusion detection [4]. [35] establishes the equivalence of EFSA and REE, so the two notations can be freely mixed in BMSL. (This capability of BMSL is analogous to the ability to mix regular expressions and state machine constructs in Lex.) Further details on REEs and EFSA, including their formal semantics, matching complexity and expressive power can be found in [35]. Below, we provide a short description of BMSL.

Events. Events may be further classified as follows:

- *Primitive events:* There are two primitive events associated with each system call, one corresponding to the system call invocation and the other to its exit. The invocation event has the same name as the system call, while the return event has an "`_exit`" appended to its name. The arguments of the entry event include all of the arguments at the point of call. The arguments to an exit event include all of the arguments at the point of return, plus the value of the return code from the system call.
- *Abstract events:* Abstract events can be used to denote classes of primitive events, e.g., we may define `FileModificationOps` as an event that corresponds to a set of events that modify files. More generally, abstract events may be defined using the notation $event(args) = pat$, where $event$ denotes the abstract event name, and pat is defined below.

Patterns. The simplest patterns, called *primitive patterns*, are of the form $e(x_1, ..., x_n)|cond/asg$, where $cond$ is a boolean-valued expression on the event arguments $x_1, ..., x_n$ and state variables, and asg contains zero or more assignments to state variables. The scope of event arguments is limited to the primitive pattern within which it occurs.

Compound patterns are obtained by composing primitive patterns using operators similar to those in regular expressions. The meaning of compound patterns is best explained by the following definition of what it means for a history H to satisfy a pattern:

- *event occurrence:* $e(x_1, ..., x_n)|cond$ is satisfied by the event history consisting of the single event $e(v_1, ..., v_n)$ if and only if $cond$ evaluates to $true$ when variables $x_1, ..., x_n$ are replaced by the values $v_1, ..., v_n$.
- *alternation:* $pat_1||pat_2$ is satisfied by H if either pat_1 or pat_2 is satisfied by H.
- *sequencing:* $pat_1 \cdot pat_2$ is satisfied by an event history H of the form H_1H_2 provided H_1 satisfies pat_1 and H_2 satisfies pat_2.
- *repetition:* $pat*$ is satisfied by H iff H is empty, or is of the form H_1H_2 where H_1 satisfies pat and H_2 satisfies $pat*$.
- *negation:* $!pat$ is satisfied by H iff pat is not satisfied by H. (The use of negation is not permitted in BMSL if pat involves sequencing or repetition.)

The notion of satisfaction extends in the obvious way when state variables are included, and the details can be found in [35].

We say that a history H matches a policy pat provided that *some prefix* of H satisfies pat.

2.2 Illustrative Examples

Often, it is convenient to group similar events into one abstract event. For instance, there are a number of system calls that can result in the creation or modification of a file, such as `open`, `creat`, and `truncate`. By defining an abstract event:

```
FileWriteOp(f) = (open(f, mode) | writeFlags(mode))
                 || creat(f) || truncate(f)
```

we can use `FileWriteOp` subsequently to denote any of these operations. For readability, we have abstracted a test on the value of `mode` into a call to a function `writeFlags`, which returns true whenever the mode corresponds to opening the file for writing. We have also omitted trailing arguments to `creat` and `truncate` as we are not interested in their values.

Figure 2 illustrates three simple policy examples using REE as well as EFSA notation. Note that the special event any stands for any event, while $other$ stands for an event other than those matching the rest of the transitions on the same state. Since a history H matches an REE whenever a prefix of H satisfies the REE, the REE patterns do not need to have the any transitions that occur in the final state of the EFSA policies.

Figure 2(a) is a simple *access control* policy that prevents writes to all files, and reads from any of the files in a set `admFiles`. Note that the operator `||` is overloaded so that it can represent pattern

alternation as well as the boolean-or operation. If any of these prohibited operations are performed by a program, then the automaton makes a transition from the start state (marked with a ">" symbol) to the final state (marked with a double circle). For any other operations, the transition marked "any" is taken, i.e., the EFSA stays in the start state.

Resource usage policies can also be expressed using EFSA very easily. For instance, a state variable can be used to keep track of the number of open file descriptors, and deny further opens when too many files are open. We do not dwell on resource usage examples in this paper since resource usage properties are not very amenable to fully automated verification.

Figure 2(b) illustrates a *history-sensitive* policy that allows an untrusted application to remove only those files that it previously created. This policy illustrates the use of a list variable `fileList` to remember the names of files that were created by an application. (Here, `FileCreateOp` is an abstract event that should have previously been defined to denote successful returns from all system calls that may create a file.) Any file that the application attempts to delete is checked in this list, and if absent, a policy violation is flagged. Another example of a history-sensitive policy is one that requires an application to close all the files it opened before executing another program. In REE, this policy is expressed as:

```
any* · open_exit(f, fd)|(fd > 0) / (FD = fd)
       · (!close(g)|(g == FD))* · execve()
```

Note that this policy uses only a single state variable FD, but the nondeterministic nature of matching will ensure that a policy violation is reported when any successfully opened file remains open at the time of the `execve` system call.

Figure 2(c) shows the "no network accesses and no file write operations" policy. This policy prevents an application from sending information to an untrusted remote site or write it to an output file. A possible scenario for the use of this policy is the case when an application needs to operate on confidential information.

3. MODEL GENERATION

The problem of generating abstract models from programs has been studied by several researchers in software model-checking [3, 20, 26, 21, 9, 10, 5]. However, very few of these approaches are fully automated, and furthermore, they generate distinct models which are customized for each property to be proved. Property-specific customization greatly simplifies the model, and makes it possible to prove complex properties that could not be proved otherwise. However, in MCC, the code producer generating the model is unaware of consumer security policies. Hence, a *single* model must be generated that is usable for almost all policies. For this reason, some of the previous works in generating program behavior models for intrusion detection are more closely related to MCC model generation than the software model-checking approaches. In particular, [38] develops an approach to derive automata models of program behavior from source code. This approach can generate FSA as well as PDA (push-down automata) models. The principal difficulty in applying this approach to MCC is its inability to systematically reason about system call arguments. Clearly, it is not enough to know that *something* is being written by a program — we need to identify the object being modified by the write. For this reason, an EFSA (or EPDA) model is more appropriate than an FSA (or PDA) model. Moreover, the model generation step needs to capture values of system call arguments, as well the relationships among the arguments of different system calls. For instance, the model should associate a `write` operation with the file being written by capturing the relationship that the file descriptor argument of the `write` is the same as the return value of a previous `open` operation. In the rest of this section, we describe a new technique for generating such models in the context of MCC.

3.1 Model Generation Approaches

MCC models are intended to capture program behavior, which was defined in the previous section to be the set of all possible sequences of security-relevant operations made by a program. In order to capture all possible sequences of operations, our model extraction approach preserves the looping and branching structure present in programs, while abstracting away details such as assignments to internal variables. Figure 3 illustrates a model for a sample program. Note that in Figure 3, `S0` through `S5` denote system calls.

