Lightweight Model Checking for Improving Software Security

by

Hao Chen

B.Eng. (Southeast University) 1991
M.Eng. (Southeast University) 1994

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor David A. Wagner, Chair
Professor Alex Aiken
Professor Ray Larson

Fall 2004

The dissertation of Hao Chen is approved:

Chair　　　　　　　　　　　　　　　　　　　　　　　Date

Date

Date

University of California, Berkeley

Fall 2004

Lightweight Model Checking for Improving Software Security

Abstract

Lightweight Model Checking for Improving Software Security

by

Hao Chen

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David A. Wagner, Chair

Software assurance is a grand challenge in computer security: software vulnerabilities are responsible for many recurring attacks on computer systems. However, developers have few effective means of improving software security. Manual code inspection is tedious and unreliable. Complete formal verification is an automated approach that provides high assurance for software systems; however, it does not scale to large programs and it requires high expertise for writing specifications. As a result, software developers acutely need an effective yet practical approach that they can use for discovering and preventing security bugs in software development process.

This dissertation shows that lightweight model checking is an effective, practical approach for improving software security. First, we designed a formal framework for checking source code for violations of security properties. Then, we implemented the framework in a tool called MOPS and developed the following key features for making MOPS practical: it scales to large programs; it allows ordinary programmers to write specifications; it reports concise, comprehensive error traces;

and it integrates into software build process automatically. To validate our design goal of MOPS, we used MOPS to audit both application and kernel code: we ran MOPS on 60 million lines of C programs in RedHat Linux, during which we discovered dozens of security bugs, and we used MOPS to verify design invariants in the EROS kernel.

Our success shows that lightweight model checking is a powerful, practical approach for improving security of large software systems.

Professor David A. Wagner
Dissertation Committee Chair

To my mother,

Qingzhu Chen.

Contents

List of Figures

List of Tables

Acknowledgments

I have benefited tremendously from many people throughout my Ph.D. program.

First and foremost, I am deeply indebted to my advisor David Wagner. I am incredibly fortunate to have had the opportunity to work with one of the best minds in computer security research. A role model of a beloved mentor, teacher, and colleague, Professor Wagner inspired me to pursue an academic career to nurture my own students with the great cares that I have received from him.

Professor Jonathan Shapiro was patient enough to be the first outside user of MOPS, despite the usability problems in early versions of MOPS. From him, I learned how to think critically, speak confidently, and write convincingly. Dr. Drew Dean was instrumental in starting the experiments of running MOPS on large, security-sensitive programs. His encyclopedic knowledge of Unix security helped find fertile applications for MOPS. My qualifying and dissertation committees, consisting of Professors Alex Aiken, Ray Larson, Doug Tygar, and David Wagner, have been very helpful in guiding my research and dissertation.

I would like to thank my officemates who served as peer mentors, collaborators, and references on the English language: Monica Chew, Robert Johnson, Chris Karlof, Yaping Li, Ben Liblit, Naveen Sastry, Umesh Shankar, Jimmy Su, and Daniel Wilkerson. Special thanks go to our English czar Ben Liblit, who can often not only judge but also explain proper use of the English language.

Many people have contributed to parts of this work. David Wagner, Benjamin Schwarz, Geoff Morrison, Jacob West, Jeremy Lin, and Wei Tu participated in the joint work of running MOPS on the entire RedHat Linux distribution. David Schultz and Geoff Morrison helped integrate

MOPS into the build processes of software packages and designed the user interface for displaying MOPS's error reports. The work on the formal models of the `setuid` family system calls started from a chat with Drew Dean and David Wagner. David Gay graciously provided and supported the C parser used by MOPS. As a developer of EROS, Jonathan Shapiro served as the first outside user of MOPS and, arguably, as an "ordinary programmer" for evaluating MOPS's usability. Robert Johnson implemented pattern variables in early versions of MOPS.

During my Ph.D. program, I have been supported directly or indirectly by Microsoft Research and SRI International. I am grateful to James Larus and Manuvir Das at Microsoft Research and Peter Neumann at SRI International for their insightful comments on my work.

My life during my Ph.D. program would have been less happy without many great friends, but I only have space to list a few: Hao Zhang, Hanchuan Peng, Yan Chen, Zhendong Su, Li Zhuang, and Feng Zhou. Our Friday afternoon weight lifting and Sunday afternoon running sessions kept me sane during the often-stressful dissertation work.

Finally, I would like to thank my family. My mom and my grandpa, who were the dearest people to me, passed away due to cancer during my Ph.D. program. It was my dad and my sister who helped me go through these difficult times. Their unwavering support helped strengthened my will and improve my work.

Chapter 1

Introduction

1.1 Problem statement

From the dawn of the computing era, software has performed in ways surprising to its authors. Some surprises are due to design flaws, but most surprises are due to programming errors, commonly called *bugs*. Right from the beginning, computer security researchers have recognized that assurance would be a critical challenge for the field, so much work has gone into formal methods for verifying programs. However, formal methods have yet to make it into mainstream software development, be it commercial or open source. Therefore, software in the 21st century still has bugs, and this seems likely to remain true for the foreseeable future.

All software bugs are not created equal. Some bugs cause minor annoyance, such as crashing the browser. More seriously, some bugs cause loss of work, such as crashing your editor. Yet the most serious bugs cause security violations, such as allowing an unauthorized user to access your systems and to read, modify, or delete your sensitive data. As a result, these security bugs are often costly, disruptive, and disastrous.

To make progress on security bugs, we first need to understand their causes. Most security bugs are caused by the following reasons:

- Unsafe features in the programming language. A typical example is the lack of array boundary check in C/C++, which may cause the *buffer overflow* bug.

- Program's failure to validate untrusted input data. If the program does not expect the entire range of values in the input data, the unexpected values may cause dire consequences to the program. For example, when an application requests data from the kernel, the application typically invokes a *read* API call and passes the address of a memory region that will receive the data. The kernel must validate that the address is in the application's address space; otherwise, if a malicious application passes an address in the kernel's address space, the kernel will overwrite its own data structures.

- Program's violation of secure programming rules. These rules require programs to perform security relevant operations in certain ways, and violation of these rules may result in vulnerability. For example, programs running with elevated privileges must drop their privileges before executing untrusted user programs. If this rule is violated, the untrusted programs may gain unwarranted privileges, which may allow the untrusted programs to access sensitive information or even take control of the system.

We can solve the first problem — unsafe language features — by improving the programming languages. For example, the Java programming language prevents buffer overflows by checking array boundaries at runtime. We can alleviate the second problem — untrusted input data — by validating input data at runtime.[1] The last problem — violation of secure programming rules

[1]Moreover, we can use static analysis to check if the program always validates untrusted data before using them,

— poses a unique challenge. Since most such rules are language independent, improving programming languages cannot prevent rule violations. Checking these rules at runtime would require the program to keep track of its past security relevant operations, which is tedious and expensive. This difficulty suggests that we should check these rules at compile time: we should check each path in the program to verify that the program satisfies the rule.

How do we do the check? We can check small programs manually, but manually checking large programs is very tedious and often impractical because of the large number of paths in these programs. Therefore, we need automated tools.

