MOPS: an Infrastructure for Examining Security Properties of Software

Hao Chen (hchen@cs.berkeley.edu)
David Wagner (daw@cs.berkeley.edu)

Computer Science Division
University of California at Berkeley

Report No. UCB/CSD-02-1197

September 20, 2002

Computer Science Division (EECS)
University of California
Berkeley, California 94720

This research was supported in part by DARPA contract N66001-01-C-8040 and by an equipment grant from Intel.

Abstract

We describe a formal approach for finding bugs in security-relevant software and verifying their absence. The idea is as follows: we identify rules of safe programming practice, encode them as safety properties, and verify whether these properties are obeyed. Because manual verification is too expensive, we have built a program analysis tool to automate this process. Our program analysis models the program to be verified as a pushdown automaton, represents the security property as a finite state automaton, and uses model checking techniques to identify whether any state violating the desired security goal is reachable in the program. The major advantages of this approach are that it is sound in verifying the absence of certain classes of vulnerabilities, that it is fully interprocedural, and that it is efficient and scalable. Experience suggests that this approach will be useful in finding a wide range of security vulnerabilities in large programs efficiently.

1 Introduction

Software vulnerabilities are an enormous cause of security incidents in computer systems. A system is only as secure as its weakest link, and often the software is the weakest link.

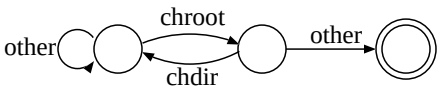
We can attribute software vulnerabilities to several causes. Some bugs, like buffer overruns in C, reflect poorly designed language features and can be avoided by switching to a safer language, like Java. However, safer programming languages alone cannot prevent many other security bugs, especially those involving higher level semantics. As a typical example, OS system calls have implicit constraints on how they should be called; if coding errors cause a program to violate such constraints when interacting with the OS kernel, this may introduce vulnerabilities.

In this paper, we focus on detecting violations of ordering constraints, also known as *temporal safety properties*. A temporal safety property dictates the order of a sequence of security-relevant operations. Our experience shows that many rules of good programming practice for security programs can be described by temporal safety properties. Although violating such properties may merely indicate risky features in a program in some cases, it often renders the program vulnerable to attack, depending on the nature of the violation. In either case, the ability to detect violations of the properties or to verify the satisfaction of them would be a significant help in reducing the frequency of software vulnerabilities.

To illustrate the relevance of such temporal safety properties, we give next a few examples that reflect prudent programming practice for Unix applications.

- Property 1. Suppose a process uses the *chroot* system call to confine its access to a sub filesystem. In this case, the process should immediately call *chdir("/")* to change its working directory to the root of the sub filesystem.

  This rule can be described by the temporal safety property that any call to *chroot* should be immediately followed by a call to *chdir("/")*. The program in Figure 1(b) violates this property: it fails to call *chdir("/")* after *chroot("/var/ftp/pub")*, so its current directory remains `/var/ftp`. As a result, a malicious user may ask the program to open the file `../../etc/passwd` successfully even though this is outside the chroot jail and the programmer probably intended to make it inaccessible. Here, the malicious user takes advantage of the method by which the operating system enforces *chroot(new_root)*. When a process requests access to a file, the operating system follows every directory component in the path of the file sequentially to locate the file. If the operating system has followed into the directory `new_root` and if the next directory name in the path is "`..`", then "`..`" is ignored. However, in the above example, since the current directory is `/var/ftp`, the path `../../etc/passwd` never comes across the new root `/var/ftp/pub` and is therefore followed successfully by the operating system. In short, the *chroot* system call has subtle traps for the unwary, and Property 1 encodes a safe style of programming that avoids some of these traps.

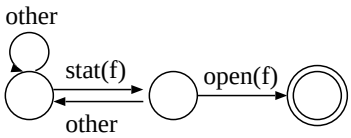


(a) An FSA describing the Property 1

```
// Here the current directory is "/var/ftp"
chroot("/var/ftp/pub");
filename = read_from_network();
fd = open(filename, O_RDONLY);
```

(b) A program segment violating Property 1. Note that the program fails to call *chdir("/")* after *chroot()*, so if `filename` is *"../../etc/passwd"*, a security violation ensues.

Figure 1: An FSA illustrating Property 1 (*chroot()* must always be immediately followed by *chdir()*) and a program violating it



(a) An FSA describing Property 2

```
// Here ruid=x (a normal user), euid=0 (root)
stat(logfile, &st);
if (st.st_uid != getuid())
    return -1;
open(logfile, O_RDWR);
```

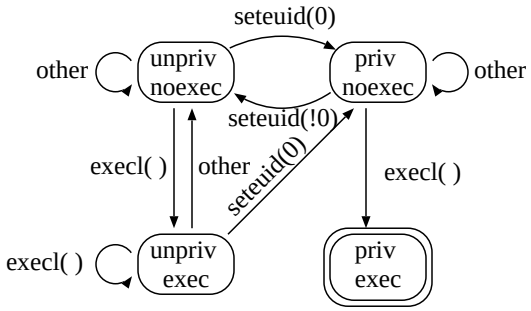
(b) A program segment violating Property 2. Note that the program is susceptible to a race condition, since the binding of `logfile` to a file may change between the *stat()* and *open()* calls.

Figure 2: An FSA illustrating Property 2 (*stat(f)* must not be followed by *open(f)*) and a program violating it.

- Property 2. A call to *stat(f)* should not be followed immediately by a call to *open(f)* (otherwise, it is a suspicious code sequence that tends to indicate potential security weaknesses [5]).

  Before explaining this property, we give some background. In Unix systems, each process has an effective user ID (*euid*), which determines the file access permission of the process. If the *euid* of the process is zero, the user ID of the super-user *root*, the process has full access to the filesystem and is said to be *privileged*. Consider a privileged process that runs on behalf of a normal user and that wants to constrain itself to access only files owned by the normal user. A naive implementation involves two steps: (1) call *stat("foo")* to identify the owner of the file `foo`; (2) only open the file if it is owned by the current user. This strategy, however, is insecure because of a race condition: an attacker may change the file associated with the name `foo` (e.g., through modifying a symbolic link) between the *stat("foo")* and *open("foo")* calls. The program in Figure 2(b) illustrates this race condition. Suppose the filename `foo` in the variable `logfile` initially is a symbolic link to a file owned by the attacker. When *stat(logfile, &st)* is called, the program verifies that the attacker is the owner of the file. But before the program proceeds to open the file by calling *open(logfile, O_RDWR)*, the attacker changes `foo` to be a symbolic link to `/etc/passwd`, a file that should not be writable to him. So *open(logfile, O_RDWR)* ends up opening `/etc/passwd` for him in read/write mode. We see that violations of Property 2 often point to potential security vulnerabilities in the code.

- Property 3. Since a privileged process has full access permission to the system, it should not make



(a) An FSA describing Property 3

```
// Here ruid=x (a normal user), euid=0 (root)
execl("/bin/sh", "sh", NULL);
```

(b) A segment from a setuid-root program that violates Property 3. The user will receive a shell with full root access, which may not have been intended. Probably the programmer should have called *seteuid(x)* to drop privilege before spawning the shell.

