Automated Discovery of Mimicry Attacks

Jonathon T. Giffin, Somesh Jha, and Barton P. Miller

Computer Sciences Department, University of Wisconsin
{giffin,jha,bart}@cs.wisc.edu

Abstract. Model-based anomaly detection systems restrict program execution by a predefined model of allowed system call sequences. These systems are useful only if they detect actual attacks. Previous research developed manually-constructed mimicry and evasion attacks that avoided detection by hiding a malicious series of system calls within a valid sequence allowed by the model. Our work helps to automate the discovery of such attacks. We start with two models: a program model of the application's system call behavior and a model of security-critical operating system state. Given unsafe OS state configurations that describe the goals of an attack, we then find system call sequences allowed as valid execution by the program model that produce the unsafe configurations. Our experiments show that we can automatically find attack sequences in models of programs such as `wu-ftpd` and `passwd` that previously have only been discovered manually. When undetected attacks are present, we frequently find the sequences with less than 2 seconds of computation.

Key words: IDS evaluation, model checking, attacks, model-based anomaly detection

1 Introduction

A model-based anomaly detector restricts allowed program execution by a predefined model of acceptable behavior [6,8,12,14,19,23]. These systems compare a sequence of system calls generated by the executing program against the model. The detector classifies any system call sequence that deviates from the model as malicious and indicative of a program exploit. The ability of the model to detect actual attacks depends upon the implicit assumption that attacks always appear different than valid execution.

An attack that is accepted by the model as valid will not cause an anomaly and will not be detected (Fig. 1). Mimicry and evasion attacks avoid detection by transforming an attack sequence of system calls so that it is accepted by a program model yet still carries out the same malicious action. Previous research found examples of mimicry attacks against high-privilege processes restricted by a model-based detector [20–22,24]. However, the attacks were constructed manually by iterating between an attack sequence and a program model until the attack could be made to appear normal. Although these manually-constructed attacks served as a successful proof-of-concept, manual approaches remain unsuitable as a general attack discovery strategy.

This paper automates the discovery of mimicry attacks. Our intent is not to propose a new detection system but rather to provide the means to evaluate an existing program model's ability to detect attacks. We address two primary questions:

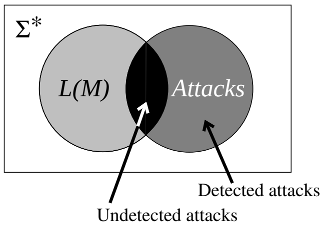


Fig. 1. If Σ is the set of system calls, then Σ^* is the infinite set of all possible system call sequences. A program model M accepts a subset of system call sequences $L(M)$ as valid program execution. Any attack sequence accepted as valid is a missed attack.

- What attacks does a program model fail to detect?
- What attacks can we prove that a model will always detect?

Finding missed attacks reveals the weaknesses of a program model and indicates that a model-based detector provides insufficient security for that particular program. Conversely, proving that a model always detects an attack establishes strong indications that a computer system using model-based detection is secure, even when an attacker attempts to hide an attack within legitimate execution.

An attack is any sequence of system calls that produces a malicious change to the operating system (OS). For a given attack sequence, an attacker can produce variations of the sequence having the same attack effect by inserting extraneous system calls into the sequence or replacing existing system calls with alternative sequences having the same effect. A program model that detects one sequence may allow a different, obfuscated sequence. The net result remains the same: the model fails to detect an attack. We must verify that a model detects each of the attack variants.

We use a novel formalism that requires neither knowledge of particular attack sequences nor knowledge of particular obfuscations that try to hide those sequences from a detector. We develop a model of an OS with respect to its security-critical state and then characterize attacks only by their effect upon the OS. This leverages a key insight: *the commonality among the obfuscated attack sequences is that the sequences are semantically equivalent with respect to their malicious effect upon the OS*. Although we manually produce the OS model and the definitions of malicious OS state, this is a one-time effort that is reused for subsequent analyses of all models of programs executing on that operating system.

The program model specifies what sequences of system calls are allowed to execute. By specifying how each system call transforms the OS's state variables, we are able to compute the set of OS configurations reachable when a program's execution is constrained by the model. We apply model checking [4] to prove that no reachable configuration corresponds to the effect of an attack. If the proof fails, then some system call sequence allowed by the model produces the malicious effect. The model checker reports this sequence as a counter-example that caused the proof to fail, providing precisely an undetected attack as output. In terms of Fig. 1, we are finding system call sequences contained in $L(M) \cap Attacks$ without explicitly computing the set $Attacks$ of malicious system call sequences.

This approach automates the previous manual effort of finding mimicry attacks. In experiments, we show that we can automatically discover the mimicry attack against the Stide [8] model for `wu-ftpd` [24] and the evasion attacks against the Stide models for `passwd`, `restore`, and `traceroute` [20–22]. The model checking process completed in about 2 seconds or less when undetected attacks were present in the models. When a model is sufficiently strong to detect an attack, the model checking algorithm will report that no attack sequence could be found. This requires exhaustive search and completed in 75 seconds or less for all attacks detected by the models of the four test programs. Note that proofs of successful detection hold only with respect to our abstraction of the OS state. If this abstraction is erroneous or incomplete, undetected attacks may still be present when using the model to protect a complete operating system.

Our work addresses outstanding problems in model-based anomaly detection. We provide a method for model evaluation that exhaustively searches for sequences of system calls allowed as valid by a program model but that induce a malicious configuration of OS state. Although our current work evaluates the context-insensitive Stide model, we have designed our system so that it can evaluate any program model expressible as a context-sensitive pushdown automaton (PDA). One of our long-term goals, not yet realized, is to compare the detection capabilities of different model designs proposed in the literature.