1.2 Summary

This dissertation aims at improving software security, and focuses on finding security bugs and verifying their absence in source programs using static analysis. I propose a framework for checking user-specified security properties using lightweight model checking, and describe a tool MOPS[2] for checking these properties in source programs. I have made both theoretical and engineering efforts to make MOPS useful and usable for ordinary programmers. On the theoretical aspect, I designed a new algorithm that reports concise, comprehensible bug findings; on the engineering aspect, I developed tools for integrating the MOPS process into the build process of software packages to allow users to run MOPS on lots of packages very easily. A common but serious usability problem with software auditing tools is that writing specifications for complex properties is tedious and error-prone. To make progress on this problem, I introduced an approach that can generate formal specifications for certain structured security properties automatically, and I used

which is another interesting research topic.

[2]MOdel checking Programs for Security properties

this approach to generate the first formal specification of the `setuid` API on Unix systems. To evaluate MOPS's potential as an effective, practical tool that can be used by ordinary programmers in software development process, I ran MOPS on over one million lines of C programs on Red-Hat Linux and discovered over a dozen new security bugs and weaknesses. I also used MOPS to verify security properties in the EROS kernel. My experience demonstrates that lightweight model checking is a powerful, practical approach for improving software security.

1.3 Contributions

This dissertation makes the following contributions:

- A new framework for checking security properties in software. This framework models the problem of deciding whether a program satisfies a security property as deciding whether the intersection of two languages is empty. This model allows us to use state of the art theories and algorithms from the programming languages community.

- A new algorithm for reporting concise, comprehensible errors in the program. A common problem with many code auditing tools is that they report numerous, redundant errors, which overwhelm users because the users have to review all the errors manually to determine which are real bugs. My algorithm reports only unique errors and, for each error, reports the shortest error trace. This enhancement improves the user's productivity tremendously.

- Tools for integrating MOPS into the build process of software packages on Unix systems seamlessly. These tools interpose on the normal build process to invoke MOPS in the background. This allows the user to run MOPS on lots of software packages easily and automatically.

- An approach for generating formal specifications for certain structured security properties. This approach has made progress on a common problem with code auditing tools that writing specifications for complex properties is tedious and error-prone. Using this approach, I generated the first formal specification of the `setuid` API on Unix systems.

- Large scale auditing on both application and kernel code using MOPS. I have run MOPS on over 60 million lines of application programs and found dozens of security bugs and weaknesses. Using MOPS, I have verified several security properties in the EROS kernel. These experiments demonstrate that MOPS is a powerful, practical tool for improving software security.

Chapter 2

Framework

Our goal is to verify whether programs satisfy secure programming rules, and we can describe many such rules in *temporal safety properties*. A temporal safety property specifies a temporal constraint on a sequence of operations.[1] I will call these temporal safety properties that describe secure programming rules *security properties*.

Example. Unix systems' access control is mainly based on user IDs (*uid*s). Typically, each process has three user IDs: the real user ID (*ruid*), the effective user ID (*euid*), and the saved user ID (*suid*). The real uid identifies the owner of the process, the effective uid represents the privilege of the process (such as its permission to access the file system), and the saved uid provides the process with a means to swap between two user IDs (such as a privileged and an unprivileged user ID). Certain user IDs carry privileges; particularly, the user ID 0 carries the *root privilege*, which gives the process complete control over the system. A secure programming rule requires that a process with the root privilege should drop the privilege before executing an untrusted user

[1] More precisely, if a sequence violates a safety property, the violation must occur in a finite prefix of the sequence; in contrast, a finite sequence cannot violate a *liveness* property.

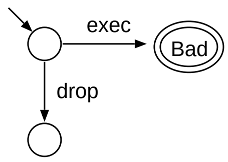


Figure 2.1: An FSA showing the property: *the operation* exec *must follow the operation* drop *in a privileged program*

program. Otherwise, the untrusted user program may regain the privilege and then take control of the system.

Let us formalize this rule in a temporal safety property. This rule has two security-relevant operations: dropping the root privilege (*drop*) and executing an untrusted user program(*exec*). Figure 2.1 illustrates the safety property: *the operation* exec *must follow the operation* drop *in a privileged program*. If the program calls *exec* without calling *drop* first, it will enter the *bad* state. □

2.1 Formal framework

In this section, I will show a formal framework for deciding whether a program satisfies a temporal safety property. I will start with an abstract formal model of the problem. Let Σ be the set of security operations in the property. Let $B \subseteq \Sigma^*$ be all the sequences of these security operations that violate the property (B stands for *bad*). A trace $t \in \Sigma^*$ is a sequence of security operations that the program executes on a path, and the trace is feasible if the path is feasible. Let $T \subseteq \Sigma^*$ denote the set of all feasible traces in the program (T stands for *trace*). If $T \cap B = \emptyset$, the program satisfies the property; otherwise, some feasible traces in the program may violate the property.

This abstract model defines the problem succinctly; however, it does not have a general solution. If B and T are arbitrary languages, whether $T \cap B = \emptyset$ is an undecidable problem. To make the problem decidable, we must restrict the language B and T, as follows.

Let us restrict B, all the sequences of security operations that violate the property, to be a *regular language*. Experience shows that one can describe most temporal safety properties by regular languages. Since B is a regular language, there exists a finite state automaton (FSA) M that accepts B, or $B = L(M)$. We will often use an FSA to identify the security property that it represents.

Although we restrict B to be regular language, we should not restrict T, the set of all feasible traces, to be also a regular language, because a regular language is not powerful enough to describe traces that involve functions calls and returns precisely. For example, if multiple call sites call the same function, a regular language cannot enforce the *context sensitivity* on the traces: a call from a call site must later return to the same call site. Extending regular languages with context sensitivity gives us *context free languages*. Therefore, let us restrict T to be a context free language. It follows that there exists a pushdown automaton (PDA) P that accepts T, or $T = L(P)$.

Now the original problem becomes deciding if $L(M) \cap L(P)$ is empty. Since $L(M)$ is a regular language and $L(P)$ is a context free language, $L(M) \cap L(P)$ is a context free language, and there are efficient algorithms for deciding if a context free language is empty. We will discuss the implementation in the next section.

2.2 Implementation of the framework

To implement the approach described in the previous section, we need to (1) create B, the FSA containing all the sequences of security operations that violate the property, (2) create T, the PDA containing all the feasible traces in the program, (3) compute the intersection $B \cap T$, and

(4) decide if $B \cap T$ is empty. When $B \cap T$ is not empty, we need an extra step of finding program traces that violate the property. Among the five steps, the first step — creating an FSA describing the security property — is done by the user, although we will show an approach that automatically creates FSAs for certain complex but structured properties in Chapter 4. We developed a tool MOPS – MOdel checking Programs for Security properties – for doing the rest of the steps automatically.

2.2.1 Creating an FSA from a security property

The user creates a finite state automaton (FSA) for describing a security property. A *finite state automaton* is a quintuple $\mathcal{F} = (Q, \Sigma, \rightarrow, Q_0, F)$ such that the *state space* Q is a finite set, the *input alphabet* Σ is a finite set, $\rightarrow \subseteq (Q \times \Sigma \times Q)$ is the set of *transitions*, $Q_0 \subseteq Q$ is the set of *initial state*, and $F \subseteq Q$ is the set of *final states*. We often represent a transition $(q_1, \sigma, q_2) \in \rightarrow$ more compactly as $q_1 \xrightarrow{\sigma} q_2$. If there is a sequence of transitions $q_1 \xrightarrow{\sigma_1} q_2$, $\ldots$, $q_{n-1} \xrightarrow{\sigma_n} q_n$, we say that there is a *path* $q_1 \overset{\sigma_1 \ldots \sigma_n}{\longrightarrow^*} q_n$ in $\mathcal{F}$. $\mathcal{F}$ accepts the language $L(\mathcal{F})$, defined as the set of all finite words w such that there exists a path $q_1 \overset{w}{\rightarrow^*} q_n$, where q_1 is an initial state and q_n is a final state.