As the above example shows, FSA provide a convenient representation for MCC models, concisely preserving the looping and branching structures within the program. However, the example of Figure 3 omits system call arguments for the sake of simplicity. When argument properties are incorporated into the model, EFSA (rather than FSA) become more natural. It is also possible to use pushdown automata (PDA) for expressing models, which have the benefit of capturing call-return relationships, and hence are more accurate. However, this added accuracy may not be fully useful, since enforcement of PDA models would require secure interception of all function calls made by a program, which is not possible for arbitrary binaries. On the other hand, one important advantage PDA have over FSA is their modularity, i.e., the PDA model of one procedure in a program does not depend on the models of other procedures invoked by it. This factor enables models of different program components (such as libraries) to be extracted independently, and then be composed together to obtain the overall model for the program. Accurate models can hence be synthesized even when the code comes from multiple sources — the most common case of this occurring when an executable from an untrusted producer uses dynamically linked libraries resident on the consumer's workstation (e.g., `libc`).

One approach to model extraction is to use a program analysis technique, such as that described in [38, 7]. The main benefit of this approach is that, if the model generation process strictly avoids unsound assumptions, then the models will be conservative — in this case, using the notation of Section 1.1, we are guaranteed that for an application A and its model M derived by source code analysis, $\mathcal{B}(A) \subseteq \mathcal{B}(M)$. The drawback is that, due to the limitations of source code analysis, M may include execution sequences that can, in fact, never be performed by A. This may lead to a spurious policy violation report from the verifier.

To overcome the spurious violation problem, models may be generated from actual program behaviors observed under different test conditions. The downside of this approach is that program behaviors that are not observed during model generation may not be captured in the model. This may lead the verifier to conclude that a program satisfies a policy, when it actually does not. To minimize this possibility, as many program behaviors as possible should be exercised during the learning process. This can be accomplished using a comprehensive test suite, which the code producer most likely will have already developed for testing purposes. Depending on the comprehensiveness of the test suite, there may still be a probability that the application deviates from its model. In such cases, the MCC enforcement mechanism will terminate the program. Note that in this case, safety is still preserved, but the convenience (of not having runtime aborts) is lost. Fortunately, this happens only in the (hopefully very small) fraction of runs that deviate from the model.

We are currently pursuing the extraction of EFSA models us-

```
1.  S0;
2.  while (..) {
3.        S1;
4.        if (...) S2;
5.        else S3;
6.        if (S4) ... ;
7.        else S2;
8.        S5;
9.  }
10. S3;
11. S4;
```

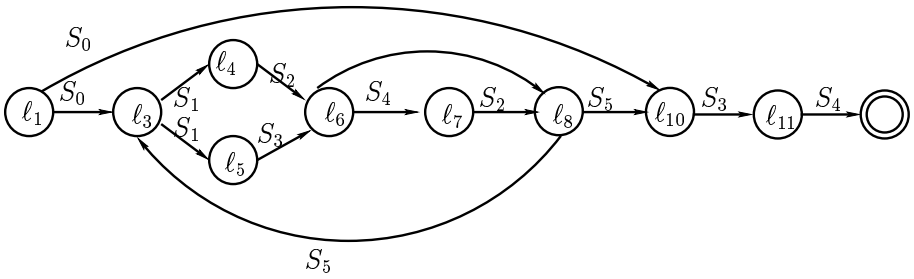


Figure 3: A sample program and its model

ing execution monitoring, and EPDA (standard PDA extended with state variables) models from source code. In both cases, the main focus of our research has been in tracking data flow relationships affecting critical system call arguments such as the resource accessed using a system call. Currently, our implementation of source-code model extraction is not mature enough, so our description below focuses on execution monitoring-based model extraction.

3.2 Model Generation via Execution Monitoring

In the context of intrusion detection, a number of techniques have been developed for extracting program behavior models in terms of system calls [16, 40, 32, 6, 15, 27, 25]. Some of these techniques [16] learn a finite set of fixed-length strings of system calls, while some others are capable of learning (a finite set of) variable-length strings [40]. We developed a new approach in [32] that is capable of representing an infinite number of strings of unbounded length using a finite-state automaton. Another approach [6] for learning FSA has been developed recently. [15] builds on [32] to develop an approach that learns PDA models rather than FSA models, but this approach incurs significantly higher overheads.

While FSA can serve as a starting point for MCC models, they are not sufficient by themselves — in particular, MCC models need to capture crucial information, such as file names or network addresses that will be referenced by security policies. The main focus of MCC research in model extraction has been to develop a *fully automated* algorithm for extracting such argument relationships. This contrasts with human-assisted approaches such as [2], where a programmer was required to identify the subsequences within an execution trace within which such argument relationship will be attempted. Moreover, our approach is aimed at learning relationships involving complex data types such as file names, whereas [2] is concerned only with integer arguments. Below, we describe our approach for model extraction via machine learning from execution traces. We first provide a brief overview of our approach for learning FSA from system call sequences [32], and then proceed to describe the extensions to this algorithm for learning argument relationships.

3.2.1 Overview of FSA Learning Algorithm

It is well-known that learning FSA from strings (execution traces, in our case) is a computationally hard problem [30]. The primary difficulty is that the strings, by themselves, do not give any clue as to the state of the automaton. For instance, if we see a string $abcda$, we cannot determine whether the two a's in the string correspond to the same state of the automaton or different states. The key insight behind the technique of [32] is that we can indeed obtain state-related information if we knew the location from where the system call was made. Based on this call location information, an FSA is constructed as follows. Whenever a system call is made from program location ℓ_i, we first create a new automaton state labeled ℓ_i if it is not already present. In addition, if S' was the previous system call and was made from location ℓ_{i-1}, then an edge labeled with S' is created from the previous system call location ℓ_{i-1} to ℓ_i. Using this approach, Figure 3 illustrates the model learned from the following two execution traces.

- $\frac{S_0}{\ell_1}\frac{S_3}{\ell_{10}}\frac{S_4}{\ell_{11}}$
- $\frac{S_0}{\ell_1}\frac{S_1}{\ell_3}\frac{S_2}{\ell_4}\frac{S_4}{\ell_6}\frac{S_5}{\ell_8}\frac{S_1}{\ell_3}\frac{S_3}{\ell_5}\frac{S_4}{\ell_6}\frac{S_2}{\ell_7}\frac{S_5}{\ell_8}\frac{S_3}{\ell_{10}}\frac{S_4}{\ell_{11}}$

In these traces, the notation $\frac{S}{\ell_i}$ denotes that the system call S is being made from the line ℓ_i. (For illustrative purposes, line numbers are used in place of locations of machine instructions in this example.) Note that automaton states are labeled with the location from where each system call was made.

The simple description given above needs to be extended when we take into account the fact that most programs make extensive use of libraries. For instance, system calls are usually made from wrapper functions provided by `libc`. In this case, note that each system call will be made from exactly one location in `libc`, and hence the automaton will not capture useful information about the executable that is making these system calls. To address this problem, our learning algorithm ignores the program locations within libraries, instead using the location within the executable from where these library calls were invoked. This requires a "walk" up the program stack at the point of system call. We describe an implementation of the stack-walk for Linux/x86. Implementations for a different OS/architecture will typically be very similar. On Linux/x86 the EBP (extended base pointer) register is used to chain together activation records of a caller and callee functions. Specifically, the return address for the current procedure is found at the location (EBP+4), while the base pointer of the caller is stored at the location (EBP). The range of locations within the executable can be found by reading the pseudo file `/proc/`pid`/maps`, where pid denotes the process identifier for the monitored process. Using this information, the stack frame is traversed up from the point of the system call until a return address R within the executable is located. This location R is used in the model as the location from where this system call was made.