In summary, this paper makes the following contributions:

- *Automated discovery of mimicry attacks.* We use model checking to find sequences of system calls accepted as valid by a program model but that have malicious effects upon the operating system. Our system produces the exact sequences of system calls, with arguments, that comprise the undetected attacks.
- *A system design where attack sequences and obfuscations need not be known.* Our system does not require that attack system call sequences be known or enumerated. In fact, we strive for the opposite: our system will automatically find new, unknown attack sequences accepted by a program model and will produce those sequences as output. Likewise, we automatically find the obfuscations used by attackers to hide attack system calls within a legitimate sequence. As a result, our approach is not limited by *a priori* knowledge of attacker behavior.

Section 2 presents related work in manual attack analysis. Section 3 gives an overview of our system. Section 4 describes the operating system abstraction and Sect. 5 explains how a model checker uses that abstraction to find undetected attacks in a program model. Section 6 presents the architecture of our implementation, and Sect. 7 uses that implementation to demonstrate experimentally that we have automated the previously manual process of discovering undetected attacks.

2 Related Work

The seminal research on mimicry [9, 24] and evasion attacks [20–22] demonstrated a critical shortcoming of model-based anomaly detection. Attackers can avoid detection by altering their attacks to appear as a program's normal execution. These altered attacks are sequences of system calls allowed by a program model but that still cause malicious execution. Previous work constructed mimicry and evasion attacks by converting some detected attack system call sequence A into an equivalent undetected sequence A'. If A and A' are semantically equivalent and A' is a sequence allowed by the program model, then A' is a successful, undetected attack.

Determining that a model expressed as a pushdown automaton accepts A' is a computable intersection operation provided that A' is regular; finding a sequence A' to intersect is a manual, incomplete procedure with several drawbacks:

- The procedure requires known attack sequences A.
- The equivalence of two system call sequences is not well defined. For example: an undetected attack sequence A' may include legitimate execution behavior that is irrelevant to the original attack sequence A. Are A and A' equivalent?
- There is no clear operational direction to find mimicry and evasion attacks automatically. Identifying two sequences as equivalent attacks was a manual procedure based on intuition. There was no algorithmic process amenable to automation.

Our model evaluation takes a different approach that advances the state of the art. By defining attacks only by their malicious effects upon the system, our work is not restricted to known attack sequences of system calls or known attack transformations producing evasive attacks. Attack sequences are not part of the input to our system; in fact, our work produces the sequences as its output. We can further define two system call sequences as equivalent with respect to the attack if they produce the same malicious effect upon the operating system. This formalism provides the operational direction allowing our work to automate the procedure of finding undetected attacks.

Previous attempts have been made to quantify the ability of a model to detect attacks. *Average branching factor* (ABF) [23] calculates, for any finite-state machine model, the average opportunity for an attacker to undetectably execute a malicious system call during a program's execution. A predefined partitioning divides the set of system calls into "safe" calls and "potentially malicious" calls. As the runtime monitor follows paths through the automaton in response to system calls executed by the program, it looks ahead one transition to determine the number of potentially malicious calls that would be allowed as the next operation. The average branching factor is then the sum of the potentially malicious calls divided by the number of system call operations verified during execution. An extension to average branching factor, called the *average reachability measure* (ARM) [10], similarly evaluated pushdown automaton models.

Although these measurements provide a convenient numeric score enabling model comparisons, they do not provide a clear measure of a model's ability to actually detect attacks. These metrics do not effectively embody an attacker's abilities:

- An attacker may alter a program's execution to reach a portion of the program model that admits an attack sequence by first passing through a sequence of safe system calls. By only looking at the first system call branching away from a benign execution path, ABF and ARM fail to show the strength of one model over another.
- The ABF or ARM value computed depends upon the benign execution path followed and hence upon program input. A complete evaluation of the model requires computing the score along all possible execution paths. This is extremely challenging and itself forms an entire body of research in the program testing area.

– Attacks frequently are comprised of a sequence of system calls. The previous metrics look at each system call in isolation and have no way to characterize longer attack sequences.

Consequently, these metrics provide limited insight into a model's ability to detect attacks. Our work improves the evaluation of a program model's attack detection ability by decoupling the evaluation from both a particular execution path and from the need to describe malicious activity as unsafe system calls.

MOPS [3] is similar to our work in the first aspect: it statically checks a program model to determine properties of the model. Unlike our work, however, MOPS characterizes unsafe or attack behavior as regular expressions over system calls and requires users to provide a database of malicious system call patterns. Just as commercial virus scanners syntactically match malicious byte sequences against program code, MOPS syntactically matches unsafe system call sequences against a program model. Likewise, when a new malicious behavior is discovered, the database of system call patterns must be updated. Conversely, by understanding the semantics of system calls, the system in our paper does not require known malicious system call sequences, and it in fact automatically discovers them for the user. Our work is not tied to known patterns of malicious system call execution.

Model checking is a generic technique used to verify properties of state transition systems, and it has been applied previously to computer security. Bessen et al. [2] described how model checkers can verify safety properties [16] expressed in linear-time temporal logic (LTL). They verified the properties over annotated control-flow graphs, where both the graph and the annotations expressing security properties of the program code came from some unspecified source. We analyze automatically constructed program models, and our model checking procedure automatically derives security properties of the model as it traverses the model's edges.

Guttman et al. used model checking to find violations of information-flow requirements in SELinux policies [13]. They modeled the SELinux policy enforcement engine and the ways in which information may flow between multiple processes via a file system. They could then verify that any information flow was mediated by a trusted process on the system. Our work has a different goal: verification of safety properties using an OS model where system calls alter OS state.

Ramakrishnan and Sekar [15] used model checking to find vulnerabilities in the interaction of multiple processes. They abstracted the file system and specified each program's execution as a file system transformer. The program specifications were complicated by the need to characterize interprocess communication. Our work expands the system abstraction to include the entire operating system, shifts the checked interface from coarse-grained process execution down to system calls, and has no need to model communication channels between processes.