FSAs are convenient for describing many security properties. The state space describes security-relevant state of the program — such as privileged, unprivileged, or risky — the input alphabet describes security-relevant operations, and the transitions describe how the operations change the security state of the program. For example, Figure 2.2 shows an FSA describing how a process gains and drops the root privilege on Unix.

Most security properties come from books on secure programming, Unix manual pages, and folklore in the OS and security community. Many properties are simple, and the user can easily describe them in an FSA. Chapter 4 will present an approach for automatically generating FSAs for certain complex but structured security properties,

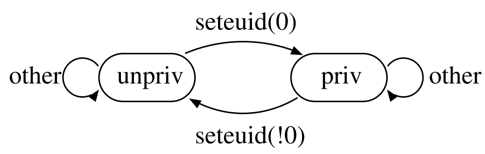


Figure 2.2: An FSA describing how a process gains and drops the root privilege by calling the `seteuid` system call

2.2.2 Creating a PDA from a program

MOPS creates a pushdown automaton from a source program in two steps. First, MOPS parses the program into a control flow graph (CFG). Currently MOPS uses the front end of RC [28] as the parser. A CFG is a directed graph where each node represents a program point and each edge represents an expression in the program[2]. Note that no edge links a caller to its callee, implying that each function body is an isolated graph in a CFG.

Next, MOPS creates a PDA from the CFG. A *pushdown automaton* (PDA) is a seven-tuple $\mathcal{P} = (Q, \Sigma, \Gamma, \Delta, Q_0, \gamma_0, F)$ where the *state space* Q is a finite set, the input alphabet Σ is a finite set, the *stack alphabet* Γ is a finite set, $\Delta \in (Q \times \Sigma \times \Gamma) \times (Q \times \Gamma^*)$ is the set of *transitions*, $Q_0 \subseteq Q$ is the set of *initial states*, $F \subseteq Q$ is the set of *final states*, and $\gamma_0 \in \Gamma$ is the *initial stack symbol*. For brevity, we often write a transition $((q, \sigma, \gamma), (q', w)) \in \Delta$ as $\langle q, \gamma \rangle \overset{\sigma}{\hookrightarrow} \langle q', w \rangle$.

A *configuration* of $\mathcal{P}$ is a pair $\langle q, w \rangle$ where $q \in Q$ and $w \in \Gamma^*$. An *initial configuration* is a pair $\langle q_0, \gamma_0 \rangle$ where $q_0 \in Q_0$ is an initial state and γ_0 is the initial stack symbol. A *final configuration* is a pair $\langle q_f, \gamma \rangle$ where $q_f \in F$ is a final state. A configuration is a snapshot of the PDA, and evolves according to the transitions of the PDA. If the PDA has a transition $\langle q, \gamma \rangle \overset{\sigma}{\hookrightarrow} \langle q', \gamma' \rangle$, then for all words w, the configuration $\langle q, \gamma w \rangle$ may evolve into the configuration $\langle q', r' w \rangle$. We denote this configuration transition as $\langle q, rw \rangle \overset{\sigma}{\Rightarrow} \langle q', r' w \rangle$. If there is a sequence of transitions from

[2]This definition is complementary to the more popular definition of CFG where each node represents an expression and each edge represents a program point.

the configuration c_0 to c_n, $c_0 \overset{\sigma_1}{\Rightarrow} \ldots \overset{\sigma_n}{\Rightarrow} c_n$, we say that there is a path from c_0 to c_n, denoted as $c_0 \overset{\sigma_1 \ldots \sigma_n}{\Longrightarrow^*} c_n$, and that c_0 is a predecessor of c_n, and c_n is a successor of c_0. A configuration c is *reachable* if there exists a path from any initial configuration to c. The language $L(\mathcal{P})$ accepted by $\mathcal{P}$ is the set of all finite words w such that $c_0 \overset{w}{\Rightarrow^*} c_f$ where c_0 is an initial configuration and c_f is a final configuration.

From a CFG, MOPS builds a PDA $\mathcal{P}$ whose state space Q contains a single state, whose input alphabet Σ contains the CFG edges (i.e., all the expressions in the program), whose stack alphabet Γ contains all the nodes in the CFG (i.e., all the program points), and whose initial stack symbol is the entry program point (usually the first expression in the function `main`) MOPS adds transitions to $\mathcal{P}$ according to the following rules. For each edge $n_1 \overset{e}{\to} n_2$ in the CFG:

- If the expression e is neither a function call nor a function return, add the transition $\langle, n_1\rangle \overset{e}{\hookrightarrow} \langle, n_2\rangle$.
- If the expression e is a call to a function f, add the transition$\langle, n_1\rangle \overset{e}{\hookrightarrow} \langle, n_f n_2\rangle$ where n_f is the entry point of the function f.
- If the expression e is a function return, add the transition $\langle, n_1\rangle \overset{e}{\hookrightarrow} \langle, \epsilon\rangle$ where ϵ represents an empty word.

2.2.3 Computing the intersection of the FSA and PDA

Given an FSA $\mathcal{F} = (Q, \Sigma, \to, Q_0, F)$ representing a security property and a PDA $\mathcal{P}' = (\{s\}, \Sigma, \Gamma, \Delta, \{s\}, \gamma_0, \{s\})$ representing a program, MOPS computes their intersection $\mathcal{P}$. In other words, $L(\mathcal{P}) = L(\mathcal{F}) \cap L(\mathcal{P}')$. $\mathcal{P}$ is also a PDA and has the form $(Q, \Sigma, \Gamma, \Delta', Q_0, \gamma_0, F)$. MOPS adds transitions to $\mathcal{P}$ according to the following rules:

For each transition in the form $\langle, \gamma\rangle \stackrel{\sigma}{\hookrightarrow} \langle, \gamma'\rangle$ in $\mathcal{P}'$

For each state q in $\mathcal{F}$

If there exists a transition $q \stackrel{\sigma}{\hookrightarrow} q'$ in $\mathcal{F}$

Then add a transition $\langle q, \gamma\rangle \hookrightarrow \langle q', \gamma'\rangle$ to $\mathcal{P}$

For each transition in the form $\langle, \gamma\rangle \hookrightarrow \langle, \gamma'\gamma''\rangle$ in $\mathcal{P}'$

For each state q in $\mathcal{F}$

Add a transition $\langle q, \gamma\rangle \hookrightarrow \langle q, \gamma'\gamma''\rangle$ to $\mathcal{P}$

For each transition in the form $\langle, \gamma\rangle \hookrightarrow \langle, \epsilon\rangle$ in $\mathcal{P}'$

For each state q in $\mathcal{F}$

Add a transition $\langle q, \gamma\rangle \hookrightarrow \langle q, \epsilon\rangle$ to $\mathcal{P}$

2.2.4 Deciding if the PDA is empty

Recall that the language accepted by the PDA $\mathcal{P}$ is $L(\mathcal{P}) = \{w \mid \exists c_0 \in Q_0 \times \{\gamma_0\}, c_f \in F \times \Gamma^* \,.\, c_0 \stackrel{w}{\Rightarrow}^* c_f\}$ where c_0 is an initial configuration and c_f is a final configuration. Therefore, $L(\mathcal{P})$ is empty if no final configuration is reachable. To decide if $L(\mathcal{P})$ is empty, MOPS needs to find all the reachable configurations, or in other words, all the successors of the initial configurations. The following algorithm was originally described elsewhere [19].