3.2.2 Learning Argument Values

Before describing the algorithm for learning system call argument values, it is necessary to provide an overview of the implementation architecture of the model extractor. The model extractor consists of an online and an offline component. The online component consists of a runtime environment to intercept system calls and a logger that records these system calls and their arguments into a file. The logger incorporates some level of knowledge about

what system call arguments (and return values) are useful for model extraction, and whether any preprocessing is necessary on these arguments. For instance, the logger converts file and directory name arguments into a canonical form before recording them. Similarly, it extracts the IP address and port information from sockets and records them explicitly. It ignores some system call arguments and return values, such as buffers returned by the `read` system call, most fields of the structure returned by `stat` system call, etc. The offline component consists of two parts: the EFSA learning algorithm, and a log file parser.

The extension of the FSA algorithm to learn system call argument values proceeds as follows. First, we may be interested in absolute values of arguments. For instance, a model should capture names of specific files opened by an application. To accomplish this, our algorithm records system call argument values together with each system call in the FSA. If there are multiple invocations of a system call along an edge in the FSA, the model extractor collects argument values from each of the invocations. If the number of such values crosses a threshold, then an aggregation algorithm is employed to summarize the values, rather than listing each one. In principle, the learning algorithm should support different aggregation operations, but in practice, we have so far found the need for only two such operations: the longest common prefix operation for file names, and the union operation for sets represented using bit vectors, e.g., file open modes or permissions. For each system call argument type, a configuration file specifies the threshold point when aggregation will be used, and the aggregation operation to be used for that type. For file name arguments, the threshold can be specified as a function of the file name prefix. For instance, we can set a threshold of 2 for files of the form `/tmp/*`, while using a threshold of 10 for files of the form `/etc/*`.

We point out that the use of the basic FSA approach, and in particular, the use of program location information, is crucial for the success of this approach. Without this information, we could potentially be forced to summarize argument values across all system calls with the same name. Alternatively, we may try to partition system calls with the same name into subsets that yield good aggregation, but such subset construction algorithms will likely be expensive. If the algorithm needs to incorporate relationships among the arguments of a single system call, e.g., the fact that a certain file name is always opened in read-only mode, then the subset construction will become even more complex. In effect, the program location information, provides an efficient and effective way to construct such subsets. Its effectiveness stems from the fact that system calls made by the same point of code in a program are more likely to be related than those made from different program locations.

3.2.3 Learning Argument Relationships

The most interesting aspect of model extraction is our approach for learning temporal relationships between arguments of *different* system calls. We observed that such relationships are crucial for tracking many security-related properties. For instance, in order to relate a `write` system call to the file being written, we need to associate the file descriptor argument of `write` with the return value of a previous `open` system call. Similarly, to track the remote location from which data was read by a process, we need to associate the socket argument of a `recv` or `read` with the return value of a preceding `accept` or argument of `connect`. Finally, to identify the child process to which a signal is being sent by a parent process, one needs to relate the return value of `fork` with the return value of `wait`.

One of the main difficulties in learning system call argument relationships is in identifying which pairs of system calls need to be considered. A naive approach, which considers every possible pair, will be unacceptably inefficient. Such an algorithm will have complexity that is quadratic in the size of the trace, which is typically of the order of 10^3 to 10^7 events, depending upon the comprehensiveness of the test suites used in generating the traces. Even worse, such an approach can generate relationships that are quadratic in the size of the trace. However, we would like the number of relationships learned to be of the same order as the size of FSA, which is typically in the range of a few hundred states.

To overcome the above difficulties, we rely on the observation that we are typically interested in specific relationships among arguments of the same kind. For instance, we are interested in the equality relationship between file descriptor arguments, but not in inequalities or other relationships. Moreover, it is meaningless to compare file descriptors with process identifiers. Based on this observation, our approach is to specify, through a configuration file, the "kind" of a system call argument, and the relationships of interest involving arguments of this kind. (Note that return values are treated as if they are additional arguments to a system call.) In our implementation, we currently support equality relationships among integral and string types, and prefix and suffix relationships over strings.

Once the relationships of interest are specified, they can be learned as follows. First, a distinct (EFSA) state variable is associated with each (system call, invocation location, argument number) triple. Note that because of the way system calls are traced back to locations in the executable, multiple system calls that are made from a library function f invoked by the executable at location ℓ will all appear as transitions from the state corresponding to ℓ. Thus it is possible to have multiple system calls that are all executed from the same location ℓ, and hence the need to consider system call numbers in addition locations. (This means that there will be two distinct variables corresponding to the file name argument of an `open` system call that are made from two different locations in the program.)

Each variable that is a candidate for an equality relationship is stored in a hash table, indexed by its most recent value. The hash tables for different kinds of arguments will be different, e.g., a separate hash table will be maintained for file descriptors and process ids. At any point during learning, associated with each file descriptor value fd will be the list of variables (of file descriptor type) whose most recent value was fd. When another system call with a file descriptor variable v with value fd' is made, the learning algorithm will look up fd' in the table, and obtain the associated list V of variables. If this is the first time v has been seen, then the relationship information associated with v is set to V. This indicates that every variable in V is equal to v. If it is not the first time, then there will already be a set V' of variables that was associated with v the last time v was encountered during learning. We associate $V \cap V'$ with v and proceed. (This means that the relationships may weaken over many runs, but cannot be strengthened. Finally, the previous value fd_{old} of v is deleted from the hash table, and v is added to the hash table entry for fd'.)

For prefix and suffix relationships, a trie data structure is used in place of the hash table. (A trie can be viewed as a tree-structured finite-state automaton for matching strings.) In particular, when a variable v is encountered in the trace with value s, we traverse down a path in the trie that matches s. If there is a complete match for s, and this match takes us to a state S in the trie, then the variables associated with S are candidates for equality with v. Each variable v' associated with any descendant state S' of S is a candidate for the relationship $prefix(v, v') = v$ and $prefix(v, v') = s$. Similarly, any variable v'' associated with an ancestor state S''