Walker et al. used formal proof techniques to verify properties of a specification of a UNIX security kernel [25]. This work is notable because the authors rigorously proved that the abstract specification of the kernel matched the actual implementation. As a result, properties proved using the abstraction also hold true in the real operating system. Due to the difficulty of producing proofs of correct specifications, little other research actually demonstrates that abstractions are accurate. We adopt this simpler approach: we produced our operating system abstraction manually and have not proved it correct. As a result, discovered attacks or proofs of the absence of attacks hold only with respect to the abstraction. A discovered attack can be validated by actually running the system call sequence against a sandboxed operating system. Conversely, if we do not find any attack, then this provides good indication that the program model is secure even though this is not provably true in the real operating system.

3 Overview

We provide here an overview of model-based anomaly detection, including the attacker threats addressed, context-sensitive program models, and the purpose of attack discovery.

3.1 Threat Model

Our system automatically constructs undetected attack sequences possible within a particular threat model. This threat model is simple and strong:

> Let Σ be the set of system calls invoking kernel operations. If program P is under attacker control, then P can generate any sequence of system calls $A \in \Sigma^*$.

Attackers may subvert a vulnerable program's execution at any execution point, including the point of process initialization. Attackers can then arbitrarily alter the code and data of the program, and can even replace the program's entire memory image with an image of their choosing. Alternatively, the attacker could replace the disk image of a program with, for example, a trojan before the OS loads the program for execution. The attacker can generate any sequence of system calls and system call arguments, and the operating system will execute the calls with the privilege of the original program.

This threat model matches real-world attacks. In remote execution environments, programs execute on remote, untrusted machines but send a sequence of remote system calls back to a trusted machine for execution. An attacker controlling the remote host can arbitrarily alter or replace the remote program. The attacker's program image can then send malicious system calls back to the trusted machine for execution [11].

Common network-based attacks against server programs have a more restrictive threat model. Attackers can subvert execution only at points of particular program vulnerabilities and face greater restrictions in the attack code that they can then execute. As a result, if our system proves that a program model detects an attack in the strong threat model, it will also detect the attack in a more restrictive model. However, successful attacks discovered by our system are specific to the strong threat model. Although the program model would fail to detect the attack sequence even in the restricted threat model, a restricted attacker may be unable to cause the program to execute that attack. Our system currently does not make this determination and will report all attacks discovered in the strong threat model.

Consider the example in Fig. 2. This is a vulnerable program that reads command characters and filenames from user input. This input may come from the network if

```
void main (void) {
  char input[32];
  gets(input);
  if (input[0] == 'x') {
    setreuid(42, -1);
    syslog(1, "Execing file");
    execve(input+2, 0, 0);
  } else if (input[0] == 'e') {
    struct stat buf;
    syslog(1, "Echoing file");
    stat(input+2, &buf);
    int fd = open(input+2, O_RDONLY);
    void *filedata =
        mmap(0, buf.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
    write(1, filedata, buf.st_size);
  }
}
```

Fig. 2. Code example. We show system calls in boldface and library calls in italics. The unsafe call to *gets* allows an attacker to execute arbitrary code.

the program is launched by a network services wrapper daemon such as `xinetd`. The command-code and argument input resembles the usage of programs such as ftp servers or http servers. Suppose that the program is executed with stored but inactive privilege: its real and effective user IDs are a low-privilege user, but the saved user ID is root. If the input contains the command character 'x', then the program drops all of its saved privilege and executes a filename given in the input. If the input contains the command character 'e', then the program echoes the contents of a specified file to its output, which may be a network stream.

In our threat model, an attacker can arbitrarily alter the execution of this program. Perhaps the attacker exploits the vulnerable *gets* call; perhaps they use an attack vector that we have not considered. The attacker can cause the program to execute any system call, including system calls not contained in the original program code. The role of host-based intrusion detection is to detect any such subverted program execution.

3.2 Program Model

Readers familiar with pushdown automaton (PDA) models may elect to bypass this section, as it presents background material and standard notation previously used for PDA-based program models.

Model-based anomaly detection restricts allowed execution to a precomputed model of allowed behavior. A program model M is a language acceptor of system call sequences and is an abstract representation of the program's expected execution behavior. If Σ denotes the alphabet of system calls, then $L(M) \subseteq \Sigma^*$ denotes the language accepted by M. A system call sequence in $L(M)$ is valid; sequences outside $L(M)$ indicate anomalous program execution. In this paper, we implement a program model as a non-deterministic pushdown automaton (PDA).

Definition 1. *A* pushdown automaton (PDA) *is a tuple* $M = \langle S, \Sigma, \Gamma, \delta, s_0, Z_0, F \rangle$, *where*

- S *is a set of states;*
- Σ *is a set of alphabet symbols;*
- Γ *is a set of stack symbols;*
- $\delta \subseteq \{\langle s, \gamma \rangle \xrightarrow{\sigma} \langle s', \gamma' \rangle \mid s \in S, \gamma \in \Gamma \cup \epsilon, \sigma \in \Sigma \cup \epsilon, s' \in S, \gamma' \in \Gamma \cup \epsilon\}$*;*
- $s_0 \in S$ *in an initial state;*
- $Z_0 \in \Gamma^*$ *is an initial stack configuration;*
- $F \subseteq S$ *is a set of final states.*

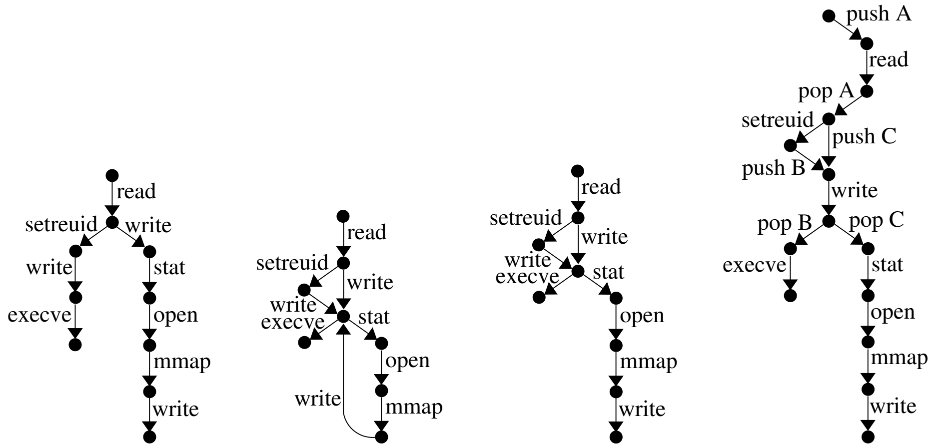
A PDA model has close ties to program execution. A state corresponds to a program point in the program's code. The initial state corresponds to the program's entry point. The final states correspond to program termination points, which generally follow an **exit** system call. The alphabet symbols are the system calls generated by a program as it executes. The stack symbols are return addresses specifying to where a function call returns. The initial stack Z_0 is empty, as a program begins execution with no return addresses on its call stack.