Let *post*$^*(C)$ denote all the successors of the configurations in the set C, i.e., *post*$^*(C) = \{c | \exists c' \in C \text{ and } c' \Rightarrow^* c\}$. Since *post*$^*(C)$ may be infinite, we cannot represent it explicitly, but rather must represent it symbolically.

For PDA $\mathcal{P} = (P, \Sigma, \Gamma, \Delta, Q_0, F, \gamma_0)$, define a $\mathcal{P}$-automaton to be an FSA $(Q, \Gamma, \rightarrow$

$, F, \{f_0\})$ where $Q \supseteq P$, $\rightarrow \subseteq Q \times \Gamma \times Q$, and f_0 is the single final state. Note that the initial states of the $\mathcal{P}$-automaton are the final states of the PDA $\mathcal{P}$. We use a $\mathcal{P}$-automaton to represent a set of configurations of the PDA $\mathcal{P}$: each path $q \xrightarrow{w}{}^{*} f_0$ in the $\mathcal{P}$-automaton, where q is an initial state and f_0 is the final state, represents the configuration $\langle q, w \rangle$ in $\mathcal{P}$.

Let C_0 be the set of initial configurations. MOPS computes *post*$^*(C_0)$ in two steps. First, MOPS constructs a $\mathcal{P}$-automaton that represents C_0; then, MOPS saturates the $\mathcal{P}$-automaton with all the successors of C_0. The following shows the details.

Construct a $\mathcal{P}$-automaton Construct a $\mathcal{P}$-automaton $\mathcal{Q}$ that represents C_0. For each configuration $\langle q, \gamma_1 \ldots \gamma_n \rangle \in C_0$, MOPS creates $n-1$ intermediate states $q_1, \ldots, q_{n-1}$ in $\mathcal{Q}$, and then adds n transitions $q \xrightarrow{\gamma_1} q_1$, $q_1 \xrightarrow{\gamma_2} q_2, \ldots, q_{n-1} \xrightarrow{\gamma_n} f_0$ to $\mathcal{Q}$.

Saturate the $\mathcal{P}$-automaton with *post*$^*(C_0)$ MOPS saturate $\mathcal{Q}$ in two steps:

- For each pair (p', γ') such that $\mathcal{P}$ contains one or more transitions of the form $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \gamma'' \rangle$, add a new state $q_{p',\gamma'}$ to $\mathcal{Q}$.
- Add new transitions to $\mathcal{Q}$ according to the following saturation rules:
  - If $\langle p, \gamma \rangle \hookrightarrow \langle p', \epsilon \rangle \in \mathcal{P}$ and $p \xrightarrow{\gamma} q \in \mathcal{Q}$, add a transition $p' \xrightarrow{\epsilon} q$ to $\mathcal{Q}$.
  - If $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle \in \mathcal{P}$ and $p \xrightarrow{\gamma} q \in \mathcal{Q}$, add a transition $p' \xrightarrow{\gamma'} q$ to $\mathcal{Q}$.
  - If $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \gamma'' \rangle \in \mathcal{P}$ and $p \xrightarrow{\gamma} q \in \mathcal{Q}$, add transitions $p' \xrightarrow{\gamma'} q_{p',\gamma'}$ and $q_{p',\gamma'} \xrightarrow{\gamma''} q$ to $\mathcal{Q}$.

After saturation, $\mathcal{Q}$ represents all the reachable configurations of the PDA $\mathcal{P}$: each path $q_0 \xrightarrow{w}{}^{*} f$ in $\mathcal{Q}$ represents the reachable configuration $\langle q_0, w \rangle$ in $\mathcal{P}$. Recall that the language accepted by $\mathcal{P}$, $L(\mathcal{P})$, is empty if and only if no final configuration of $\mathcal{P}$ is reachable, and that a final configuration of $\mathcal{P}$ is reachable if and only if there exists a path from an initial state to the final state in the $\mathcal{P}$-automaton. Therefore, $L(\mathcal{P})$ is empty if and only if there does not exist any path from an initial state to the final state in the $\mathcal{P}$-automaton. We can solve this problem using a graph algorithm, such as depth-first search.

2.2.5 Backtracking error traces

If the PDA $\mathcal{P}$ accepts a non-empty language, it means that the program may violate the security property. Each string in the language represents an *error trace*, an execution path in the program that may violate the security property. Error traces would help the MOPS user not only determine whether the reported errors are real errors or false warnings, but also locate the sources of these errors. The saturated $\mathcal{P}$-automaton computed in Section 2.2.4, however, does not provide error traces; it just shows the stack of $\mathcal{P}$ when $\mathcal{P}$ enters error states. Each reachable final configuration $\langle q_0, \gamma_1 \ldots \gamma_n \rangle$ indicates that when $\mathcal{P}$ enters the error state q_0, its stack has the string $\gamma_1 \ldots \gamma_n$; interpreted in the program being checked, this final configuration indicates that when the program enters the error state q_0, its current program point is γ_1 and its stack has the return addresses $\gamma_2 \ldots \gamma_n$. To help the user verify and locate the error, MOPS needs to recover the entire error traces that start from the beginning of the program and end in γ_1, and whose stacks contain the return addresses $\gamma_2 \ldots \gamma_n$. This process is called *backtracking*.

Let us define *predecessor*: if the program point n_1 is immediately before the program point n_2 on any trace, n_1 is a predecessor of n_2. To backtrack error traces, we need to record all the predecessors of each program point. We can record predecessors easily during the saturation of the $\mathcal{P}$-automaton named $\mathcal{Q}$ (Section 2.2.4). Recall that each transition $p \xrightarrow{\gamma} q$ in $\mathcal{Q}$ represents the program point γ. We can easily verify that when a saturation rule instructs MOPS to add a new transition t_2 into $\mathcal{Q}$ based on a current transition t_1 in $\mathcal{Q}$, the program point in t_1 is a predecessor of the program point in t_2. Hence, MOPS associates a *predecessor list* with each transition in $\mathcal{Q}$, and fills the predecessor lists during the saturation of $\mathcal{Q}$. The following describes the new procedure for saturating $\mathcal{Q}$:

- For each pair (p', γ') such that $\mathcal{P}$ contains one or more rules of the form $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle$, add a new state $q_{p',\gamma'}$ to $\mathcal{Q}$.
- Add new transitions to $\mathcal{Q}$ according to the following saturation rules:
  - If $\langle p, \gamma \rangle \hookrightarrow \langle p', \epsilon \rangle \in \mathcal{P}$ and $t = p \xrightarrow{\gamma} q \in \mathcal{Q}$, add a transition $t' = p' \xrightarrow{\epsilon} q$ to $\mathcal{Q}$ and add t to the predecessor list of t'.
  - If $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle \in \mathcal{P}$ and $t = p \xrightarrow{\gamma} q \in \mathcal{P}$-automaton, add a transition $t' = p' \xrightarrow{\gamma'} q$ to $\mathcal{Q}$ and add t to the predecessor list of t'.
  - If $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle \in \mathcal{P}$ and $t = p \xrightarrow{\gamma} q \in \mathcal{P}$-automaton, add transitions $t' = p' \xrightarrow{\gamma'} q_{p',\gamma'}$ and $q_{p',\gamma'} \xrightarrow{\gamma''} q$ to $\mathcal{Q}$, and add t to the predecessor list of t'.