Figure 3: An FSA illustrating Property 3 (*execl()* must not be called in privileged state) and a program violating it

certain system calls that run untrusted programs without first dropping all privileges (thereby granting them with full access permission to the system).

One such system call is *execl*. For example, the program in Figure 3(b) calls *execl("/bin/sh", "sh", NULL)* in the privileged state, giving the untrusted user a shell with full filesystem access permission. It violates the property that a privileged process should drop privilege (by calling *seteuid(u)* with some user ID *u≠0*, for example[1]) before calling *execl*.

In summary, the Unix system call interface comes with various pitfalls and implicit requirements on how this interface should be invoked. The temporal safety properties listed above encode some of these requirements in an explicit form. To reduce the risk of security vulnerabilities we would like to verify that these security properties are all satisfied.

Although checking temporal safety properties by hand is feasible in small programs, it does not scale to large programs because the sequence of operations in a property may span multiple functions or files in a program. Moreover, we would like to be confident that the property is satisfied on *all* execution paths in the program, yet manually checking all paths is infeasible in most cases. This point is illustrated in the program in Figure 4 where the path $[d_0 d_2 d_3 d_4]$ in the function *drop_privilege* drops privilege, but the path $[d_0 d_1]$ fails to do so. So the path $[m_1 d_0 d_2 d_3 d_4 m_2 m_3]$ satisfies Property 3, but the path $[m_1 d_0 d_1 m_2 m_3]$ violates it. These types of path-dependent errors are common in programs, but such interprocedural errors are difficult to discover with testing or manual review, especially if the caller and callee are in different source files. As a result, we conclude that automated tools to help with this task are needed.

In this paper, we describe an automated approach to help examine security-related temporal safety properties (abbreviated as *security properties* henceforth) in software. We have built MOPS [2], a program analysis tool that allows us to make these properties explicit and to verify whether they are properly respected by the source code of some application.

MOPS determines at compile time whether there is *any* execution path through a program that may violate a security property. Since it is infeasible to traverse every execution path because there are prohibitively many paths, we use techniques from model checking and program analysis to structure the analysis. We model the security property as a Finite State Automaton (FSA) and the program as a Pushdown Automaton (PDA). We then use model checking to determine whether certain states representing violation of the

[1] For additional security, a privileged process should call *setuid(u)* or *setresuid(u,u,u)* to drop all the privileges in its *ruid*, *euid*, and *suid*. We simplify the property by considering only the *euid* and the *seteuid* system call.

[2] MOdel Checking Programs for Security properties

```
    int main(int argc, char *argv[])
    {   // start with root privilege
m0:     do_something_with_privilege();
m1:     drop_privilege();
m2:     execl("/bin/sh", "/bin/sh", NULL); // risky syscall
m3:}

    void drop_privilege()
    {
        struct passwd *passwd;

d0:     if ((passwd = getpwuid(getuid())) == NULL)
d1:         return; // but forget to drop privilege!
d2:     fprintf(log, "drop priv for %s", passwd->pw_name);
d3:     seteuid(getuid()); // drop privilege
d4: }
```

Figure 4: A program where the security property is violated on one execution path but not on the other one.

security property in the FSA are reachable in the PDA. Our approach may be viewed as an application of lightweight formal methods to an interesting class of security properties.

MOPS is distinguished from other related tools in the following aspects. First, since it is based on a solid formal foundation, i.e., model checking, it can take advantage of existing algorithms and future advances from the model checking community. Second, because it fully supports interprocedural analysis and because interprocedural bugs are more elusive than intraprocedural ones, MOPS promises to complement manual auditing where an automated tool is needed the most. Third, MOPS is sound (modulo the mild assumptions to be discussed in Section 7): it reliably catches all bugs of the specified types. This property makes MOPS useful not only in finding security bugs but also in verifying security properties. Fourth, thanks to a novel technique that substantially reduces the size of a program without affecting the result of model checking, MOPS scales well to large programs in both time and space, overcoming the scalability problem that hinders many software model checking systems (see Section 5. Other tools have some of these properties, but to the best of our knowledge MOPS is the only tool that has *all* of these desirable properties.

This paper is organized as follows. Section 2 and 3 describe the formal models that are the foundations of this approach. The former presents an abstract view of the models and the latter describes their implementation. Section 4 discusses how to derive a security model from the operating system accurately. Section 5 describes some important algorithms of MOPS. Section 6 presents our experiences in using MOPS to examine several security-relevant software. Section 7 discusses the soundness of this approach and its limitations. Section 8 reviews the related work and compares them to MOPS.

2 Formal Models

MOPS is based on a formal approach that builds a formal model of a program and of a security property and then analyzes the models. We start by describing the problem.

2.1 The Problem

Given a program and a security property, the goal is to verify whether the program satisfies the property, and if not, identify why. Typically, the program performs several security-relevant operations, and the security property specifies certain sequences of security operations that lead to potential security violations and that should be avoided. The problem is to determine if there exists *any* execution path through the program that contain such a sequence of operations.

2.2 The Formal Framework

We start with a highly abstract model. Let Σ be the set of security-relevant operations. Let $B \subseteq \Sigma^*$ be all sequences of security operations that violate the security property (B stands for *bad*). A trace $t \in \Sigma^*$ will represent a sequence of operations executed by a path p through the program, and we say that t is a feasible trace if p is a possible execution path through the program. Let $T \subseteq \Sigma^*$ denote the set of all feasible traces, extracted from all execution paths of the program (T stands for *trace*). The problem is to decide if $T \cap B$ is empty. If so, then the security property is satisfied. If not, then some execution path in the program violates the security property.

In the above model, B and T are arbitrary languages. Since in general T is an uncomputable set, deciding whether $T \cap B = \emptyset$ is an undecidable problem. To make the problem decidable, we specialize the problem by restricting the form of B and T.

First, we assume that B, the set of sequences of security operations that violate the security property, is a regular language. Our experiences show that most temporal safety properties can be described by regular languages (see Sections 3 and 6 for examples). Since B is a regular language, there exists a Finite State Automaton (FSA) M that accepts B (M stands for *model*); in other words, $B = L(M)$. We will usually identify the security property with its representation as an FSA.

Although we assume that B is a regular language, it is not sufficient to assume that T, the set of all feasible traces, will always be a regular language. The problem is that a regular language cannot describe the execution paths that cross function calls very well. In the case of a function call, a stack is needed to record the return address in the caller, and the language generated with a stack is context free rather than regular. Therefore, in this paper we model the set T of feasible traces as a context free language. It follows that there exists a Pushdown Automaton (PDA) P that accepts T (P stands for *program*). A PDA consists of a set of states, stack symbols, input symbols, and transitions. A snapshot of the PDA, called a *configuration*, consists of its current state and all the symbols on the stack. A transition specifies that the PDA moves from one configuration to another upon receiving a certain input symbol. With these specializations of B and T, the original problem becomes deciding if $L(M) \cap L(P)$ is empty.