The transition relation δ describes valid control flows within a program. Our PDA model has three types of transitions:

- **System calls:** $\langle s, \epsilon \rangle \xrightarrow{\sigma} \langle s', \epsilon \rangle$ for $\sigma \neq \epsilon$ indicates that the program can generate system call σ when transitioning from state s to state s'. The PDA stack of function call return addresses remains unchanged.
- **Function calls:** $\langle s, \epsilon \rangle \xrightarrow{\sigma} \langle s', \gamma \rangle$ for $\gamma \neq \epsilon$ indicates that the program pushes return address γ onto the call stack when transitioning from state s to s'. Here, s corresponds to a function call-site in the program and s' corresponds to the entry point of the call's destination.
- **Function returns:** $\langle s, \gamma \rangle \xrightarrow{\sigma} \langle s', \epsilon \rangle$ for $\gamma \neq \epsilon$ indicates that the program returns from a function call and pops return address γ from the call stack. This transition can be followed only when γ is the top symbol of the PDA stack. The state s corresponds to a program point containing a function return instruction and s' is the program point to which control is actually returned.

Many program model designs proposed in academic literature are not presented as pushdown automata. However, the generality of a PDA allows us to characterize those models as PDA suitable for analysis using the techniques presented later in this paper. The context-free languages recognized by PDA completely contain the class of regular languages. All program models of which we are are aware accept either regular or context-free languages, and hence can always be characterized by a PDA. This includes:

- window-based models, such as the Stide model [8] (Fig. 3a) or the digraph model [23] (Fig. 3b);
- non-deterministic finite automata (NFA) [12, 14, 19, 23] (Fig. 3c);
- bounded-stack PDAs [11];
- deterministic PDAs, such as the VPStatic model [6];
- stack-deterministic PDAs, such as the Dyck model [6]; and
- non-deterministic PDAs [23] (Fig. 3d).



(a) Stide Model (b) Digraph Model (c) NFA Model (d) PDA Model

Fig. 3. Four different program models for the code of Fig. 2, each expressed as a pushdown automaton. For simplicity, we assume that the *gets* function call generates the system call **read** and the *syslog* function call generates **write**.

When a model accepts a regular language, we simply have $\Gamma = \varnothing$ and transitions in δ are only of the form $\langle s, \epsilon \rangle \xrightarrow{\sigma} \langle s', \epsilon \rangle$. Although the experiments in Sect. 7 consider the Stide model, a regular language acceptor, we intentionally designed our system to analyze pushdown automata so that it is relevant to a wide collection of program models of varying strength.

Commensurate with our threat model, we assume that an attacker has prior knowledge of the particular program model used to constrain execution of a vulnerable program. The security of the system then relies entirely upon the ability of the program model to detect attacks.

3.3 Finding Undetected Attacks

We have developed a model analysis system that evaluates a PDA-based program model and finds undetected attacks. Our design has three features of note:

- It operates automatically. A user must provide an initial, one-time operating system abstraction that can then be reused to analyze the model of any program executing on that operating system. This subsequent analysis requires no human input, allowing the analysis to scale easily to large collections of program models.
- Attacks, which are sequences of system calls, do not need to be known. In fact, our system provides attack sequences as output.
- System call arguments can significantly alter the semantic meaning of the calls. When our system finds an undetected attack sequence of system calls, it additionally provides the system call arguments necessary to effect the attack.

We construct an abstraction of the operating system with respect to its security-critical state. This abstraction can be repeatedly used to find attacks in the models of programs that execute on that operating system. Consider a simple example:

Example 1. Running our tool for each of the four models in Fig. 3 shows that none detect all attacks that execute a shell with root privilege. The tool automatically identifies a system call sequence, with arguments, that defeats each model:

read(0);
setreuid(0, 0);
write(0);
execve("/bin/sh");

The **read** and **write** calls are nops that are irrelevant to the attack. The **setreuid** call alters OS state to gain root access, and the **execve** call executes a shell with that access.

One of our long-term goals is to use discovered undetected attacks to guide the future design of program models and intrusion detection systems. Comparing the undetected attack sequence with the original program code of Fig. 2 suggests a model alteration that would eliminate this undetected attack. If the model constrains statically-known system call argument values, then an attacker cannot undetectably use the **setreuid** call to set the effective user ID to root. Although the attacker remains able to execute the shell, that shell will not have increased privilege.

We will consider additional examples in Sect. 5.

4 Operating System Model

Given a program model M, answering the question first posed in Sect. 1, *what attacks does M fail to detect?*, requires understanding of what "attack" means. Previous work defined attacks as known, malicious sequences of system calls [24]. Directly searching program models for these sequences unfortunately has two drawbacks:

- An attacker could transform an attack sequence detected by the program model into a different sequence that produces the same malicious effect but is allowed by the model. For example, meaningless *nop* system calls could be inserted into the attack, and system calls such as **write** could be changed to other calls such as **mmap**. In previous work, the onus of finding all attack variants was upon the human.
- This approach poorly handles program models that monitor both system calls and system call arguments [11, 23]. Identifying *nop* system calls is not straightforward when the allowed system call arguments are constrained by the model.