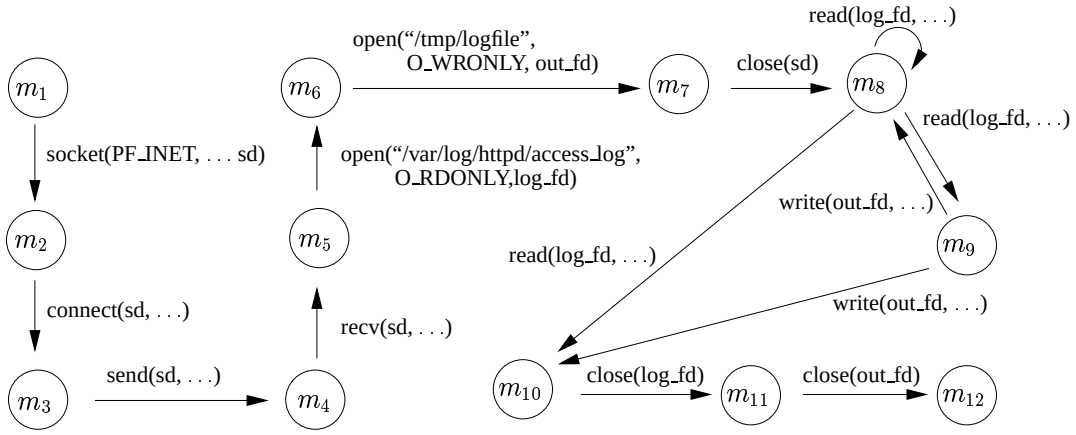


Figure 4: Model EFSA for Figure 5

of S is a candidate for the relationship $prefix(v, v'') = v''$ and $prefix(v, v'') = s''$, where s'' is the string corresponding to state S''. Finally, any variable v''' associated with a descendant state S''' of an ancestor state of S (such as S'') is a candidate for the relationship $prefix(v, v''') = s''$. If only a (possibly empty) prefix of s is present in the trie, then the treatment is similar, except that there will be no descendant states (such as S') mentioned above.

Once the candidates for relationship with the current instance of v are identified, they are compared with the candidates for the previous occurrence of v in the trace, and only the common relationships are preserved. At this point, note that v would be stored in a state S_{old} which corresponds to its previous value s_{old}. v is then deleted from S_{old}, and inserted into S. The state S_{old} is deleted if it is no longer associated with any variables, and the same is done for the ancestors of S_{old}. The new state S is created if it is not already present.

For suffix relationships, the exact same algorithm is used, but the tries are constructed after reversing the strings. In addition, to improve the speed of the algorithm, we can restrict the lengths of paths from S to states S', S'' and S''' described above.

The final step of the algorithm is to prune redundant relationships. Suppose that a program whose current location is ℓ_0 opens a file, and performs read operations on this file from n different program locations $\ell_1, ..., \ell_n$. Let $x_0, x_1, ..., x_n$ be the corresponding state variables. The above algorithm will associate the set $\{x_1, ..., x_n\}$ with x_0, $\{x_0, x_2, ..., x_n\}$ with x_1 and so on. Obviously, this is redundant information — for instance, we can associate $\{x_0\}$ with x_1, $\{x_1\}$ with x_2 and so on. Note that such pruning is difficult to perform during the learning phase itself. This is because premature pruning can lose information. For instance, the first two occurrences of x_2 may have been equal to both x_1 and x_0, but subsequent occurrences may be equal only to x_0. If the relationship was pruned pre-maturely, it is possible to have retained $\{x_1\}$ with x_2. In this case, when it is subsequently discovered that x_1 is not equal to x_2, this relationship is lost, and thus we are left with no relationships involving x_2.

We use the example shown in Figure 5 to illustrate model extraction. This program is a simplified version of a hypothetical freeware program which analyzes web server logs for unusual activity. (Our experience with a *real* program that analyzes web logs is described in Section 6.) In particular, the log entries are compared against signatures of known attacks. Since the signature set is constantly updated as new attacks are discovered, it is better for the analyzer program to download these signatures from a central server rather than encoding them within the analyzer program. Hence, the first step in the execution of the example program is to connect to this signature server over the network, and download a set of signatures. It then opens the log file, and matches each line in the log file with the signatures. To simplify the example, we have used just a single pattern as a signature. In addition, we do not check error cases. Any matches are written into an output file. The lines of code where system calls are made by the program are marked with the symbol "◀" in Figure 5.

Figure 4 shows an abstracted version of the EFSA learned by the above algorithm for the example program. The abstracted details include the following. The learning algorithm makes a number of system calls at program start up, which correspond to calls made

```
1. int main(int argc, char *argv[]) {
2.     int sd, rc, i, log_fd,out_fd,flag = 1;
3.     struct sockaddr_in remoteServAddr;
4.     char recvline[SIG_SIZE+1], *request = "...";
5.     char buf[READ_SIZE];

6.     init_remote_server_addr(&remoteServAddr,...);
7.     sd = socket(PF_INET,SOCK_STREAM,0); ◀
8.     connect(sd,(struct sockaddr*)&remoteServAddr,sizeof(...)); ◀
9.     send(sd, request, strlen(request)+1,0); ◀
10.    recv(sd, recvline, SIG_SIZE,0); ◀
11.    recvline[SIG_SIZE] ='\0';
12.    log_fd = open("/var/log/httpd/access_log",O_RDONLY); ◀
13.    out_fd = open("/tmp/logfile",O_CREAT|O_WRONLY); ◀
14.    close(sd); ◀
15.    while (flag!=0) {
16.        i = 0;
17.        do {
18.            rc=read(log_fd,buf+i,1); ◀
19.            if (rc == 0) flag =0;
20.        } while (buf[i++] != '\n' && flag != 0);
21.        buf[i]='\0';
22.        if (strstr(buf,recvline) !=0)
23.            write(out_fd,buf,strnlen(buf,READ_SIZE)+1); ◀
24.    }
25.    close(log_fd); ◀
26.    close(out_fd); ◀
27.    return 0;
28. }
```

Figure 5: A Freeware Program for Web Log Analysis

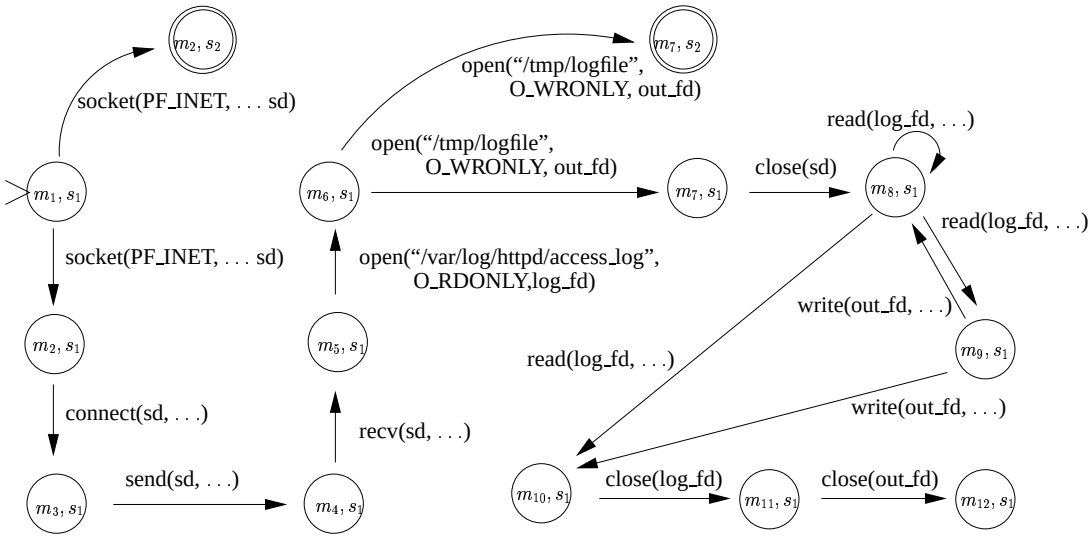


Figure 6: Product Automaton

by the (dynamic) program loader such as `/lib/ld-linux.so`. In addition, the need to allocate heap storage may lead to additional system calls. These details have been abstracted away, and the argument relationships are represented in a human-readable form to obtain Figure 4 from the EFSA learned by the above algorithm. In the figure, each state label ℓ_i corresponds to program line i.