To solve the problem, first we need to compute $L(M) \cap L(P)$. Since $C = L(M) \cap L(P)$ is the intersection of a regular language ($L(M)$) and a context free language ($L(P)$), C is a context free language. It also follows that C is accepted by the PDA that is the intersection of M and P. Second, we need to determine if the language C is empty. According to automata theory, there are efficient algorithms to compute the intersection of a PDA and an FSA and to determine if the language accepted by a PDA is empty [14, §6.2 and §6.3]. Hence we obtain a means to verify whether the security property is satisfied by the program.

Using a context free language to model the set of feasible traces does introduce some imprecision. In general we have $T \subseteq L(P)$: the PDA P will indeed accept all feasible traces, yet it might also accept some additional, spurious traces that are in fact infeasible due to the presence of other effects (such as data flow) not modeled in our framework. Nonetheless, since $T \subseteq L(P)$, we are guaranteed that $T \cap B \subseteq L(P) \cap L(M)$. Consequently, if $L(M) \cap L(P)$ is empty, we can conclude that $T \cap B$ is also empty, hence the program definitely satisfies the security property; in contrast, if $L(M) \cap L(P)$ is non-empty, then we can

only say that $T \cap B$ may or may not be empty, hence the program might not satisfy the security property, but there are no guarantees in this case.

This means that our analysis is *sound*: it may make mistakes by giving false alarms (warnings that do not correspond to an actual security vulnerability), but it will not overlook a real violation of the security property. This limitation is unavoidable. Since the general problem is undecidable, no algorithm can both avoid false alarms and avoid overlooking real bugs. Our experience is that false alarms are tolerable enough in practice that the approach is still useful despite occasional bogus warning messages.

2.3 A Concrete Example

To illustrate the formal framework, let us work through a concrete example. The problem is to check if the program in Figure 4 violates the security property that a process should not make the *execl* system call while it is in the privileged state (Figure 3(a))

In this problem, the set of security operations is $\Sigma = \{\textit{execl}(), \textit{seteuid}(0), \textit{seteuid}(!0)\}$, where the last element represents any call to *seteuid* with a non-zero parameter (representing a non-root user ID). The set $B \subseteq \Sigma^*$, the sequences of security-relevant operations that violate the security property, is accepted by the FSA M shown in Figure 3(a). The set $T \subseteq \Sigma^*$, the feasible traces of the program in Figure 4, is $T = \{[\textit{seteuid}(!0), \textit{execl}()], [\textit{execl}()]\}$. Since this is a setuid-root program, the initial state in the FSA M is $(\textit{priv}, \textit{noexec})$. According to Figure 3(a), although the path $[\textit{seteuid}(!0), \textit{execl}()]$ in T is not accepted by M, the path $[\textit{execl}()]$ in T is accepted by M. Therefore, we find that $T \cap L(M) \neq \emptyset$, or in other words, an execution path in the program violates the security property. This indicates the presence of a security vulnerability.

3 Implementation of Formal Models

In this section, we describe how to construct formal models from security properties and programs.

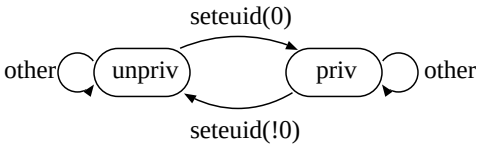
3.1 Modeling Security Properties

We call an FSA that describes a security property a *security model*. A transition in the FSA represents an execution of a security-relevant operation. All sequences of operations that violate the property end in the final states of the FSA. So the final states might also be thought of as *risky states* and are shown in double circles in the figures. The FSAs describing Properties 1, 2, and 3 in Section 1 are shown in Figures 1(a), 2(a), and 3(a) respectively. Note that in these figures each transition labeled *other* is a special transition, which is taken when no other transition from the same state can be taken.

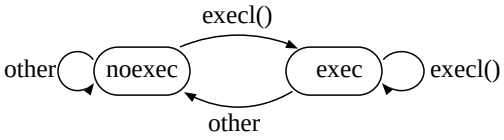
3.1.1 Modularization

One important feature of MOPS is that it allows complex security properties to be decomposed into simpler security models which are easier to describe. MOPS is able to combine these simpler models into a complex model on the fly [3]. For example, consider the property that a process should not make a risky system call such as *execl* while it is in the privileged state. This property can be decomposed into two simpler models: the first one describes the transition of a process between the privilege state and the unprivileged state (Figure 5(a)) and the second one describes the execution of a risky system call (Figure 5(b)). MOPS automatically combines the two simpler FSAs into the product automaton shown in Figure 5(c). Checking the program in Figure 4 against this security model shows that the risky state is reachable at the program point m_3.

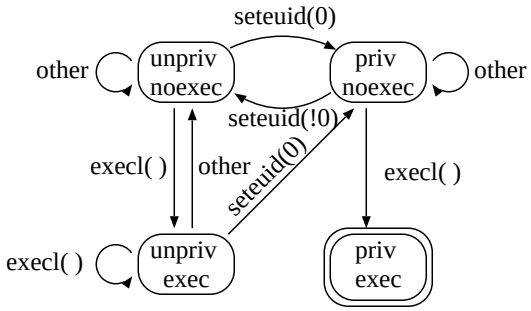
[3]The complex model is a product automaton of the automata of the simple models



(a) A model of process privilege.



(b) A model of risky system calls.



(c) A composite model describing the property that a process should not make risky system calls while it is in the privileged state. The model is automatically constructed as the product of 5(a) and 5(b). Note that the outgoing transitions from the final state *(priv, exec)* are omitted for clarity.

Figure 5: Building a complex model from simpler models.

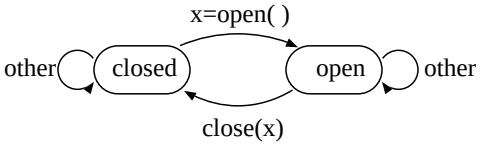
Modularization also makes it possible to reuse existing models. Suppose we have already built the model of process privilege (Figure 5(a)) and we want to build the model for the property that a process should not make risky system calls in the privileged state (Figure 5(c)). Instead of building it from scratch, we only need to build the model describing risky system calls (Figure 5(b)) and then plug in the existing model of process privilege (Figure 5(a)). This allows the construction of a *model library* which supplies building blocks for new models.

Enabling modularity is very important for practical use. For ease of presentation, we have so far described only security properties that have concise representations as small FSAs, but in practice our security models may be very complex. For instance, our model of user IDs in Linux has dozens of states and many more transitions. If we had to re-specify this every time we wanted to check some security property that involves privileges, the result would be too unwieldy for practical use. Modularity comes to the rescue here: it lets us build a few base models once, then we can compose and extend them in many interesting ways.

3.1.2 Pattern Variables

MOPS is control flow and path sensitive but data flow insensitive. In other words, we ignore most data flow: for instance, when processing an `if-then-else` statement, we conservatively assume that either branch could be taken, and we do not try to analyze whether the condition to the `if` statement is true or not. We make this choice for the following reasons. First, we conjecture that many security properties do not require the analysis of data flow. Second, analysis of data flow is expensive and will severely limit the scalability of MOPS. Third, we can do rudimentary data flow analysis by encoding data values into a security model. For example, if we want to analyze the value of a boolean variable b, then we can split each state s_i in the security model into two states $s_{i,0}$ and $s_{i,1}$, where $s_{i,0}$ represents when b is true and $s_{i,1}$ represents when b is false.