We decouple our approach from the need to know particular system call sequences that execute attacks. Instead, we observe that regardless of the system call sequence transformations used by an attacker, their attack will still impart the same adverse effect upon the operating system. It is precisely this adverse effect that characterizes an attack: it captures the malicious intent of the attacker. The actual system call sequence used by the attacker to bring about their intent need not be known *a priori*, and in fact is discovered automatically by our system.

To formalize attacks by their effect upon the operating system, we must first formalize the operating system itself. Our formalization has three components:

- a set of state variables,
- a set of initial assignments to those variables, and
- a set of system call transition relations that alter the state variables.

After developing the definitions of these components, we finally define attack effects.

4.1 State Variables

A collection of state variables model security-critical internal operating system state, such as user IDs indicating process privilege, access permissions for files in the filesystem, and active file descriptors. A state variable v has a value in the finite domain $dom(v)$ which contains either boolean values or integer values.

Definition 2. *The set of all* state variables *is V. The set of all* assignments *of values to variables in V is S. A* configuration *is a boolean formula over V that characterizes zero or more assignments.*

Model checking algorithms operate over boolean variables; variables in a finite domain are simply syntactic sugar and are represented internally as lists of boolean variables. We additionally allow variables to be aggregated into arrays and C-style structures, both of which our implementation automatically expands into flat lists of variables.

Consider the example of the operating system's per-process file descriptor table. We abstract this structure as an array of file descriptors, each of which has a subset of actual file descriptor data that we consider relevant to security:

FILEDESCRIPTORTABLE : **array** [0 .. MAXFD] of FILEDESCRIPTOR
FILEDESCRIPTOR : **struct** of
  INUSE : boolean
  FORFILE : integer
  CANREAD : boolean
  CANWRITE : boolean
  ATEOF : boolean

The INUSE field indicates whether or not this file descriptor is active. The remaining fields have meaning only for active descriptors. FORFILE is an index into an array of file structures, not shown here, that abstract the file system. CANREAD and CANWRITE indicate whether the file descriptor can be used to read or write the file pointed to by the FORFILE field. ATEOF is true when the file descriptor's offset is at the end of the file and allows us to distinguish between writes that overwrite data in the file and writes that simply append data to the file.

Identifying what operating system data constitutes "security-relevant state" is currently a manual operation. Whether the subsequent model checking procedure finds an undetected attack or reports that no attack exists, these results hold only with respect to the chosen OS abstraction. An attack sequence is executable and can be validated against the real operating system by actually running the attack in a sandboxed environment and verifying that it was successful. However, when the model checker finds

setuid (uid_t uid)

$$
\begin{array}{ll}
\{ & \\
\quad [uid \neq -1 \wedge euid = 0 \implies ruid' = uid \wedge euid' = uid \wedge suid' = uid] \wedge & (1) \\
\quad [uid \neq -1 \wedge euid \neq 0 \wedge (ruid = uid \vee suid = uid) \implies euid' = uid] \wedge & (2) \\
\quad [uid = -1 \vee (euid \neq 0 \wedge ruid \neq uid \wedge suid \neq uid) \implies true] & (3) \\
\} &
\end{array}
$$

Fig. 4. Specification for the **setuid** system call. Unprimed variables denote preconditions that must hold before the system call, and primed variables denote postconditions that hold after the system call. Any variable not explicitly altered by a postcondition remains unchanged.

no attack, there is no tangible artifact that may be verified. If relevant OS data is not included in the abstraction, then our system may fail to discover a mimicry attack. As a result, the absence of an attack in the abstract OS provides evidence but not a mathematical proof that the model will detect the attack when operating in a real OS.

The initial assignments of values to OS state variables encodes the OS state configuration present when a process is initialized for execution. We write these assignments as a boolean formula I over the state variables V; any assignment satisfying I is a valid initial state. In our work, we developed two different boolean formula for different classes of programs. The formula I for setuid root programs set the initial effective user ID to root; the formula for all other programs set the user ID to a low-privilege user.

4.2 System Call Transformers

System calls transform the state variables. For each system call, we provide a relation specifying how that call changes state based upon the previous state.

Definition 3. *Let π be a system call. Recall that V is the set of all OS state variables and S is the set of all value assignments. The set of* parameter variables *for π is Λ_π where $\Lambda_\pi \cap V = \varnothing$. The* system call transformer *for π is a relation $\Delta_\pi \subseteq S \times S$.*

In English, each system call transformer produces new assignments of values to OS state variables based upon the previous values of the OS state. We write each transformation function as a collection of preconditions and postconditions that depend on parameter variables. Preconditions are boolean formulas over $V \cup \Lambda_\pi$, and postconditions are boolean formulas over V. If a precondition formula holds before the system call executes, then the corresponding postcondition formula will hold after the system call.

Consider the example in Fig. 4. The specification for **setuid** shows that the system call has one parameter variable of type `uid_t`, which is an integer valued type. The boolean formula encodes three sets of preconditions and postconditions. From line (1), if the uid argument is valid and the effective user ID before the **setuid** call is root, then after the call, the real, effective, and saved user IDs are all set to the user ID specified as the argument to **setuid**. Implicitly, all other OS state variables remain unchanged by the call. Line (2) handles the case of a non-root user calling **setuid**. If either the real or saved user IDs match the argument value, then the effective user ID is changed to that value. Again, all other state is implicitly unchanged. Line (3) allows **setuid** to be used as a nop transition that does not change OS state when neither the line (1) nor line (2) preconditions hold true. We note that line (3) is redundant and can be omitted from the **setuid** specification; we show it here only to emphasize the ability of **setuid** to be used as a nop.

We now have all components of the operating system abstraction:

Definition 4. *The* operating system (OS) model *is $\Omega = \langle V, I, \Delta \rangle$ where V is the collection of OS state variables, I is a boolean formula over V indicating the initial OS state configuration, and $\Delta = \{\Delta_1, \ldots, \Delta_n\}$ is the collection of system call transformers.*

4.3 Attacks

An attack is a sequence of system calls that executes some malicious action against the operating system. However, these sequences are not unique. Attackers can produce an infinite number of obfuscated attack sequences by inserting extraneous, *nop* system calls into a known sequence and by changing attack system calls into other semantically-equivalent calls. Manual specification of actual attack sequences can be incomplete, as there may be attack obfuscations not known to the individual specifying the attacks. We circumvent this problem by specifying the effects of attacks rather than the sequences themselves.