4. VERIFICATION

Verification is concerned with determining whether or not a model M satisfies a security policy P. Formally, we need to check whether behaviors captured by M is a subset of behaviors permitted by the policy P — $\mathcal{B}[M] \subseteq \mathcal{B}[P]$ where M, B and P were introduced earlier. Noting that the policy automaton actually represents the negation $\overline{P}$ of P, we simply need to determine if $\mathcal{B}[M] \cap \mathcal{B}[\overline{P}] = \phi$. Thus, our verification approach is to build the product automaton $M \times \overline{P}$, which will accept the intersection of the behaviors accepted by M and $\overline{P}$. If there are feasible paths in this product automaton that lead to final (i.e., violating) states of $\overline{P}$, then the policy is violated and $M \times \overline{P}$ is a representation of all such violations.

All common operations, such as computing the product of two automata and checking it for reachability, have well-known solutions in the case of FSA, but become complex in the case of EFSA due to the presence of infinite domain variables. We begin by computing the EFSA product in much the same way as an FSA product construction. Specifically, the product automaton $MP = M \times \overline{P}$ is constructed as follows:

- The state variable set of MP is the union of the state variables of M and $\overline{P}$.
- The start state of MP is a tuple (ℓ_0, s_0), where ℓ_0 and s_0 are the start states of M and $\overline{P}$, respectively. Similarly, the final state set is $F_{MP} \subseteq F_M \times F_{\overline{P}}$, where F_M is the set of *all* states in M and $F_{\overline{P}}$ denotes the set of *final* states in $\overline{P}$.
- Whenever there exists a transition from a state ℓ to ℓ' in M on event e with condition C_1 and assignment A_1, and a transition from s to s' in $\overline{P}$ on the same event e with condition C_2 and assignment A_2, then (and only then) there is a transition from (ℓ, s) to (ℓ', s') in MP on condition $C_1 \wedge C_2$ with assignment $A_1 \cup A_2$.

A transition in the product automaton is said to be enabled only when the associated condition $C_1 \wedge C_2$ is satisfiable. Given that our EFSA is defined over infinite-domain variables representing strings and integers, the problem of determining satisfiability of arbitrary constraints appearing as enabling conditions of transitions is undecidable in general. We therefore focus on a tractable subset of constraints over infinite-domain variables; specifically equality ($=$) and disequality ($\neq$) relationships between the variables. The model checker relies on an underlying constraint processing system to decide the satisfiability of these constraints. The constraint processing system maintains a store of conjunctions of constraints between the variables. A product transition is feasible if the corresponding enabling conditions are satisfiable in the existing store present in the constraint processing system. In this case, the constraint store is updated by adding the enabling conditions. Otherwise, the transition is considered infeasible.

As alluded before, EFSA are defined over infinite-domain variables. Hence, the model checker empowered with the constraint solver, as described above, is incapable of inferring the satisfiability of constraints in some cases, e.g. range ($\geq$) constraints over integer variables or prefix constraints over strings. Such situations are handled conservatively as follows. If the arguments to the constraint are sufficiently defined then the constraint processing system evaluates them. Otherwise it considers these "undecided" constraints as satisfiable and adds them to the existing constraint store. This strategy results in the incompleteness of the constraint-based model checker due to the generation of infeasible paths in the product automaton. Such infeasible paths, in general, could lead to spurious policy violation sequences. However, we have not experienced such spurious reports in the programs we have considered so far.

Consider the example program and the corresponding model in Figures 5 and 4 respectively. In order to verify whether the model conforms to the policy (see Figure 2(c)) of no socket and write-to-file operations, a product (Figure 6) of the model automaton and the policy automaton is constructed. Two violating traces are obtained in the model — (a) the transition from (ℓ_7, s_1) to (ℓ_8, s_2) in the model consisting of `socket(PF_INET,...,sd)` and (b) the transition sequence from (ℓ_{13}, s_1) to state (ℓ_{14}, s_2) due to the `open` operation of a file in `O_WRONLY` mode.

4.1 Conflict presentation

One important aspect of the verifier is to give a comprehensive view of why/how a violation of the policy occurred. This information is crucial for policy selection.

Owing to the size of the product (which is of the order of the size of the model), presentation of the product "as is" does not provide a clear and precise view of the violations in the model. The product automaton is hence presented to the user by projecting it onto the policy automaton, because the root cause of policy violation (leading to a final state of the policy automaton) can be attributed to the sequences of policy-specified actions that are present in the model.

Note that the product contains all violating paths in the model. During projection, we combine common aspects of multiple violating paths. Frequently, this combination requires merging transitions that are associated with different conditions on event arguments. For instance, `open` events corresponding to opening different files may all need to be combined. We use an approach similar to the model extraction algorithm for doing this combination: If the number of different argument values is small, we retain the set of possible values. If it exceeds a certain threshold, then they are combined using an appropriate aggregation technique. For instance, the file names { `/tmp/a1`, `/tmp/a2`, `/tmp/a3`, `/etc/xyz`, `/var/f1`, `/var/f2` } may be combined into { `/tmp/a*`, `/etc/xyz`, `/var/f1`, `/var/f2` }. Using this approach produces the following summary of conflicts from Figure 6:

- `open` operation on file `/tmp/logfile` in write mode,
- `socket` operation involving the domain `PF_INET`

The refinement for the first violation is relatively obvious — the user can simply permit write access to files in `/tmp`. For the second violation, we relax the policy to permit network access, as long as these accesses are completed before reading any sensitive files. The list of sensitive files needs to be specified, but we can assume that any file that the consumer does not consider to be "public" is classified as sensitive. In particular, this means that the web log file is considered sensitive. The new policy with these refinements is shown in Figure 7.

The ability of the conflict presentation technique to summarize the violations provides help to a user in identifying suitable policies. One approach that can provide additional assistance to a consumer in policy selection is based on catalogs of acceptable policies. Given such a catalog, the verifier can search this catalog to identify a policy that is compatible with a given piece of code. The conflict summary can provide direction to this search, so that the verifier does not have to consider all policies in the catalog.

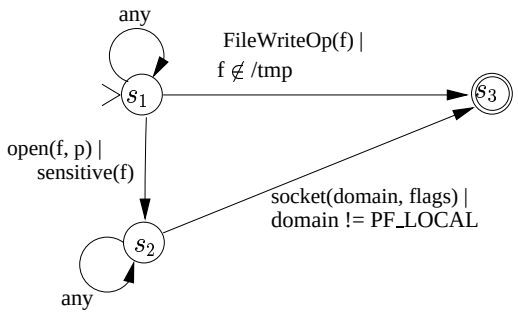


Figure 7: Refined Policy

5. ENFORCEMENT

Runtime monitoring consists of intercepting system calls, obtaining the argument values to these system calls, and matching them against models of expected behavior of the untrusted application. Recall that we enforce models (not policies), which are large nondeterministic automata. To avoid having to simulate the nondeterminism, which can lead to high overheads, we simply use the program counter value to determinize the transitions. Policy enforcement based on system call interception is a well-understood topic [18, 33, 17], so we do not describe it any further here.