MOPS supports a special form of data flow analysis via pattern variables. A pattern variable used in an FSA may be bound to any expression that satisfies context constraints in a program. For example, if *x* is a pattern variable in the FSA in Figure 6(a), then *x* can be bound to the expression either *a* or *b* in the program in Figure 6(b). In other words, pattern variables enable syntactic matching.



(a) An example of a security property using pattern variables.

```
int main()
{
    int a, b;
    a = open("foo", O_RDONLY);
    b = open("bar", O_RDONLY);
    ...
    close(a);
    close(b);
}
```

(b) An example of a program motivating the utility of pattern variables. By using the pattern variable to match up each *open* call to its corresponding *close* call, we can accurately track the state of each file descriptor.

Figure 6: An example showing the use of pattern variables

3.2 Modeling Programs

Since we only care about all feasible paths in a program and the statements executed on these paths, we can model the execution of the program by a pointer and a stack. The pointer points to the program position of the next statement to be executed, and the stack records the return addresses of all unfinished function calls. Therefore, the value of the pointer and the values on the stack uniquely identify a snapshot of the program in execution. If we merge the pointer and the stack by regarding the pointer as the top element on the stack, we get a Pushdown Automaton(PDA). The control flow in the program determines the transitions in the PDA. An algorithm that constructs the PDA from the program is described in Section 5.

Once we have an FSA describing a security property and a PDA representing a program, our goal is to check if any risky state in the FSA is reachable at any program point in the PDA. To answer this question, MOPS *composes* the FSA M with the PDA P using standard techniques [14, Theorem 6.5]. This results in a new PDA, called the *composite PDA*, which accepts the language $L(M) \cap L(P)$. The initial configuration of the composite PDA represents the snapshot when the program starts, where the state of the PDA is the initial state of the security model and the stack of the PDA only contains the entry point of the program. By using model checking techniques, MOPS can determine if any risky state is reachable within the composite PDA. If this is the case, then MOPS has found a potential security violation and outputs an execution path in the program that causes this violation. For example, MOPS finds that the path in Figure 7 from the program in Figure 4 violates the security property in Figure 3(a).

Furthermore, MOPS can determine, for each statement in a program, all the states in an FSA that the statement can be executed in. For example, if the FSA contains a privileged and an unprivileged state, MOPS tells which statement may be executed in the privileged state. If this does not meet a programmer's expectation, a vulnerability is likely. We speculate that this additional functionality may be very useful when auditing security-critical programs by hand.

Function	**Program Point**	**Statement**
$main$	m_0:	do_something_with_privilege();
$main$	m_1:	drop_privilege();
$drop\_privilege$	d_0:	if ((passwd = getpwuid(getuid())) == NULL)
$drop\_privilege$	d_1:	return; // but forget to drop privilege!
$main$	m_2:	execl("/bin/sh", "/bin/sh", NULL); // risky system call

Figure 7: An execution path that causes a security violation, from the program in Figure 4.

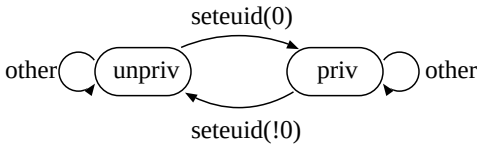


Figure 8: A simplified FSA describing process privilege in Linux 2.4.17

4 Modeling Operating Systems Semantics

Since a security model is an abstract representation of the security operations in an operating system, we need to understand the semantics of the security operations precisely to construct an accurate security model. This, however, is often difficult because the semantics of security operations is subtle and varies among different operating systems (such as different flavors of the Unix system). Moreover, their documentation is sometimes incomplete or incorrect [8].

We advocate relying on the kernel code for the construction of security models, since the kernel code determines the semantics of the security operations. We adopt a two step process: in the first step, find out all the kernel variables that affect the security operations and then determine the states in the FSA based on these kernel variables; in the second step, determine the transitions among these states in the FSA. The first step can usually be done by hand, but manually doing the second step is often laborious and error prone because of the large number of transitions. We tackle this problem by writing a state-space explorer that exhaustively executes all the security operations on the operating system and automatically creates all the transitions in the security model.

To illustrate this process, we will show how to build a security model that describes the transition of privilege in a process in Linux 2.4.17. Further details may be found in a companion paper [8].

4.1 A simple model

Since the privilege of a process is carried in its *euid*, we start with a simple model with two states: the privileged state *priv* representing when the *euid* is zero and the unprivileged state *unpriv* representing when the *euid* is non-zero, as shown in Figure 8. The call *seteuid(0)* causes a transition from *unpriv* to *priv* and any call to *seteuid* with a non-zero argument, denoted by *seteuid(!0)*, causes a reverse transition. Furthermore, the *euid* can also be changed by the *setuid*, *setreuid*, and *setresuid* system calls, so they are also added into the model (not shown in Figure 8 for legibility). We will refer to these system calls that modify the user IDs of a process as the *uid-setting system calls*.

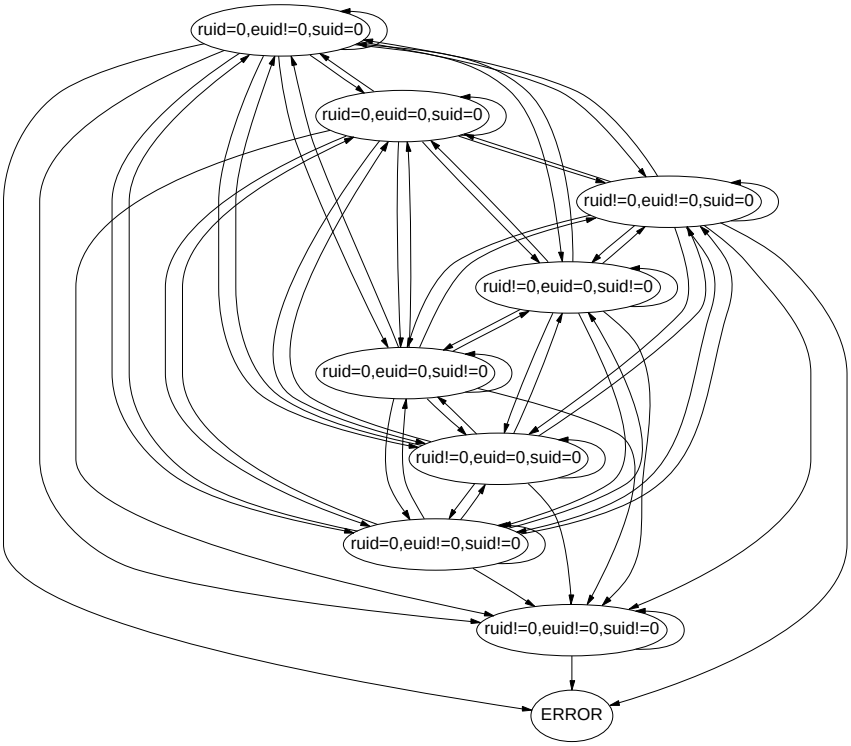


Figure 9: A refined view of the Linux 2.4.17 process privilege model, capturing the *ruid*, *euid*, and *suid*.