Definition 5. *An* attack effect *$\mathcal{E}$ is a boolean formula over V.*

The formula $\mathcal{E}$ characterizes bad operating system configurations indicative of a successful intrusion. It describes the attacker's intent and the effect of the attack upon the OS. Any system call sequence A that takes the OS from an initial, safe configuration satisfying I to a configuration satisfying $\mathcal{E}$ is then an attack sequence. If A is allowed by the program model, than A is an undetected attack.

5 Automatic Attack Discovery

The role of automatic attack discovery is to determine if any system call sequences accepted as valid execution by a program model will induce an attack configuration $\mathcal{E}$.

Let $\mathcal{E}$ be an attack effect. The notation $\Box\neg\mathcal{E}$ expresses a safety property in linear-time temporal logic (LTL) that means "globally, $\mathcal{E}$ is never true". A program model M will detect any attack attempting to induce the effect $\mathcal{E}$ if and only if $M \vDash \Box\neg\mathcal{E}$. That is, within the executions allowed by M interpreted in the OS model Ω, the attack goal can never occur. The model checker attempts to prove this formula true. If the proof succeeds, then the attack goal could not be reached given the system call sequences allowed by the program model. If the proof fails, then the model checker has discovered a system call sequence that induces the attack goal.

We consider several examples:

Example 2 (Expanded from Sect. 3 Example 1). First, we find attacks that execute a root-shell undetected by the four models of Fig. 3.

If the attack succeeds, then the executing image file is `/bin/sh` and the effective user ID is 0:

$$\mathcal{E} : image = /bin/sh \wedge euid = 0$$

This boolean expression expresses the effect of the attack rather than any particular sequence of system calls that produces the effect. Running our tool for each of the four models shows that none detect the attack, as shown in Sect. 3.3.

Example 3. Next, we try to find undetected attacks that write to the system's password file.

If this attack succeeds, then the file `/etc/passwd` will have been altered:

$$\mathcal{E} : file[/etc/passwd].written = true$$

The tool automatically finds a successful attack against the Digraph and NFA models:

read(0);
setreuid(0, 0);
write(0);
stat(0, 0);
open("/etc/passwd", O_WRONLY | O_APPEND) = 3;
mmap(0, 0, 0, 0, 0, 0);
write(3);

The attack sequence first sets the effective user ID to root, which then allows the process to open the password file and add a new user. The **read**, **stat**, **mmap**, and first **write** calls are all nops irrelevant to the attack.

Conversely, the tool discovers that the Stide and PDA models will always detect any attack that tries to alter the password file. These models accept no system call sequence that ever has write privilege to the file `/etc/passwd`.

Example 4. Finally, we try to find undetected attacks that add a new root-level account to the system and execute a user-level shell, with the expectation that the attacker can subsequently switch to high privilege via the new account.

This combines elements of Examples 2 and 3:

$$\mathcal{E} : image = /bin/sh \wedge file[/etc/passwd].written = true$$

The system finds an attack against the Digraph model:

read(0);
setreuid(0, 0)
write(0);
stat(0, 0);
open("/etc/passwd", O_WRONLY | O_APPEND) = 3;
mmap(0, 0, 0, 0, 0, 0);
write(3);
execve("/bin/sh");

The system proves that the Stide, NFA, and PDA models all detect this attack regardless of any attempts to obfuscate a system call sequence. This is evident from the models: although they accept sequences that open and write to a file, they do not allow subsequent execution of a different program.

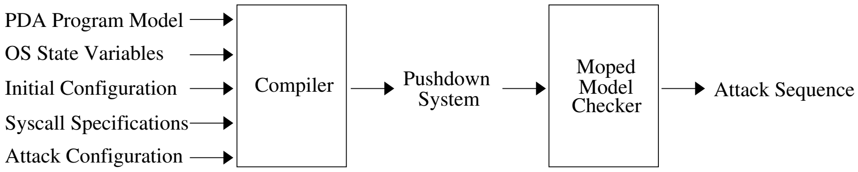


Fig. 5. Architecture.

6 Implementation

Model checking either proves that an unsafe OS configuration cannot be reached in a program model or provides a counter-example system call trace that produces the unsafe configuration. As we are verifying transition systems that may be pushdown automata, we are limited in implementation options to pushdown model checkers [5, 17]. Moped [18] and Bebop [1] are interchangeable tools that analyze pushdown systems. Our implementation uses Moped simply because of its public availability.

When a context-sensitive program model is used to verify a stream of system calls generated by an executing process, we call that model a pushdown automaton (PDA). The system calls are the input tape and the PDA has final states that correspond to possible program termination points. When we analyze a model to verify its ability to detect attacks, we call the model a *pushdown system*.

Definition 6. *A* pushdown system (PDS) *is a tuple* $Q = \langle S, \Sigma, \Gamma, \delta, s_0, Z_0 \rangle$, *where each element of the tuple is defined as in Definition 1.*

The definition of a PDS is identical to that of a PDA, with the exception that the PDS has no final states and no input tape. A PDS is just a transition system used to analyze properties of sequences and is not a language acceptor. Moped verifies that no sequence of system calls in the PDS will produce an unsafe operating system configuration.

The input to Moped is a collection of variables and a PDS where each transition in δ is tagged with a boolean formula. The formula expresses preconditions over the variables that are required to hold before traversing the transition and postconditions that hold after traversal. If no preconditions hold, then Moped will not traverse the transition and will not alter the state variables. The Moped input language allows both boolean and integer variables, although the integer variables are represented internally as ordered lists of boolean bits.