If the application violates the behavior captured by the model, the enforcement module aborts the program. When this happens, there are only two possibilities:

- producer intentionally misrepresented the program behavior, or
- the model does not capture all possible program behaviors. (This can happen when models are constructed through runtime monitoring, but the test cases used for these runs were not sufficiently comprehensive.)

In the first case, termination is the right choice. Since the second case is indistinguishable to a consumer from the first case, the only safe response is to terminate the application in this case as well.

While runtime aborts cause inconvenience, we point out that safety is not violated, since the program is aborted before it violates a security policy. Moreover, either of the above two cases are likely to be rare, given (a) the predominance of benign software in the Internet, and (b) the fact that a consumer, in a single run, is not very likely to uncover new behaviors unless the code producer did a very poor job of testing his/her code.

Note that within the MCC framework, it is possible to enforce policies, rather than models. However, since the model EFSA captures a subset of behaviors permitted by the policy EFSA, some behaviors that would violate the model EFSA would go undetected during policy enforcement. It is questionable whether a consumer wants to allow such behaviors, which the producer, in some sense, has identified as illegitimate. For instance, consider an untrusted application that reads image files, and saves them back with a preview image included in the file. A policy that allows execution of this application needs to grant write-access to such a file. However, it is possible that a third-party attacks this program and causes it to simply delete the image file. In this case, previous experience in applying such models in intrusion detection [32, 39] suggests that it is very likely that such attacks will be prevented by model enforcement, while it will be missed by policy enforcement.

A possible benefit of policy enforcement is that it may be simpler than model enforcement. Even though model EFSA are much larger than policy EFSA, they are deterministic, and hence can be efficiently enforced. The enforcement algorithm is also simple, as it simply needs to (a) keep track of the most recently encountered value of every state variable, and (b) before making a state transition, perform the relationship checks associated with the transition. Policy automata, although smaller, are nondeterministic. This factor requires them to potentially keep an unbounded number of previous values encountered for a state variable, and perform checks with respect to each such value. Although we have not encountered this worst case behavior in our previous work in intrusion prevention [33, 4], the mere possibility shows that policy enforcement is not necessarily simpler than model enforcement..

Another possible reason for favoring policy enforcement over model enforcement is that the soundness of the approach will no longer depend on the correctness of the verifier. However, since the verifier is already very simple, we believe that the reduction in the size of the trusted computing base achieved as a result of such a choice is not compelling.

Application	Program Size (KB)	Model Size			Enforcement Overhead		Verification	
		States	Transitions	Relationships	Interception only	Total	Time (msec.)	Space (MB)
xpdf 1.0	906	125	455	305	2%	30%	1.00	0.5
gaim 0.53	3173	283	937	432	2%	21%	1.80	0.7
http-analyze 2.4.1.3	333	158	391	247	0%	2.4%	0.70	0.4

Figure 8: Results on Generation, Verification and Enforcement of Models

6. IMPLEMENTATION

In this section, we summarize the status of our implementation, and describe the results of applying MCC to some common programs such as instant messengers, web log analyzers, and document viewers. These programs range in size from a few thousand lines to tens of thousands of lines of source code.

Security Policies. As mentioned previously, security policies are specified in our BMSL language [4]. A compiler for this language, which produces EFSA from BMSL specifications, has been described in an earlier paper [33]. As shown in the policy examples discussed so far, policy automata tend to be very small (2 to 6 states).

Model generation. We have implemented model generation using execution monitoring. Our implementation learns system call argument values as described in Section 3.2.2. Argument relationship learning has been implemented for equality relationships for arguments such as file descriptors, and prefix/suffix relationships for strings such as pathnames. The sizes of the models, in terms of number of states, transitions and relationships, are shown in Figure 8.

In terms of the time needed to learn the models — we first note that model generation is an offline process, hence we have not attempted to optimize it. Currently, the logging of system calls is done using user-level system call interception, built with the `ptrace` facility on Linux. The frequent context switches involved in this approach, together with the need to fetch system call argument data, introduces significant overheads, which range from 40% to 200%. However, since learning itself is an offline activity, this overhead is quite acceptable. The logger and system call interceptor are implemented in C and C++ and have a combined size of 9 KLOC.

Different applications tend to generate different volumes of system calls. For instance, `xpdf` and `gaim` generate a large number of system calls, while `http-analyze` generates much fewer calls in each run. These two factors balance each other out. We used training traces in the range of 10^3 to 10^5 system calls for all programs.

The current implementation of model extraction is not optimized for performance, since model extraction is an offline process. For this reason, model extraction is relatively slow, taking of the order of few minutes for traces of size of the order of 10^5 system calls on a machine with a 800 MHz Pentium III processor running Red Hat Linux 7.3 with 384 MB of memory. The size of the learning algorithm implementation is about 6000 lines.

Verifier. The verifier is implemented in XSB [41] Prolog, a version of Prolog that supports "memoization" of query results using a technique called tabling. Tabling avoids redundant subcomputations — instead, previously saved results of queries are reused. This factor greatly simplifies the implementation of verification and program analysis techniques, which involve so-called *fixpoint* computation. The analysis can be specified using a set of recursive rules, leaving the tabling technique to automatically compute the fixpoint. This facility enables the verifier to be implemented using about 300 lines of Prolog code. This code implements the automata product construction, as well as the constraint-handling operations described in Section 4. The constraint processing system interprets equality constraints using Prolog's variable unification mechanism and handles disequality constraints by storing them in the form of a list.

We have verified the application models described above using policies similar to the ones described in Section 2. Here, we present a brief description of the policies corresponding to each of the applications. Figure 8 tabulates the results of verification.

- *PDF viewer* application. We use conventional sandboxing policies on the PDF viewer application. Such a policy prevents the application from creating any new files (except in the `/tmp` directory), disallows it from overwriting files, restricts network connections and prevents it from executing other applications. No violations were reported for these policies. Thus, for applications such as document viewers, it appears that MCC is as simple to use as sandboxing approaches.
- *http-analyze* application. The http-analyze application [22] is a real application which is similar to the abstract log analyzer example discussed earlier. We generated the model of the program by learning its behavior and verified it against the refined policy shown in Figure 7. Recall that this policy disallows the application from (a) writing to non-temporary directories, and (b) performing network operations after reading sensitive files. The second part of the policy is not violated by the model. The first part of the policy is violated, since the application creates a file called `index.html` in the current directory, and two subdirectories called `btn` and `www`, and several files in each of these subdirectories. The model extraction process summarizes this information, so that the following violations are reported by the verifier:
  - attempt to create directories: `btn` and `www`
  - attempt to write files: `index.html`, `btn/*` and `www/*`

  The policy is refined to permit these accesses, and then the application is run successfully.
- *Gnu instant messaging* application. The model for Gaim application is verified against a "no file access" policy that disallows reading/writing of any files except for those that are commonly accessed by graphical user interface applications. The policy is violated by the model. Projecting the product on the policy, a violating trace in the model is obtained — an `open` operation is performed on `.gaim` (read/write mode), `.imrc` (read mode) and `.gaimrc` (write mode) files in the user's home directory. The policy is relaxed by restricting file access of the user's home directory to only `.gaim`, `.gaimrc` and `.imrc`. The refined policy is not violated by the model.