Note that after MOPS applies the above rules, only the transitions at the front of any paths may have predecessors or may themselves be predecessors of other transitions. We call the transitions at the front of any paths *front transitions*. This is expected, because only front transitions represent the program points on the program's execution paths (the non-front transitions represent the return addresses on the program's stack), and only these programs points have predecessors or can be the predecessors of other program points.

This new saturation procedure provides MOPS with sufficient information for backtracking; however, one problem remains. During backtracking, if a transition has a single predecessor, MOPS simply follows the predecessor; however, if a transition has multiple predecessors, MOPS has to decide which one to follow. MOPS could pick an arbitrary predecessor, say the first one, but this naive choice could trap the backtracking in an infinite loop if there is a circular predecessor relation. Alternatively, MOPS could choose a random predecessor. Although this approach may with high probability avoid being trapped in an infinite loop, it is undesirable. This is because after the user fixes the bug indicated by an error trace, he wants to know if MOPS would stop considering the trace as an error, but he cannot tell this if MOPS report a random trace each time. Besides the requirement that MOPS reports deterministic traces, the user would also like error traces to be as short as possible, because often short traces are easier to review. This latter requirement suggests that we might use shortest-path algorithms in graph theory to backtrack error traces.

Given a $\mathcal{P}$-automaton, define a graph $G_{\mathcal{P}} = (V, E)$ over the $\mathcal{P}$-automaton where the node set V consists of all the front transitions in the $\mathcal{P}$-automaton, and $v_1 \to v_2 \in E$ if the front transition v_1 is a predecessor of the front transition v_2. Set a weight on each edge (e.g., we could let each edge have a unit weight). To backtrack an error trace from a node n in G, MOPS computes the shortest path from a node in $G_{\mathcal{P}}$ that is an initial configuration in $\mathcal{P}$ to the node n using Dijkstra's algorithm, and the program points in all the nodes on the path constitute an error trace in the program.

There is, however, a problem with this approach. Dijkstra's algorithm applies to finite state graphs, where all the paths in the graphs are feasible. The graph $G_{\mathcal{P}}$, however, is a graph over the PDA $\mathcal{P}$, so some paths in $G_{\mathcal{P}}$ may be infeasible because they contain function calls and returns that are not properly nested, but Dijkstra's algorithm may output these paths because it is unaware of this constraint. We could replace Dijkstra's algorithm with a shortest-path algorithm for pushdown systems, but such algorithms are very expensive [45]. To keep its efficiency, MOPS uses a compromise: it still uses Dijkstra'a algorithm, but it checks whether every reported path is feasible. For each infeasible path, MOPS tries to find a feasible path related to the infeasible path by choosing alternative predecessors that result in properly nested function calls and returns during backtracking. Since this trick occasionally causes the backtracking to be trapped in an infinite loop, MOPS monitors backtracking and aborts it if MOPS detects a loop.

2.2.6 Consolidating error trace

The algorithm in the previous section allows MOPS to report error traces, but how many error traces should MOPS report? On one hand, reporting too few traces is undesirable, because users like to view many bug reports at once, just as they like to view many syntactic error reports from a compiler at once. Moreover, if an error trace does not indicate a real error (i.e., the trace is a false positive), users want MOPS to report other traces, which may indicate real errors. This requirement seems to suggest that MOPS should report all the error traces. Reporting all traces, however, is sometimes infeasible, because the program may contain infinitely many error traces.[3] Even if a program contains finite error traces, many of them are similar because they indicate the same programming error, so it is desirable to report only one of them. This would save the user's time considerably because the user has to review each trace manually.

Researchers have not agreed on what constitutes a unique programming error. We choose the following definition. When a program trace goes from a program point p_1 with a state s_1 to a program point p_2 with an error state s_2, the tuple (s_1, p_1, s_2, p_2) represents the unique programming

[3] This may happen, for example, when an error trace goes through a loop. Because MOPS does not do data flow analysis, it does not know how many times the loop may iterate, so it assumes that the loop may iterate an arbitrary number of times. Different number of iterations creates different error trace.

error that occurs on the last transition in the trace before the trace enters the error state. In other words, if multiple error traces all go from the program point p_1 with the state s_1 to the program point p_2 with the state s_2, MOPS consider them to represent the same programming error. We choose this definition because it is easy to implement in the model checker, and it works fairly well in practice. We need future research to evaluate this definition of unique programming errors.

After MOPS builds the $\mathcal{P}$-automaton and saturates it, it repeats the following steps on the graph G over the $\mathcal{P}$-automaton (Section 2.2.5) to find error traces that represent unique programming errors:

1. Find one of the shortest error traces using the algorithm in Section 2.2.5.

2. Find the first location on the trace where the program enters a error state. Recall that a trace consists of a sequence of nodes in G, so MOPS finds the first edge $v_1 \rightarrow v_2$ on the trace where v_1 has a good state but v_2 has a error state.

3. Remove the edge $v_1 \rightarrow v_2$ from the graph $G_{\mathcal{P}}$.

We can easily verify that this algorithm finds a unique error trace each time. This is because each edge $v_1 \rightarrow v_2$ represents a unique tuple (s_1, p_1, s_2, p_2) where s_1 is the starting state and p_1 is the program point in v_1, and s_2 is the starting state and p_2 is the program point in v_2 (recall that each node in $G_{\mathcal{P}}$ is a transition in the $\mathcal{P}$-automaton). By removing the edge $v_1 \rightarrow v_2$ from $G_{\mathcal{P}}$, the algorithm will not find another trace that has the same tuple (s_1, p_1, s_2, p_2).

Although this approach works very well in practice by reducing substantially the number of error traces that users have to review manually, it has two shortcomings. The first problem is that a false error trace may mask other real error traces. Of all the error traces from the same programming error, some may be false positives but others may be real. This may happen, for example, because MOPS does no data flow analysis and assumes that a trace can follow each branch of a conditional statement regardless of the history on this trace. This approximation sometimes leads infeasible traces. Since MOPS reports a single trace per unique programming error, when the trace is infeasible, the user should examine whether there exist feasible error traces that are similar to this trace (i.e., they indicate the same programming error).

The second problem is that a false error discovered on a trace will mask later real errors on the trace. Since the above algorithm only looks for the first location on a trace where the trace enters an error state, it will not discover later errors on the trace. This is not a problem when the first error is real, because the user can correct the first error and run MOPS to find later errors on the trace. This is a problem, however, when the first error is a false warning, because the user will disregard the trace entirely and therefore overlook potential later errors on the trace.

2.2.7 Soundness

A major goal in the design of MOPS is *soundness*. A sound tool will not overlook any bugs in a program that may violate a security property. In many security applications, we need sound tools, which can not only find bugs but also verify the absence of bugs.