We have written a specification compiler that will produce valid Moped input files from a PDA program model, the OS state variables, the initial OS configuration, the system call transformers, and the attack that we wish to prove is detected (Fig. 5). The compiler converts the PDA to a PDS in a straightforward manner by simply removing the designations for final states. It compiles each system call transformer into a boolean formula expected by Moped and annotates all system call transitions in the PDS with these formulas. If the PDS contains other transitions, such as push and pop transitions that do not correspond to system calls, the compiler annotates the transitions with a formula whose preconditions match any OS variable assignment and whose postconditions simply maintain that assignment. We add one new state $\mathcal{A}$ to the PDS and new transitions to $\mathcal{A}$ after each system call transition. The precondition on these new transitions is exactly the OS attack configuration $\mathcal{E}$ that we wish to prove cannot be reached in the model. We then invoke Moped so that it proves that state $\mathcal{A}$ cannot be reached or provides a counterexample trace of system calls reaching state $\mathcal{A}$.

7 Experiments

We used our implementation to find undetected attacks in program models that have appeared in academic literature. We show that our automated approach can find the mimicry and evasion attacks that previously were discovered manually [20–22, 24]. The automated techniques allow for better scaling of the number of test cases when compared to manual approaches.

We can automatically find mimicry and evasion attacks that previous research found only with manual analysis. Previous work considered four test programs—`wu-ftpd`, `restore`, `traceroute`, and `passwd`—that had known vulnerabilities allowing attackers to execute a root shell. Forrest et al. [8] successfully detected known attack instances using a model called *Stide*. The Stide model is a context-insensitive characterization of execution learned from system call traces generated by a series of training runs. Wagner and Soto [24] and Tan et al. [20–22] demonstrated that attackers could modify their attacks to evade detection by the Stide model. In some cases, the undetected attacks were *not* semantically equivalent to the original root shell exploit, although the attacks adversely modified system state so that the attacker can subsequently gain root access. For example, successful attack variants may:

- write a new root-level account to the user accounts file `/etc/passwd`;
- set `/etc/passwd` world-writable so that an ordinary user can add a new root account; or
- set `/etc/passwd` owned by the attacker so that the attacker can add a new root account.

We automatically found these undetected attacks. We used our infrastructure to analyze the Stide model for each of the four programs with respect to each of the four attack goals. For `wu-ftpd`, we constructed the Stide model using the original Linux training data of Forrest et al. [7]. We were unable to obtain either the `wu-ftpd` training data used by Wagner and Soto or the Stide models that they constructed from that data. As a result, we were able to find attacks in the `wu-ftpd` model constructed from Forrest's data that were reportedly not present in the model constructed from Wagner's data. For the remaining three test programs, we constructed models from training data generated in the manner described by Tan et al. [20]. Our specification compiler combined PDA representations of the Stide models with specifications of Linux system calls to produce pushdown systems amenable to model checking.

Table 1 lists the size of the PDA representation of the Stide model for each program. The OS state model included 119 bits of global state and 50 bits of temporary state for system call argument variables. This temporary state reduces Moped's resource demands because it exists only briefly during the model checker's execution.

Table 1. Number of states and edges in the transition systems describing the Stide model for each of the four test programs. The boolean OS state includes 119 bits for global state variables and 50 bits of temporary state for system call argument variables.

	wu-ftpd	*restore*	*traceroute*	*passwd*
Edge count	2,085	1,206	623	1,058
State count	1,477	892	459	766

Table 2. Evaluation of the Stide model's ability to detect classes of attacks. A "yes" indicates that the Stide model will always detect the attack because the model checker was unable to find an undetected attack. A "no" indicates that Stide is unable to protect the system from the attack because the model checker discovered an undetected attack sequence. Writing to `/etc/passwd` is normal behavior for *passwd*.

	wu-ftpd	*restore*	*traceroute*	*passwd*
Execute root shell	No	No	Yes	Yes
Write to `/etc/passwd`	No	No	No	—
Set `/etc/passwd` *world-writable*	Yes	Yes	Yes	No
Set `/etc/passwd` *attacker-owned*	Yes	Yes	Yes	No

Table 2 presents the ability of the Stide model to detect any attack designed to reach a particular attack goal, as determined by Moped. A "yes" indicates that the model will always prevent any attacker from reaching their goal, regardless of how they try to transform or alter their attack sequence of system calls. A "no" indicates the reverse: the model checker was able to find a system call sequence, with arguments, accepted by the model but that induces the unsafe operating system condition. Figure 6 shows the undetected attack against *traceroute*'s Stide model discovered by our system. We automatically found all attacks that researchers previously found manually, one additional attack due to differences between Forrest's training data and Wagner's training data for `wu-ftpd`, and an additional attack against `restore` not found by previous manual research.

Previous work missed this attack because manual inspection does not scale to many programs and attacks, and hence the research did not attempt to compute results for all attack goals in all programs. When using manual inspection, it is likewise difficult to show that an attack is not possible: has the analyst simply not considered an attack that would be successful? Model checking can prove that a goal is unreachable regardless of the actual system calls used by the attacker in their attempt to reach the goal. We

close; munmap; **open("/etc/passwd", O_RDWR | O_APPEND) = 3;**
fcntl64; fcntl64; fstat64; mmap2; read; close; munmap; **write(3);**

Fig. 6. Undetected attack against the Stide model of *traceroute* that adds a new root-level user to `/etc/passwd`. The system calls producing the attack effect are shown in boldface. Although our system discovers arguments for the nop system calls, we omit the arguments here for conciseness. We do not discover the actual string value written to `/etc/passwd`; a suitable string would be "`attacker::0:0::/root:/bin/sh\n`".

Table 3. Model checking running times, in seconds.

	wu-ftpd	*restore*	*traceroute*	*passwd*
Execute rootshell	0.34	0.75	2.38	2.70
Write to `/etc/passwd`	0.39	0.73	1.33	—
Set `/etc/passwd` *world-writable*	39.10	74.41	0.90	2.02
Set `/etc/passwd` *attacker-owned*	41.11	65.21	1.15	1.81

can show that the models of the first three test programs detect all attacks that try to set `/etc/passwd` world-writable or owned by the attacker—assertions that previous manual efforts were unable to make. Although the proofs of detection hold with respect to the OS abstraction and may not hold in the actual OS implementation, as described in Sect. 2, the proofs do provide a strong indication that runtime attack detection in the real system will be effective.