4.2 Improving the model

The above simple model, however, is inaccurate. The behavior of the uid-setting system calls depends not only on the *euid* but also on the *ruid* and *suid* [8]. Therefore, we extend the model to consider all the three user IDs. The range of values in each user ID determines the number of states in the model. Typically, a process switches its user IDs between *root*, whose user ID is zero, and a non-root user, whose user ID is non-zero. In this case, the model needs eight states to describe all possible combinations of the values in the three user IDs. In addition, there is an *error* state, which represents a failed system call. This resulting model is shown in Figure 9. For legibility, all input symbols (system calls) on the transitions are omitted.

To verify if the states in this model are complete, we need to find out if other kernel variables besides the *ruid*, *euid*, and *suid* of a process can affect the behavior of the uid-setting system calls. A search through the kernel finds that the effective capability (*cap_effective*), the permitted capability (*cap_permitted*), and the variable *keep_capabilities* of a process are also relevant. Like the *ruid*, *euid*, and *suid*, they are also per-process variables. We add *cap_effective* and *cap_permitted* into the state space, each of which is represented as a binary value. We ignore *keep_capabilities* because few programs modify it and we let MOPS warn about such programs.

4.3 Determining Transitions

Having determined the states in the security model, the next step is to create transitions in the model. However, it would be too laborious and error prone to create the huge number of transitions in Figure 9 and Figure 10. Instead, we write a state-space explorer that creates the transitions automatically. From each state in the FSA, the explorer determines all the outgoing transitions from this state by making all the uid-setting system calls from this state and examining the state transitions resulting from the calls. A proof

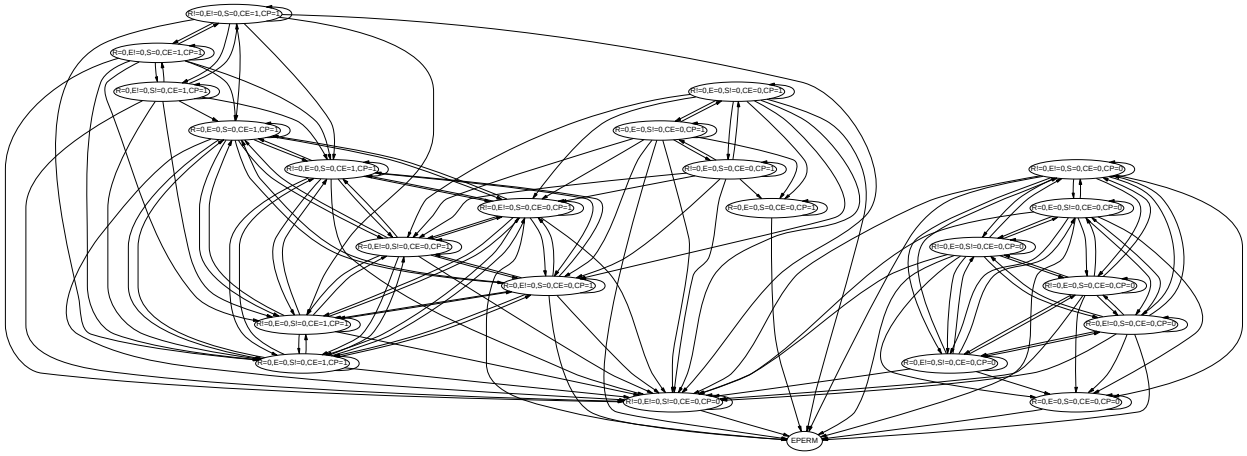


Figure 10: A further refined model of process privilege in Linux 2.4.17, this time capturing all of the *ruid*, *euid*, *suid*, effective and permitted capabilities.

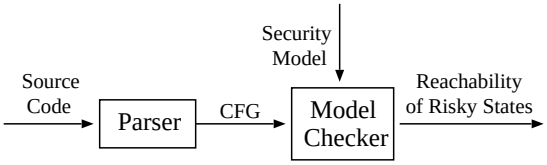


Figure 11: The high-level architecture of MOPS.

of correctness of this approach and other details may be found elsewhere [8].

5 Algorithms

MOPS consists of a parser and a model checker, as shown in Figure 11. MOPS checks whether a source program satisfies a security property by the following steps: first, the parser builds a Control Flow Graph (CFG) of the program; then, the model checker builds a PDA from the CFG and checks the PDA against the security property.

5.1 Parser

The parser builds a CFG from a source program. Each edge in the CFG represents a statement in the program by an Abstract Syntax Tree (AST), and each node in the CFG represents a program point. The parser is derived from RC [12], which is based on *GCC*. By using a *GCC*-derived parser, MOPS is able to parse any source program that *GCC* parses. If the program consists of multiple source files, MOPS merges the multiple CFGs each of which is generated from one source file into a single CFG.

As source programs get larger, the sizes of their CFGs increase rapidly. For example, *sendmail* 8.12.0, a popular Internet mail program with 53k lines of code, has a CFG with 182k nodes and 197k edges. Most model checkers cannot handle such a large CFG. Therefore, we must find a way to drastically compact large CFGs.

We propose a novel technique for compacting CFGs because we have observed that, for most programs and most security properties, the majority of the statements in the program are irrelevant to the security property. For example, since a security property describing the transition of privileges in a Unix process

1. Find all the relevant functions by computing the transitive closure of relevant functions.
2. For each node v in the CFG, attach a bit vector b_v with all bits clear.
   /* In the following, $b_v[i]$ denotes the ith bit of the bit vector attached to the node v */
3. Set a variable *msb* $:= 0$.
4. Identify the entry node v. Set $b_v[0] := 1$. Add v to a work queue W.
5. While the work queue W is not empty, do:
   Pick a node v from W.
   If any incoming edge to v has a relevant statement, then
   If all the bits in b_v are clear, then
   Set *msb* $:=$ *msb* $+ 1$ and $b_v[$*msb*$] := 1$.
   else
   For each preceding node u of v, do
   Set $b_v := b_v \cup b_u$.
   If any bit in b_v has been changed in this iteration, then
   Add all the succeeding nodes of v into the work queue W.
6. Remove every edge $v \rightarrow v'$ where $b_v = b_{v'}$ and merge v with v'.
7. Remove the bodies of all the irrelevant functions.

Figure 12: An algorithm for compacting CFGs

cares only about the uid-setting system calls, all the other statements in the program are irrelevant to the security property. We describe the observation formally as follows. With regard to a security property, we define a *relevant function* as a function that is defined by a program and that contains at least a *relevant statement*, and we define a *relevant statement* as a statement that may trigger a state change in the security model or that is a call to a relevant function. We consider two program paths to be *equivalent* with regard to a security property if they contain the same sequence of relevant statements. For example, if the first path contains the sequence [*A D E B F C*] , the second path contains the sequence [*A D B C*], and only the statements *A*, *B*, and *C* are relevant to the security property, then the two paths are equivalent with regard to the security property.