MOPS strives to be sound. We can divide the MOPS process into three steps: (1) describing a security property using an FSA; (2) representing all the traces in a program using a PDA; (3) computing the intersection of the FSA and the PDA and decide whether it is empty. The last step is sound: it will not overlook any trace represented in the PDA that may violate the property represented by the FSA. The first two steps, however, may introduce unsoundness. Let us first look at possible sources of unsoundness in the second step:

1. MOPS assumes that the program is single-threaded. Therefore, the PDA does not describe concurrent execution of multiple threads.

2. Since MOPS constructs the PDA by parsing the source program, MOPS cannot see any code that is not in the source program. This requires that the source program be memory safe. For example, the program should not have buffer overrun vulnerabilities that can be exploited for injecting foreign code. Furthermore, MOPS cannot see any code in the program that is not seen by its parser. Some programs have directives that instruct the parser to exclude certain code from compilation based on certain conditions, such as the OS type, version, or capabilities. MOPS cannot see some code in the executable program if the conditions at the compilation time of the executable program is different from the conditions at the running time of MOPS's parser. Last, MOPS interprets the program as portable C code, so MOPS does not understand implementation dependent features in the program.

3. Since MOPS models each source program as a PDA, it cannot handle non-local jumps: e.g., `longjmp()` in C programs.

4. The current implementation of MOPS does not handle signal handlers and function pointers.

Among these four sources of unsoundness, the first three are fundamental limitations of the MOPS framework, while the last is a limitation in the current implementation.

The first step, describing a security property using an FSA, may also introduce unsoundness. Some properties are sound only under certain assumptions about the program. For example, to enable the user to express the property *do not call `open()` after calling `stat()` on the same file name*, MOPS allows the user to declare a generic pattern variable `f` to create the FSA shown in

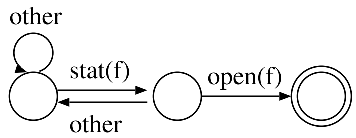


Figure 2.3: An FSA describing the property: *A program should not call* `open` *after calling stat on the same file name.*

Figure 2.3. In that FSA, the variable `f` in `stat(f)` and `open(f)` is a generic pattern variable — it refers to any variable that is syntactically used in both `stat()` and `open()`. Since this is syntactic matching, it does not survive aliasing: if the program contains "`stat(x); y=x; open(y)`", then MOPS does not know that `y` is an alias of `x`, so the property is not sound for this program.

2.2.8 Complexity

Since the MOPS process consists of several stages, we analyze the complexity of each stage separately. In the analysis, let N be the number of nodes in the CFG, E be the number of edges in the CFG, and S be the number of states in the FSA.

1. Creating the PDA from the CFG: MOPS adds one transition to the PDA for each edge in the CFG, so the running time is $O(E)$.

2. Intersecting the FSA with the PDA to create a new PDA: Since MOPS adds one transition to the new PDA for each pair of a transition in the PDA and a state in the FSA, the running time is $O(E \times S)$.

3. Deciding if the PDA is empty:

   (a) Creating the $\mathcal{P}$-automaton $\mathcal{Q}$: Let S_0 be the number of initial states in the FSA. Since the PDA has S_0 initial states and a single initial stack symbol, it has S_0 initial configurations.

Since these initial configurations are of the form $\langle q, \gamma \rangle$, and for each such configuration MOPS adds one transition to the $\mathcal{P}$-automaton, the running time is $O(S_0)$.

(b) Saturating $\mathcal{Q}$. Let F be the number of functions defined in the program. MOPS saturates $\mathcal{Q}$ in two stages. In the first stage, since MOPS creates S intermediate states for each function and there are F functions, the running time is $O(S \times F)$. In the second stage, MOPS adds transitions. The following analysis shows that the running time of this stage is $O(S^2 \times E)$. We calculate the running time by counting the maximum number of times when the algorithm may add new transitions to $\mathcal{Q}$. Because the algorithm adds a new transition only when one of the transition's predecessors is already in $\mathcal{Q}$, we first count how many successors each transition may have. Let $t = s_1 \xrightarrow{p} s_2$ be a transition in $\mathcal{Q}$. Let E_p be the number of outgoing edges from the program point p in the CFG. Since transitions in the FSA are deterministic, t has at most E_p successors. Next, we count the maximum number of transitions in $\mathcal{Q}$ whose program points are p (i.e., maximum number of different choices of t given p).[4] The algorithm requires that each predecessor must be a front transition (i.e., its starting state must be an initial state of $\mathcal{Q}$), so there are S choices of s_1 in t. Given p, there are at most S choices of s_2. This is because if p is in the function `main`, s_2 must be the final state of $\mathcal{Q}$; if p is in a function `f` other than `main`, s_2 must be one of the S intermediate states that the algorithm has created for the function `f` earlier. Therefore, there are at most S^2 transitions whose program points are p. Since each such transition may have at most E_p successors according to our analysis earlier, all the transitions whose program points are p have at most $S^2 \times E_p$

[4]The number of times when the algorithm adds transitions to $\mathcal{Q}$ is generally larger than the number of transitions in $\mathcal{Q}$, because when two different predecessors share the same successor, the algorithm keeps only one copy of the successor but adds both the predecessors to the predecessor list of the successor.

successors. Since there are N choices of p, the maximum total number of successors that the algorithm may add to $\mathcal{Q}$ is $\sum_{p=1}^{N} S^2 \times E_p = S^2 \sum_{p=1}^{N} E_p = S^2 \times E$, or $O(S^2 \times E)$.

Since the running time in the second stage ($O(S^2 \times E)$) dominates that in the first stage ($O(S \times F)$), the combined running time is $O(S^2 \times E)$.

4. Backtracking error traces. MOPS backtracks error traces by running Dijkstra's algorithm in the graph $\mathcal{G}_{\mathcal{P}} = (V', E')$, whose nodes are the front transitions in $\mathcal{Q}$ and whose edges link the transitions with their predecessors. By the above analysis, $|V'| = O(S^2 \times N)$ and $|E'| = O(S^2 \times E)$. MOPS uses a binary heap to implement the priority queue in Dijkstra's algorithm, so the algorithm's running time is $O(E' \lg V')$ on the graph $G = (V', E')$ [13]. Therefore, the running time of the backtracking error traces is $O(S^2 \times \lg S \times E \times \lg N)$.

To summarize, the running time of the MOPS process without backtracking is $O(S^2 \times E)$, and the running time of backtracking is $O(S^2 \times \lg S \times E \times \lg N)$.

2.2.9 Compaction

The current implementation of MOPS uses a technique called *compaction* to accelerate MOPS. The motivation for compaction is the observation that, for most programs and most properties, the majority of the statements in the program are irrelevant to the property. We describe the observation formally as follows. Define a *relevant function* with regard to a security property as a function defined in a program that contains at least one *relevant statement*. Define a *relevant statement* as a statement that may trigger a state change in the FSA by itself or by calling a relevant function directly or indirectly. 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 if 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 running the model checker on a program should always produce the same results as running on the compacted version of the program. Formally, it requires that for each path in the uncompacted CFG there must exist an equivalent path in the compacted CFG, and vice versa. Figure 2.4 describes an algorithm that satisfies this requirement. The time complexity of this algorithm is $O(MN)$, where M is the maximum number of relevant statements in any function 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 is a legacy technique in MOPS. Although we ran all the experiments in this dissertation using compaction, we believe that a more efficient implementation of MOPS would render it unnecessary, and we are working on that.