Table 3 lists the model checker's running times in seconds for each model and attack goal. When comparing the running times with Table 2, a loose trend becomes apparent. In cases where the model checker found an attack, the running times are very small. When no attack was found, the model checker executed for a comparatively longer period of time. This disparity is to be expected and reflects the behavior of the underlying model checking algorithms. When a model checker finds a counter-example that disproves a logical formula—here an attack sequence that violates an LTL safety property—the model checker can immediately terminate its execution and report the counter-example. However, a successful proof that the logical formula holds requires the model checker to follow exhaustively all execution paths and early termination is not possible.

We believe that automating the previously manual process of attack construction is a significant achievement. We are not surprised at our ability to find undetected attacks: attackers have significant freedom in program models that do not constrain system call arguments. For example, the system call sequence **open** followed by **write** without argument constraints can be misused by an attacker to alter the system's password file. Yet, this is a common sequence contained in nearly every non-trivial program, including programs that execute with the root-level privilege required to alter the password file.

Our automated system provides us with the means to understand exactly where a program model fails. From Table 2 we learn which classes of attack can be effectively detected by a program model and which classes of attack require alternative protection strategies. What is important is not simply that the models fail to detect some attacks, but that we know exactly what type of attacks are missed.

8 Conclusions

Model-based intrusion detections systems are useful only when they actually detect or prevent attacks. Finding undetected attacks manually is difficult, error-prone, unable to scale to large numbers of program models and attacks, and unable to prove that an attack will always be detected. We showed here that formalizing the effects of attacks upon the operating system provides the operational means to find undetected attacks automatically. A model checker attempts to prove that the attack effect will never hold in the program model. By finding counter-examples that cause the proof to fail, we find undetected attacks: system call sequences and arguments that are accepted as valid execution and induce the malicious attack effect upon the operating system. This automation let us find undetected attacks against program models that previously were found only with manual inspection of the models. The efficiency of the computation—about 2 seconds computation to find undetected attacks—provides an indication that this automated approach can easily scale to large collections of program models.

Acknowledgments. We thank the anonymous reviewers and the members of the WiSA project at Wisconsin for their helpful comments that improved the quality of the paper.

This work was supported in part by Office of Naval Research grant N00014-01-1-0708, NSF grant CCR-0133629, and Department of Energy grant DE-FG02-93ER25176. Jonathon T. Giffin was partially supported by a Cisco Systems Distinguished Graduate Fellowship. Somesh Jha was partially supported by NSF Career grant CNS-0448476. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes, notwithstanding any copyright notices affixed hereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the above government agencies or the U.S. Government.

References

[1] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *7th International SPIN Workshop on Model Checking of Software*, Stanford, California, Aug./Sep. 2000.
[2] F. Besson, T. Jensen, D. L. Métayer, and T. Thorn. Model checking security properties of control-flow graphs. *Journal of Computer Security*, 9:217–250, 2001.
[3] H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *9th ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, Nov. 2002.
[4] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.
[5] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Computer Aided Verification (CAV)*, Chicago, Illinois, July 2000.
[6] H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2004.
[7] S. Forrest. Data sets—synthetic FTP. http://www.cs.unm.edu/∼immsec/data/FTP/UNM/-normal/synth/, 1998.
[8] 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*, Oakland, California, May 1996.
[9] D. Gao, M. K. Reiter, and D. Song. On gray-box program tracking for anomaly detection. In *USENIX Security Symposium*, San Diego, California, Aug. 2004.
[10] J. T. Giffin, D. Dagon, S. Jha, W. Lee, and B. P. Miller. Environment-sensitive intrusion detection. In *8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Seattle, Washington, Sept. 2005.

[11] J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *11th USENIX Security Symposium*, San Francisco, California, Aug. 2002.

[12] R. Gopalakrishna, E. H. Spafford, and J. Vitek. Efficient intrusion detection using automaton inlining. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2005.

[13] J. D. Guttman, A. L. Herzog, J. D. Ramsdell, and C. W. Skorupka. Verifying information flow goals in Security-Enhanced Linux. *Journal of Computer Security*, 13:115–134, 2005.

[14] L.-c. Lam and T.-c. Chiueh. Automatic extraction of accurate application-specific sandboxing policy. In *Recent Advances in Intrusion Detection (RAID)*, Sophia Antipolis, French Riveria, France, Sept. 2004.

[15] C. R. Ramakrishnan and R. Sekar. Model-based vulnerability analysis of computer systems. In *2nd International Workshop on Verification, Model Checking and Abstract Interpretation*, Pisa, Italy, Sept. 1998.

[16] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, Feb. 2000.

[17] S. Schwoon. *Model-Checking Pushdown Systems*. Ph.D. dissertation, Technische Universität München, June 2002.

[18] S. Schwoon. Moped—a model-checker for pushdown systems. http://www.fmi.uni-stuttgart.de/szs/tools/moped/, 2006.

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

[20] K. Tan, K. S. Killourhy, and R. A. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. In *Recent Advances in Intrusion Detection (RAID)*, Zürich, Switzerland, Oct. 2002.

[21] K. Tan and R. A. Maxion. “Why 6?” Defining the operational limits of stide, an anomaly based intrusion detector. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2002.

[22] K. Tan, J. McHugh, and K. Killourhy. Hiding intrusions: From the abnormal to the normal and beyond. In *5th International Workshop on Information Hiding*, Noordwijkerhout, Netherlands, Oct. 2002.

[23] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2001.

[24] D. Wagner and P. Soto. Mimicry attacks on host based intrusion detection systems. In *9th ACM Conference on Computer and Communications Security*, Washington, DC, Nov. 2002.

[25] B. J. Walker, R. A. Kemmerer, and G. J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2), Feb. 1980.