Compacting a CFG should not introduce any imprecision to the analysis. Intuitively, this requires that the result of running the model checker on a program and a property is always guaranteed to be the same as running on the compacted version of the program and the property. Formally, it requires that for each path in the uncompacted CFG there must exist an equivalent path in the compacted CFG, and vice versa. The following algorithm satisfies this requirement: first, identify all the relevant statements by computing the transitive closure of all the relevant functions second, decide which edges can be safely shrunk by attaching a bit vector to each node in the CFG and compute values for these bit vectors; finally, compact the CFG by (1) shrinking all the edges whose source node and destination node have the same value in their bit vectors, and (2) removing the bodies of all but the relevant functions. Figure 12 describes this algorithm in details.

The time complexity of this algorithm is $O(MN)$, where M is the number of relevant statements and N is the number of nodes in the CFG. Since the CFG of each function body is disjoint and MOPS runs this algorithm on each of them separately, the complexity of this algorithm does not depend on the number of function bodies in the program.

Compaction can reduce the sizes of CFGs substantially. For example, the uncompacted CFG of *sendmail* has 1742 function bodies, 182k nodes, and 197k edges, which is too large for MOPS's model checker to handle. After we compacted it with regard to the security property in Figure 9, we have reduced the CFG to only 37 function bodies, 240 nodes, and 670 edges. MOPS's model checker successfully ran on the compacted CFG.

5.2 Model Checker

Taking a CFG generated by the parser and a security model represented by an FSA, the model checker decides if the program of the CFG may violate the security property in four steps. First, it constructs a PDA for the CFG by adding a transition to the PDA for each edge in the CFG according to the following rules (the PDA has a single state s):

- For an edge from a program point p_1 to p_2 with a statement i:
  - If i is not a function call, add a transition $(s, p_1) \xrightarrow{i} (s, p_2)$ to the PDA.
  - If i is a call to a function f, add a transition $(s, p_1) \xrightarrow{\epsilon} (s, f_0 p_2)$ to the PDA, where f_0 is the entry point of the function f. [4]
- For an edge that is a return statement from a function f, add a transition $(s, f_n) \xrightarrow{\epsilon} (s, \epsilon)$ to the PDA, where f_n is the exit point of the function f and ϵ denotes that no symbol is pushed onto the stack.

Second, the model checker computes the intersection of the security model with the program PDA by taking their parallel composition [14], which creates a new PDA (called the *composite PDA*), whose states come from the FSA and whose input symbols and stack symbols come from the PDA, by the following algorithm [5]:

- For each transition $(s, p_1) \xrightarrow{i} (s, p_2)$ in the program PDA and each transition $s_1 \xrightarrow{i} s_2$ in the security model, add a transition $(s_1, p_1) \rightarrow (s_2, p_2)$ to the composite PDA.
- For each transition $(s, p_1) \xrightarrow{\epsilon} (s, p_2 p_3)$ in the program PDA and each state s_1 in the security model, add a transition $(s_1, p_1) \rightarrow (s_1, p_2 p_3)$ to the composite PDA.
- For each transition $(s, p_1) \xrightarrow{\epsilon} (s, \epsilon)$ in the program PDA and each state s_1 in the security model, add a transition $(s_1, p_1) \rightarrow (s_1, \epsilon)$ to the composite PDA.

Note that we have dropped the input symbols in the composite PDA because we only care about its state reachability, not about its acceptable languages. The initial configuration of the composite PDA is (s_0, p_0) where s_0 is the initial state of the security model and p_0 is the entry point of the program (usually the entry point of the function *main*).

Third, the model checker computes all the reachable configurations from the initial configuration of the composite PDA. The set of reachable configurations could be very large, or even infinite, so representing it internally is a challenge. Fortunately, there is a beautiful theorem that comes to the rescue: the reachable configurations of a PDA form a regular language, and hence can be represented by an FSA. Our model checker represents the set of reachable states using a *P-automaton*.

A P-automaton (PA) is an FSA that describes the reachable configurations of a PDA, and it can be computed effectively using an algorithm described elsewhere [10]. To construct the PA of a PDA, we take all the states from the PDA and make them the initial states of the PA, and we add a final state to the PA. Then, we take all the stack symbols from the PDA and make them the input symbols of the PA. A path in the PA from an initial state s to the final state with input symbols $[i_1, \ldots, i_n]$ on the edges represents

[4] We only consider a function call if we have the source code of the function body (so we can build its CFG). This implies that we will not consider library calls whose source code is unavailable to us.

[5] This algorithm works only if there is no ϵ-transition in the FSA, which is guaranteed due to the way by which FSAs are constructed in MOPS.

the configuration $(s, i_1 \dots i_k)$ in the PDA [6]. This definition shows clearly how to add edges to the PA to represent an initial configuration of the PDA. Call the resulting PA PA_0. From PA_0, the following algorithm constructs post*(PA_0), a PA that contains all the successors of the initial configuration. In other words, post*(PA_0) contains all the reachable configurations of the PDA.

- For each transition $t = (p_1, r_1) \rightarrow (p_2, r_2 r_3)$ in the PDA, add a new state p_t and a new edge $p_2 \xrightarrow{r_2} p_t$ to PA_0.
- Add new edges to PA_0 according to the following saturation rules:
  - If $(p_1, r_1) \rightarrow (p_2, \epsilon)$ is in the PDA and $p_1 \xrightarrow{r_1} q$ is in PA_0, add an edge $p_2 \xrightarrow{\epsilon} q$.
  - If $(p_1, r_1) \rightarrow (p_2, r_2)$ is in the PDA and $p_1 \xrightarrow{r_1} q$ is in PA_0, add an edge $p_2 \xrightarrow{r_2} q$.
  - If $t = (p_1, r_1) \rightarrow (p_2, r_2 r_3)$ is in the PDA and $p_1 \xrightarrow{r_1} q$ is in PA_0, add an edge $p_t \xrightarrow{r_3} q$.

The space and time complexity of this algorithm ranges from $O(SP^2)$ to $O(S^3P^2)$ where S is the number of states in the security model, and P is the number of statements in the compacted CFG.

Finally, the model checker tests whether the state of any reachable configuration is a risky state of the security model. If not, the program satisfies the security property. Otherwise, the program may violate the security property.

5.2.1 Backtracking

When the model checker determines that a program may violate a security property, it is useful to identify a program path (sometimes known as an error trace) on which the violation occurs. We call this *backtracking*, and we have implemented a novel algorithm for extending the post*() computation described above to support backtracking. The algorithm attaches to each edge in the PA a predecessor list that records all the edges that have triggered this edge to be added during the post*() computation. For example, if the edge $p_1 \xrightarrow{r_1} q$ in PA_0 and the transition $(p_1, r_1) \rightarrow (p_2, r_2)$ in the PDA trigger a new edge $p_2 \xrightarrow{r_2} q$ to be added, then the algorithm adds $p_1 \xrightarrow{r_1} q$ to the predecessor list of the new edge. Intuitively, since the input symbol on an edge in the PA represents a program point, the predecessor list of the edge records all the program points that can immediately precede the current one in an error trace. Therefore, once we find a reachable PDA configuration violating the security property, we can backtrack all its preceding configurations one by one using the predecessor lists in the PA.