2.3 Formal models

MOPS checks temporal safety properties. Among the many alternatives for specifying these properties, we choose the finite state automaton (FSAs) representation. We believe that since FSA representation is straightforward and familiar, it is easy to write and to understand. For examples of properties that we have checked, see Chapter 5.

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 $v \to v'$ where $b_v = b_{v'}$ and merge v with v'.
7. Remove the bodies of all the irrelevant functions.

Figure 2.4: An algorithm for compacting CFGs

An FSA contains states and transitions. A state describes the current status of the program with regard to the security properties. Two types of states are special: initial states and final states. A program starts in an initial state of the FSA and then evolves through other states of the FSA when it executes. When the program enters a final state of the FSA, this indicates that the program has violated the security properties. As such, we also refer to final states of the FSA as *error states*. Each transition of the FSA describes the execution of an expression in the program that triggers a state change. The MOPS user represents the expression by its abstract syntax tree (AST).

The MOPS user specifies an FSA in a text file. In the specification, each state except the final state has multiple outgoing transitions, and at least one transition stays in the same state, called the *self-loop transition*, and one transition goes to another state. The multiple transitions from the same state need not be mutually exclusive: the order in which they are specified in the FSA determines their precedence except the self-loop transition, which takes the last precedence. This means that the FSA is deterministic: each expression in the program triggers a unique transition from each state in the FSA during model checking.

The FSA in Figure 2.5 describes a process gaining and dropping the root privilege by making the `seteuid` system call with two different parameters, and the FSA in Figure 2.6 describes the property "do not execute an untrusted program"[5]. Each line describes a state, initial state, final state, AST, or transition. A line describing a state starts with the keyword `state` followed by the name of the state (to be used by the user for describing transitions) and a label for the state (to be used by MOPS for annotating error traces). A line describing an initial state or a final state is similar, except that it substitutes the keyword `initialstate` or `finalstate`. A line describing an AST consists of the keyword `ast`, the name of the AST, and a serialized form of the AST. A line describing a transitions consists of the keyword `transition`, the names of the starting state and ending state of the transition, and the name of the AST that triggers the transition.

2.3.1 Modularization

Some security properties are complex, and so are their FSA descriptions. One way to tackle complexity is through divide and conquer. MOPS allows the user to decompose a complex security property into several smaller modules and specify each module individually, and MOPS can automatically compose these modules into the complex property. We call this feature *modularization*. Formally, each module is an FSA, whose parallel composition is the complex FSA.

For example, consider the property that a process should not execute untrusted programs when it has the root privilege. The user can decompose the property into two modules: the first one

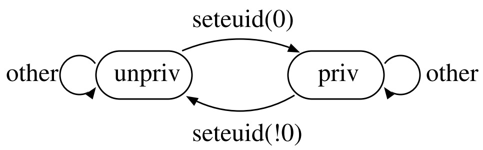
[5] This FSA over-approximates the property by catching all calls to `execl`, regardless of whether the program in its arguments is trusted. The FSA also ignores other system calls for executing new programs, such as `execl`, `system`, etc.

state priv “With root privilege”;
state unpriv “Without root privilege”;

ast seteuid(0) { function_call { identifier seteuid } { lexical_cst 0 } };
ast seteuid(!0) { function_call { identifier seteuid } { function_call { identifier getuid } } };
ast other { other };

transition priv unpriv seteuid(!0);
transition priv priv other;
transition unpriv priv seteuid(0);
transition unpriv unpriv other;

(a) Textual description



(b) Graphical representation

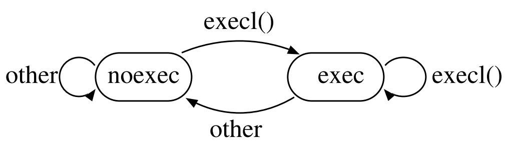
Figure 2.5: An FSA describing a process gaining and dropping the root privilege by making the `seteuid` system call with two different parameters

```
state noexec "Have not called execl()";
state exec "Have just called execl()";

ast execl { function_call { identifier execl } { ellipsis } } ;
ast other { other };

transition noexec exec execl;
transition noexec noexec other;
transition exec exec execl;
transition exec noexec other;
```

(a) Textual description



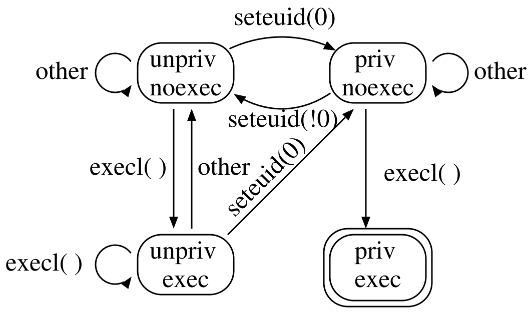
(b) Graphical representation

Figure 2.6: An FSA describing whether a program has executed another untrusted program

tracks when the process gains or drops the root privilege, and the second one tracks when the process executes untrusted programs. MOPS automatically composes these two modules into the desired property. Figure 2.5 shows the first module, and Figure 2.6 shows the second module. Figure 2.7 shows the FSA that describes the parallel composition of the two modules. The first line starts with the keyword `label` followed by a label for this FSA, which MOPS will put in the error traces. The second line starts with the keyword `fsa` followed by the file names of all the FSAs that are modules of this complete property. After that, the file contains lines each of which describing an initial state or final state of the FSA. Since the state space of the complete FSA is the Cartesian product of the state spaces of all the module FSAs, an initial or final state of the complete FSA is a tuple of states each of which is a state in one of the module FSAs. A line describing an initial state starts with the keyword `initialstate`, continues with one state from each module FSA, and ends with a label

label “Check for unsafe system calls made with root privilege”;
fsa setuid.fsa unsafe.fsa;
initialstate unpriv noexec “Initial state”;
finalstate priv exec “Make unsafe syscalls with root privilege”;

(a) Textual description



(b) Graphical representation

Figure 2.7: An FSA describing the property “*a process should not execute untrusted programs when it has the root privilege*”.

of this initial state. A line describing a final state has a similar format except using the keyword `finalstate` instead.

Modularization makes it easier for the user to write properties. Since each module isolates one aspect of the property, modularization facilitates the reuse of property components. For example, the user can reuse the module describing the raise and drop of the root privilege in the above property in other properties that deal with privilege.

Modularization, however, is not a panacea. It applies only to those properties that can be decomposed into independent modules. This is because each module must be self-contained. For example, no transition in a module should be contingent upon the current state of any other module.

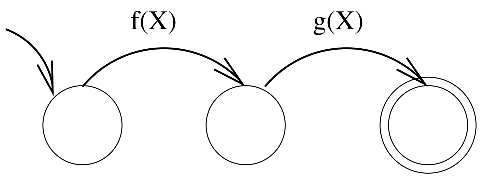


Figure 2.8: An FSA illustrating the use of pattern variables.

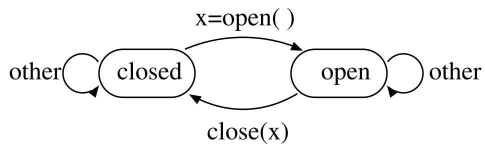
2.3.2 Pattern variables

To extend the expressiveness of ASTs, we introduced *pattern variables*, which can describe repeated occurrences of the same syntactic expression. For instance, if X denotes an pattern variable, the pattern $f(X, X)$ describes any calls to the function f with two syntactically identical arguments. In any error trace accepted by an FSA, the pattern variable X has a single, consistent instantiation throughout the trace.