All these experiments were conducted RedHat Linux 7.2 running on a 1.7GHz Xeon with 2GB of memory. As figure 8 shows, verification takes only milliseconds, and has low memory requirements, thus making it practical.

Model enforcement. The enforcement system uses an in-kernel module to perform system call interposition. Whenever the application performs a system call, the enforcement module makes the corresponding transition on the model automaton, thereby keeping track of the system state. Figure 8 shows that the runtime overheads for model enforcement are moderate— 2% to 30%. Much of this overhead arises from the stalk walk required to obtain the program location from where system calls are made. Often, this requires 10 or more stack frames to be traversed. Our current stack traversal algorithm is conservative, and cross-checks every return address found on the stack with the calling instruction. Moreover, no systematic performance tuning has been attempted yet. These factors lead to the moderate overhead. With improved implementation of the stack walk and performance tuning, these overheads may be cut down by a factor of two or more.

The variation in overheads across applications results from the variations in the frequencies of system calls made by these applications. The http-analyze application performs very few system calls, whereas xpdf and gaim make a large number of system calls.

6.1 Discussion

We make the following remarks based on our implementation experience so far.

6.1.1 Usability and Practicality

As illustrated with the `xpdf` application, for applications that are amenable to sandboxing types of policies, MCC seems to be as simple to use as execution-monitoring approaches. Indeed, the added expressive power of MCC policies can be expected to allow more applications to execute without raising violations during either the verification or the runtime stage. For instance, consider a policy that permits an untrusted application to create new directories, and to overwrite files in the directory created by it. Clearly, such a policy requires the use of state variables to remember the name of directories created by the untrusted application, and hence can be expressed as an EFSA, but not using the policy languages used in previous execution monitoring-based approaches.

The `http-analyze` example demonstrates the effectiveness of MCC in minimizing policy violation alerts. A naive execution monitoring-based approach would have resulted in close to 100 runtime prompts, corresponding to each file created by this application. Even a more intelligent system, which requests write access for entire directories after a number of violations (say, 3) have been reported for files in that directory, would result in 7 user alerts. It is our experience that users "give up" after perhaps 3 such prompts, and either discard the code, or click "yes to all."

While MCC improves over previous techniques in offering some guidance for policy refinement, there is much room for improvement. One possibility that we plan to consider in the future is that of having the verifier automatically search through a hierarchy of safety policies to find one that is suitable for a particular application. In addition, we need to improve the understandability of violations of security policies that involve nontrivial temporal relationships.

These comments on the usability of MCC, together with performance results reported above, validate our claim that MCC is indeed practical.

6.1.2 System complexity

System complexity is an important consideration in security, as complexity leads to errors that can impact security. A careful examination of MCC shows that although it is realized using several components, including a policy language compiler, model generator, verifier and model enforcer, each of these components is relatively simple. Where there was a choice between simplicity and generality, we have usually favored simplicity, in order that we be able to build a reasonably robust system with modest implementation resources. Even the verifier, which is often considered a complex piece of software, is very simple in MCC — only 300 lines of code, written in a declarative language. The compiler is of moderate size (15 KLOC), but much of this complexity arises from the fact that BMSL is designed to be a general purpose event monitoring language that is capable of monitoring diverse events, and complex data types such as network packet data. If one were to separate the code that would be needed for EFSA policies of the kind described in this paper, then it would perhaps be half the current size. The model enforcer is also simple, consisting of only 2500 lines of code. The model extractor, which consists of the implementation of the learning algorithm, the logger and the system call interceptor, is about 15000 lines of code. Thus the total system size is of the order of 33 KLOC.

Not all of the MCC components are critical for security. In particular, the correctness of the model extractor does not impact the safety of MCC. If the implementation of other components is tightened up to eliminate features unneeded for MCC, then the size of security-critical components of MCC can be brought down to below 10 KLOC. On the other hand, it should be noted that safety depends on many non-MCC components, including the underlying OS, the gcc compiler and related tools, and the XSB system. All these components must be considered part of the trusted computing base, in addition to the BMSL compiler, verifier and model enforcer.

6.1.3 Standardization

In order to enable the widespread deployment of MCC or a similar approach, a significant amount of work in standardization will be required in addition to technical solutions. Currently, the policy language uses events that are closely tied to an operating system. Moreover, several choices are made regarding which system call arguments are important, and what relationships involving them need to be preserved in a model. These need to be standardized as well, if we are to achieve interoperability and compatibility across different OSes and/or users. Finally, note that many policies need to be parameterized. For instance, we need to express the fact that a certain application may open a file in the consumer's home directory. Developing a uniform way to identify such parameters, and naming them is another important aspect of standardization. (Once such parameters are standardized, it is simple to incorporate them into MCC — the logging component of the model generator needs to ensure that system call arguments such as filenames are expressed in a parameterized form. The policies also need to be stated in a parameterized form.)

6.2 Integration into Existing Systems

It is our objective that typical users should not be required to change their ways in order to benefit from MCC. To accomplish this objective, we integrate MCC into the tools that are used in the process of explicit installation, implicit downloading, or execution of untrusted code.

Explicit installation of code. We have incorporated the MCC approach into a tool called *RPMshield* [37], an enhancement of the

RedHat Package Manager (RPM) software installation tool. This tool applies the MCC approach to the installation phase, wherein it is ensured that the installation of a package does not clobber files belonging to other packages, and that the installation scripts observe consumer-specified policies. Some of the concrete problems addressed by this enhancement include: clobbering of manually edited configuration files during package upgrades, and execution of arbitrary pre- and post-installation scripts. In the case of untrusted packages, these scripts can cause arbitrary damage, whereas with RPMshield, they cannot.

During installation, an untrusted application is associated with a set of allowable policies. In addition, it is flagged to indicate whether these policies should be silently enforced (which would be the case if the policy selection took place during installation) or whether the MCC user interface is to be invoked for policy selection during each run.

Implicitly downloaded code. The two principal mechanisms for implicit code downloading are email attachments, and web content. We integrate the MCC approach into these environments by defining a new content type, `application-mcc`, corresponding to model-carrying code. The MCC policy selection user-interface is then invoked for handling instances of such content, ensuring a smooth integration with diverse email readers and browsers.

Execution of untrusted code. The above installation and/or downloading process ensures that untrusted code will always be executed under MCC control. In particular, execution occurs within the secure enforcement environment, and with or without the support of the policy selection user-interface.

Untrusted code without a model. To facilitate the adoption of MCC, our user interface supports the execution of code without accompanying models. In such cases, a consumer-specified policy is silently enforced on the application. This may not be convenient or suitable for all applications, but it certainly works well for applications such as document handlers. Alternatively, third parties could generate models for such programs and these models could be downloaded and used for policy selection.

7. CONCLUSION

In this paper, we have presented model-carrying code, a promising solution to the problem of running untrusted code. Unlike previous approaches that were focused mainly on malicious code containment, MCC makes no assumptions on the inherent risks posed by untrusted code. It simply provides tools that consumers can use in order to make informed decisions about the risks that they are willing to tolerate so as to enjoy the functionality provided by untrusted code.