6 Applications

6.1 Checking Privilege Flow in Non-local Control Flow

6.1.1 Problem

POSIX allows a program to do a non-local jump by calling *longjmp*, in which the program jumps to the stack context saved by a previous *setjmp* call. Since non-local jumps are not in the Control Flow Graph of the program, most program analysis tools cannot analyze them. However, non-local jumps are prone to security vulnerabilities since they may cause unexpected control flow. For example, the program in Figure 13 starts with privilege. Then, it drops privilege (by calling *seteuid(getuid())*) before doing potentially risky operations in the function *main*. However, if the program subsequently receives a signal, the *longjmp* call in

[6] To avoid confusion, we use the word *edge* to refer to transitions in P-automata and reserve the word *transition* for transitions in PDAs

```
jmp_buf env;

void signalhandler()
{
    seteuid(0);
    logwtmp(message, " ", " ");
    longjmp(env, 1);
}

int main()
{
    // drop privilege
    seteuid(getuid());
    ...
    setjmp(env);
    // do something potentially risky
    ...
}
```

Figure 13: A program with a security vulnerability caused by *longjmp* carrying privilege to the call site of *setjmp*, where privilege should have been dropped

the function *signalhandler* will cause the program to jump into the function *main* (immediately after the call site of *setjmp*) with the privilege obtained in *signalhandler*. Thereafter, the program will execute potentially risky operations in the function *main* with privilege.

6.1.2 Temporal Safety Property

To prevent a *longjmp* call from carrying privilege to the call site of a *setjmp* where privilege should have been dropped, we propose the following temporal safety property:

> Property 4: the privilege of a process when it calls *longjmp* must match its privilege when it calls *setjmp*.

Obviously, an FSA describing this property should have two dimensions: one dimension records the privilege of the process when it last called *setjmp*, and the other records its current privilege. The states in the FSA whose privileges in the two dimensions are different represent violation of this property. Formally, let F be the FSA describing the transition of privilege in a process (constructed in Section 4) and let S be the set of states in F. We derive an FSA G that describes the above property. The states in G are $S \times (S \cup \{\perp\}) \cup \{\text{ERROR}\}$, where $\times$ denotes the Cartesian product and $\perp$ represents the uninitialized state (the *setjmp* buffer is in the uninitialized state before the first *setjmp* is called). Use the following rules to add transitions to G.

- For every transition $u \xrightarrow{i} v$ in F and every state $s \in (S \cup \{\perp\})$, add a transition $(u, s) \xrightarrow{i} (v, s)$ to G .
- For every state $s \in S$ and $t \in (S \cup \{\perp\})$, add a transition $(s, t) \xrightarrow{\textit{setjmp(env)}} (s, s)$ to G.
- For every state $s \in S$ and $t \in (S \cup \{\perp\})$ and $s \neq t$, add a transition $(s, t) \xrightarrow{\textit{longjmp(env, *)}} \mathsf{ERROR}$ to G where the state ERROR indicates violation of the property.

- For every state $s \in S$, add a transition $(s, s) \xrightarrow{\textit{longjmp(env, *)}} (s, s)$ to G.

6.1.3 Implementation

We used the above security model to find a known security vulnerability in *wu-ftpd* version 2.4 [6]. The vulnerability is similar to the one in Figure 13, except that *seteuid(0)* and *longjmp(env)* are called in the handlers for the signals SIGPIPE and SIGURG respectively. Therefore, by sending the signal SIGURG immediately after the signal SIGPIPE to a *wu-ftpd* process, an attacker can cause the process to call *seteuid(0)* in the handler of the signal SIGPIPE to gain privilege, and then to call *longjmp(env)* in the handler of the signal SIGURG to return to the call site of *setjmp(env)* in the function *main*. Thereafter, *wu-ftpd* will execute with root privilege, which results in giving the attacker root privilege.

Since this vulnerability involves signal handling which is not part of the control flow of a program and which most program analysis tools, including MOPS, are unable to handle, at present we need to manually insert the control flow of signal handling into the program. A naive approach would be to non-deterministically add a call to a signal handler after every statement in the program wherever the signal is enabled. This is too laborious. Fortunately, there is a better approach. We observe that it is sufficient to add such calls only after the statements that may trigger state changes in the FSA. Since only the uid-setting system calls and the *setjmp* call may trigger transitions in the above FSA, we only need to non-deterministically add a call to a signal handler after the uid-setting system calls and the *setjmp* call in the program wherever the signal is enabled. This substantially reduces the number of calls added to the program. The need to modify the program by hand is a repairable limitation of our current implementation, not a fundamental limitation of the approach. It would be straightforward to extend the control flow analysis to add transitions for signal handlers as needed automatically, and we hope to add this to a future version of MOPS.

Since *longjmp(env, *)* causes a program to jump to the stack context in *env* which has been saved by *setjmp(env)*, if the program uses multiple jump buffers, we need to match every *longjmp* with its corresponding *setjmp*. Pattern variables (Section 3.1.2) handle this naturally, so long as there is no aliasing.

6.1.4 Results

MOPS detected the vulnerability in *wu-ftpd* 2.4 beta 11 and discovered the offending path that was given in the report of the vulnerability [13].

wu-ftpd version 2.4 beta 12 fixed the vulnerability by safeguarding every *seteuid* call with enabling/disabling signals. This new version precedes every call to gain privilege (*seteuid(0)*) with a call to disable signals and follows every call to drop privilege (*seteuid(!0)*) with a call to enable signals. We used MOPS to verify that this new version satisfies Property 4, as given above.

6.2 Checking Proper Dropping of Privilege

Many server processes start with root privilege in their user IDs. They often need to drop privilege temporarily before doing untrusted operations on a user's behalf or to drop privilege permanently before passing control to the user. Failure in dropping privilege may allow an attacker to take control of the application or even the OS.

To detect this vulnerability, we need to find which statements in the program may be executed with privilege. Using the techniques described in Section 4, we built an FSA for describing privilege transitions in each process on Linux (Figure 9 shows the FSA, where for clarity all the labels of the transitions are removed). Each state in the FSA encodes whether the root privilege is present in the *ruid*, *euid*, and *suid*. By using MOPS to find, for each statement in the program, the set of states in the FSA that the statement may be executed in (Section 3.2), we are able to identify all the statements that may be executed with privilege,

and therefore to determine whether each operation that intends to drop privilege may fail. By this approach, we identified two known vulnerabilities in *sendmail*: *sendmail* 8.10.1 fails to drop root privilege in user IDs permanently due to a bug in the Linux kernel and an unexpected interaction between the user IDs and the capabilities [17], and *sendmail* 8.12.0 fails to drop privilege in group IDs permanently due to an unexpected interaction between the user IDs and the group IDs [20]. More details of these vulnerabilities may be found elsewhere [8].

6.3 Verifying Success of System Calls

Failure of certain security related system calls may cause vulnerability. For example, if *setuid(getuid())* fails, the calling process fails to drop privilege permanently which may allow an untrusted application to take over the OS. We obtain the following security property:

Property 5: the *setuid* system call should never fail.

The FSA that we built for modeling uid-setting system calls includes a state that represents failed calls. With this FSA, MOPS is able to verify that no uid-setting system calls may fail in OpenSSH 2.5.2.

6.4 Performance

We measured the performance of MOPS by sendmail 8.12.0, which has 53k lines of code, in the experiment described in Section 6.2. On an 1.5GHz Pentium machine, MOPS spent 110 seconds in parsing the source files and 95 seconds in model checking. This computation needed less than 300MB of memory. This suggests that MOPS will scale well to large security-relevant programs.