We demonstrated that MCC is practical by showing that several small to moderate size programs can be successfully handled. MCC does not require users to switch to a new programming language, nor does it require them to change their ways in terms of how they download or run untrusted code. Instead, MCC is incorporated in a transparent fashion into tools that serve as conduits for untrusted code, including software installers, email handlers, and browsers.

MCC achieves a practical balance between the security-related obligations of the producer and the consumer, thereby avoiding placement of an undue burden on either party. The producer can generate a model for an application and supply it to several consumers with different security concerns. Similarly, consumers can develop and enforce policies that address their security concerns without having *a priori* knowledge about security needs of diverse applications. Thus, MCC provides a scalable framework for permitting the networked distribution of end-user software applications while addressing security concerns.

The MCC approach is complementary to existing approaches such as PCC and code-signing, and can be gainfully combined with them. For instance, a code producer may provide a proof of model safety with the code. In this case, the consumer can statically check the correctness of this proof; runtime enforcement of models is unnecessary. Similarly, digital signatures may be combined with MCC by having the producers digitally sign their models to indicate that they certify the safety of the model. If a consumer trusts this representation by the producer, then they can skip the enforcement step. Since model safety does not rely on runtime enforcement in both cases, there is a scope for expanding the classes of properties that can be supported by MCC to include liveness and information flow. Moreover, runtime aborts can be avoided.

Acknowledgments

We thank Yow-Jian Lin for several discussions on model extraction and verification, and for feedback on earlier drafts of the paper; C.R. Ramakrishnan, Scott Smolka and Diptikalyan Saha for several discussions on verification; I.V. Ramakrishnan for overall discussions on MCC and RPMshield; Prem Uppuluri for discussions regarding the policy language and compiler; and Wei Xu for discussions regarding model extraction from source code. We also acknowledge Weiqing Sun for his support with system call interposition environment for model enforcement, Mohan Krishna and Shruthi Murthy for support with the policy compiler, Abhishek Chaturvedi for support with model extraction, Tapan Kamat for support with RPMShield, and Zhenkai Liang for support with system call interposition for model extraction. Finally, we thank the anonymous reviewers for their insightful comments that led to significant improvements in the content and organization of this paper.

8. REFERENCES

[1] Anurag Acharya and Mandar Raje. Mapbox: Using parameterized behavior classes to confine applications. In *USENIX Security Symposium*, 2000.

[2] G. Ammons, R. Bodik, and J.R. Larus. Mining specifications. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2002.

[3] Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In *Computer Aided Verification CAV*, New York-Berlin-Heidelberg, July 2001.

[4] T. Bowen, D. Chee, M. Segal, R. Sekar, T. Shanbhag, and P. Uppuluri. Building survivable systems: An integrated approach based on intrusion detection and damage containment. In *DARPA Information Survivability Conference (DISCEX)*, 2000.

[5] Guillaume Brat, Klaus Havelund, SeungJoon Park, and William Visser. Java pathfinder: Second generation of a Java model checker. *Post-CAV 2000 Workshop on Advances in Verification*, 2000.

[6] C.C Michael and Anup Ghosh. Simple, state-based approaches to program-based anomaly detection. *ACM Transactions on Information and System Security (TISSEC)*, 5(3), 2003.

[7] Hao Chen and David Wagner. MOPS: an infrastructure for examining security properties of software. In *ACM conference on Computer and Communications Security (CCS)*, 2002.

[8] E M Clarke, E A Emerson, and A P Sistla. Automatic verification of finite-state concurrent systems using temporal

logic specifications. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 8(2), 1986.

[9] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering (ICSE)*, 2000.

[10] Daniel C. DuVarney and S. Purushothaman Iyer. C Wolf — a toolset for extracting models from C programs. In *International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*, 2002.

[11] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[12] Ulfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *New Security Paradigms Workshop (NSPW)*, 1999.

[13] Ulfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, 2000.

[14] David Evans and Andrew Tywman. Flexible policy directed code safety. In *IEEE Symposium on Security and Privacy*, 1999.

[15] Henry Hanping Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly detection using call stack information. In *IEEE Symposium on Security and Privacy*, 2003.

[16] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for UNIX processes. In *IEEE Symposium on Security and Privacy*, 1996.

[17] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS software with generic software wrappers. In *IEEE Symposium on Security and Privacy*, 1999.

[18] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications: confining the wily hacker. In *USENIX Security Symposium*, 1996.

[19] L Gong, M Mueller, H Prafullchandra, and R Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java development kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[20] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Grégoire Sutre, and Westley Weimer. Temporal-safety proofs for systems code. In *Computer Aided Verification CAV*, 2002.

[21] G.J. Holzmann and Margaret H. Smith. Software model checking - extracting verification models from source code. In *Formal Methods for Protocol Engineering and Distributed Systems*, Kluwer Academic Publ., 1999.

[22] Http-analyze application. Available from http://www.http-analyze.org/.

[23] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10), 1973.

[24] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium*, 2001.

[25] Wenke Lee and Sal Stolfo. Data mining approaches for intrusion detection. In *USENIX Security Symposium*, 1997.

[26] David Lie, Andy Chou, Dawson Engler, and David L. Dill. A simple method for extracting models from protocol code. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001.

[27] Christoph Michael and Anup Ghosh. Using finite automata to mine execution data for intrusion detection: A preliminary report. In *Recent Advances in Intrusion Detection (RAID)*, 2000.

[28] Andrew C Myers and Babara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering Methodology*, 1999.

[29] G Necula. Proof-carrying code. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1997.

[30] L. Pitt and M. Warmuth. The minimum consistency DFA problem cannot be approximated within any polynomial. In *ACM Symposium on Theory of Computing (STOC)*, 1989.

[31] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1), 2001.

[32] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati. A fast automaton-based approach for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*, 2001.

[33] R Sekar and Prem Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *USENIX Security Symposium*, 1999.

[34] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format-string vulnerabilities with type qualifiers. In *USENIX Security Symposium*, 2001.

[35] Prem Uppuluri. *Intrusion Detection/Prevention Using Behavior Specifications*. PhD dissertation, Department of Computer Science, Stony Brook University, 2003.

[36] V.N. Venkatakrishnan, Ram Peri, and R. Sekar. Empowering mobile code using expressive security policies. In *New Security Paradigms Workshop (NSPW)*, 2002.

[37] V.N. Venkatakrishnan, R. Sekar, S. Tsipa, T. Kamat, and Z. Liang. An approach for secure software installation. In *USENIX System Administration conference (LISA)*, 2002.

[38] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE symposium on security and privacy*, 2001.

[39] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *ACM conference on Computer and Communications Security (CCS)*, 2002.

[40] Andreas Wespi, Herv Debar, Marc Dacier, and Mehdi Nassehi. Fixed- vs. variable-length patterns for detecting suspicious process behavior. *Journal of Computer Security (JCS)*, 8(2/3), 2000.

[41] XSB. The XSB logic programming system v2.3, 2001. Available from `http://www.cs.sunysb.edu/~sbprolog`.

[42] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2001.