7 Discussion

The two major goals of MOPS are soundness and scalability. Soundness will enable MOPS to be used not only as a bug-finding tool but also as a property-verification tool. To evaluate the soundness of MOPS, let us look at the two stages of MOPS: transforming a C program into a PDA, and model checking the PDA. The latter stage is always sound. The former stage is sound as long as every execution path in the program is captured in the PDA. This requires that the program be a portable, single-threaded C program that has no implementation-defined behavior: for example, no buffer overruns and no runtime code generation. In addition, MOPS ignores control flow by function pointers, signal handlers, and non-local jumps via setjmp/longjmp. Although this approximation introduces unsoundness, it is not a fundamental limitation of the approach but rather a limitation of the current implementation. We can overcome this problem by manually transforming the control flow that MOPS ignores to the equivalent ones that MOPS considers, as we did in Section 6.1.3. We are working on automating this process and we hope to add it to a future version of MOPS.

Scalability will enable MOPS to work on a broad range of programs, especially the more complex ones which are more error-prone. MOPS has achieved high scalability by disregarding most data flow and compacting the CFGs very efficiently. This advantage, however, comes with the price of lower precision: MOPS may mistakenly consider paths that are infeasible in the program to be feasible, and issue extraneous warnings. Although there is always a trade-off between scalability and precision, we are investigating how much we can push MOPS's precision without sacrificing scalability.

8 Related Work

A number of static analysis techniques have been used to detect specific security vulnerabilities in software. Wagner et al. used integer range analysis to find buffer overruns [19]. Koved et al. used context sensitive, flow sensitive, interprocedural data flow analysis to compute access rights requirement in Java with optimizations to keep the analysis tractable [16]. CQUAL [11] is a type-based analysis tool that provides a mechanism for specifying and checking properties of C programs. It is used to detect format string vulnerabilities [18] and to verify authorization hook placement in the Linux Security Model framework [21], which are examples of the development of sound analysis for verification of particular security properties. The application of CQUAL, however, is limited by its flow insensitivity and context insensitivity, although it is being extended to support both.

Metal [9, 1] is a general tool that checks for rule violations in operating systems, using meta-level compilation to write system-specific compiler extensions. The goals of Metal and MOPS are different. Metal is aimed at finding bugs with few false positives. Therefore, false negatives are quite possible and it is neither sound nor complete. On the other hand, MOPS is aimed at verifying security properties with no false negatives, which is achieved by its soundness (modulo the mild assumptions discussed in Section 7). Moreover, Metal is primarily an intra-procedural tool — inter-procedural checking requires extra effort from the user. However, since interprocedural bugs are more elusive, automated tools become more valuable when they find interprocedural bugs. MOPS is fully interprocedural.

SLAM [2, 3] is a pioneer project that uses software model checking to verify temporal safety properties in programs. It validates a program against a well designed interface using an iterative process. During each iteration, a model checker determines the reachability of certain states in a boolean abstraction of the source program and a theorem prover verifies the path given by the model checker. If the path is infeasible, additional predicates are added and the process enters a new iteration. SLAM, however, does not yet scale to very large programs. Compared to SLAM, MOPS trades precision for scalability and efficiency by considering only control flow and ignoring most data flow, as we conjecture that many security properties can be verified without data flow analysis. Also since MOPS is not an iterative process, it does not suffer from possible non-termination as SLAM does.

Jensen et al. model checked a special class of security properties in Java using only control flow analysis [15, 4]. Its algorithm, however, requires that one specifies a fixed, finite bound on the size of the program stack. The model checking algorithm in MOPS is based on the work by Esparza [10], which properly handles stacks of unbounded size. We have extended the algorithm with backtracking and CFG compaction.

9 Conclusions

In this paper, we have described a formal approach that is able to check a wide range of security properties in large programs efficiently. We have implemented this approach in a tool called *MOPS*. In our approach, we identify rules of safe programming practice, encode them as security properties, and describe them by Finite State Automata (FSA). To check these properties in a program, MOPS models the program as a pushdown automaton (PDA) and uses model checking techniques to determine the reachability of risky states in the PDA. The major advantages of this approach are: (1) since it is fully interprocedural, it is especially useful in finding interprocedural bugs, which are more likely to elude manual audit; (2) since it is sound (modulo mild assumptions), it can reliably catch all bugs of the specified types; (3) thanks to our novel compaction algorithm, MOPS is efficient and scales to handle large programs. Preliminary evidence suggests that MOPS will be helpful in finding various types of security vulnerabilities in C programs.

We are working on extending MOPS. We are investigating how much data flow analysis we can incorporate into MOPS without affecting its scalability. We are also experimenting with checking more security

properties in more programs so that we can improve MOPS as we gain more experience.

10 Acknowledgment

We thank Drew Dean for suggesting the security property regarding *setjmp* and *longjmp*. Robert Johnson helped with the initial implementation of pattern variable and David Schultz helped improve the usability of MOPS. We are grateful to Zhendong Su, David Schultz, Naveen Sastry, Dawn Song, Helen Wang, and the anonymous reviewers for their valuable comments.

11 Availability

MOPS is available at `http://www.cs.berkeley.edu/~daw/mops/`

References

[1] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of IEEE Security and Privacy 2002*, 2002.

[2] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001, Workshop on Model Checking of Software*, 2001.

[3] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL 2002*, 2002.

[4] F. Besson, T. Jensen, D. L. Metayer, and T. Thorn. Model checking security properties of control flow graphs. *Journal of Computer Security*, 9:217–250, 2001.

[5] M. Bishop and M. Dilger. Checking for race conditions in file access. *Computing Systems*, 9(2):131–152, 1996.

[6] CERT. CERT Advisory CA-1997-16: ftpd signal handling vulnerability. `http://www.cert.org/advisories/CA-1997-16.html`.

[7] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. Technical Report UCB//CSD-02-1197, UC Berkeley, 2002.

[8] H. Chen, D. Wagner, and D. Dean. Setuid demystified. In *Proceedings of the Eleventh Usenix Security Symposium*, San Francisco, CA, 2002.

[9] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, 2000.

[10] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. Technical report, Technische Universität München, 2000.

[11] J. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*, May 1999.

[12] D. Gay and A. Aiken. Language support for regions. In *ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, June 2001.

[13] D. Greenman. Serious security bug in wu-ftpd v2.4. http://online.securityfocus.com/archive/1/6056/1997-01-04/1997-01-10/2.

[14] J. Hopcroft and J. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 1979.

[15] T. Jensen, D. L. Metayer, and T. Thorn. Verification of control flow based security properties. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, 1999.

[16] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for java. In *Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2002.

[17] Sendmail Inc. Sendmail workaround for linux capabilities bug. http://www.sendmail.org/sendmail.8.10.1.LINUX-SECURITY.txt.

[18] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[19] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of NDSS 2000*, 2000.

[20] M. Zalewski. Multiple local sendmail vulnerabilities. http://razor.bindview.com/publish/advisories/adv_sm812.html.

[21] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for static analysis of authorization hook placement. In *Proceedings of the Eleventh Usenix Security Symposium*, August 2002.