Formally, let Σ denote the set of ASTs. We may view a program trace as a string in Σ^*, and a property B as a regular language on Σ^*. Pattern variables provides existential quantification over the set of ASTs. For instance, the pattern $\exists X.f(X, X)$ matches any call to f whose two arguments, once parsed, yield the same syntax subtree. If $B(X)$ is a language with an unbound pattern variable X, the language $\exists X.B(X)$ accepts any trace $t \in \Sigma^*$ where there exists an AST A' so that $B(A')$ accepts t. In other words, if $L(B)$ denotes the set of error traces accepted by the language B, we define $L(\exists X.B(X)) = \cup_{A'} L(B(A'))$. We use the convention that unbound pattern variables are implicitly existentially quantified at the outermost scope.

For instance, the FSA shown in Figure 2.8 accepts any trace containing a call to f followed by a call to g with the same syntactic argument. This property would accept traces such as $\langle f(0), g(0)\rangle$ or $\langle f(0), f(1), h(3), g(1)\rangle$ as error traces, but not $\langle f(0), g(1)\rangle$.

Figure 2.9 shows an example of how to use pattern variables to describe properties. x is



(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 2.9: An example showing the use of pattern variables

a pattern variable in the FSA in Figure 2.9(a), and x can be bound to either the expression a or b in the program in Figure 2.9(b).

Chapter 3

Implementation

This chapter first describes how we implemented the framework and algorithms in Chapter 2 in the tool MOPS. Then, it describes our two engineering efforts to make MOPS usable: we integrated MOPS into the build processes of software packages to make it very easy to run MOPS, and we designed an HTML-based user interface for presenting error reports to help the user understand potential errors in the programs.

3.1 Implementation

MOPS consists of a parser and a model checker. The parser takes a C source program and outputs its control flow graph (CFG). The model checker takes the CFG and a finite state automaton (FSA) describing a security property, and decide if any paths in the CFG may violate the property. If so, MOPS reports these paths. Figure 3.1 shows a brief architectural overview of MOPS.

We implemented MOPS in Java. The current version of MOPS contains 11K lines of code.

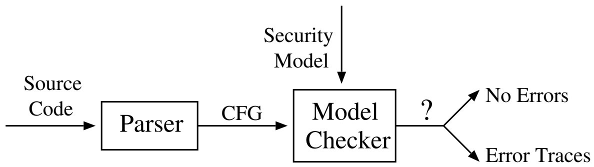


Figure 3.1: Architecture of MOPS

3.2 Build process integration

This chapter describes our engineering efforts to help users run MOPS easily. A common problem with many code auditing tools is that they are difficult to use: often they require the user to modify the build scripts of the software package and to run several different tools manually. As a result, these tools are not only difficult to learn but also tedious to use. To achieve our goal of making MOPS a useful tool for ordinary programmers, we must make MOPS easy to use.

The core MOPS process contains three steps:

1. The MOPS parser parsers each C source file into a control flow graph (CFG).

2. The MOPS linker links all the CFGs that belong to same program together.

3. The MOPS model checker checks the linked CFG against a security property.

We will describe a series of refinements that we tried to make running MOPS more easily. First, it helps to give an overview of how the user produces executables from source programs. Most source programs are distributed in packages. A packages contains source programs and instructions on how to produce executables from the source programs. These instructions are called *build scripts*. During the *build process*, the user runs certain tools, such as `make` or `rpmbuild`, that execute the build scripts to produce executables.

3.2.1 Running MOPS by hand

Before we developed additional support for running MOPS, we had three tools, the MOPS parser, linker, and model checker, each of which handles one step in the MOPS process. In principle, these tools are all we need to run MOPS: the user could invoke each of them manually. In practice, however, invoking these tools manually is very tedious and error prone, because:

- If an executable program is produced from multiple source programs, the MOPS user has to find all the source programs to run the MOPS parser on each of them separately. Finding all the related source programs can be a challenge, as the user has to parse the often complex build scripts manually.

- The user should pass to the MOPS parser all the command line options that are passed to the real parser during the package build. Again, finding these command line options requires the user to parse the build scrips in the package, which is a non-trivial, error-prone task.

3.2.2 Modifying build script

To overcome the problems with running MOPS by hand, we tried to integrate MOPS into the build process of source packages, since the build process has information on which source programs produce each executable and what command line options are passed to the parser for each source program.

Naturally, we first tried to insert the MOPS process into the build scripts. Most build scripts are makefiles, so we modified them to let them run the MOPS parser, linker, and model checker during build. This method worked fine on a few packages, but had problems with many packages. First, many packages have complicated build scripts that are difficult to understand.

Second, some packages have many build scripts, and a few packages have one build script in each sub-directory. Understanding and inserting the MOPS process into each of them would be very tedious. Last, many packages automatically generate build scripts during the configuration stage of the build process. Therefore, the next time when the user builds the package, the build process will overwrite the user's modification to the build scripts.

3.2.3 Interposing on GCC

To overcome the problems with manually modifying build scripts, we would like a method to insert the MOPS process into the build process without modifying the build scripts. We found a solution for the build processes that use GCC to produce executables; most packages on Unix systems have such build processes. GCC is the compiler front end, which invokes the back end compiler, assembler, and linker for produce assembly code, object code, and executable code respectively. GCC allows the user to substitute alternative back end tools by setting the environment variable `GCC_EXEC_PREFIX`. Using this method, we provide GCC with special back end tools that invoke the MOPS tools. Since different build processes vary considerably, we went through several iterations to make this approach work on most source packages.

MOPS backend parser, assembler, and linker

The steps in the MOPS process parallels the steps in the producing an executables from its source programs. We can view the MOPS process as producing executables on a different platform. Build an executable involves three steps: (1) compiling each source program into assembly code using a parser; (2) transforming assembly code into machine code using an assembler; (3) linking the machine code from multiple files, including libraries, into an executable. Mirroring GCC's parser, assembler, and linker, we developed MOPS backend parser, backend assembler, and backend linker. The MOPS back end parser invokes the MOPS parser, which parses a source program into a CFG. The MOPS back end assembler simply copies its input to its output, as no program transformation is necessary. The MOPS back end linker first invokes the MOPS linker to link all the CFGs together, and then invokes the MOPS model checker on the linked CFG. We had to solve several implementation issues to make the back end tools with with GCC:

- Command line options. Since these back end tools are invoked only by GCC, they have to be able to parse the command line options passed to them by GCC. However, we were unable to find comprehensive documentation of all such options, so we had to add more recognized options to these tools as our experiments continued.

- Integrated preprocessor. In GCC 2, a source program goes through the preprocessor before it is fed to the parser. In GCC 3, however, the parser contains an integrated preprocessor and receives unpreprocessed source programs. Since the MOPS parser contains no integrated preprocessor, the MOPS back end parser has to determine if its input file has been preprocessed. If not, the MOPS back end parser invokes the GCC preprocessor on the input file before invoking the MOPS parser.

Producing both CFGs and machine code

The above approach, however, causes the build of some packages to fail. The build processes of these packages need to run some executables produced earlier during the build[1], but MOPS's back end tools do not produce executables. We extended the MOPS back end tools to

[1] Packages need to run executables for several reasons: `autoconf` may run executables to test the configuration of a platform; the build process may run an executable to produce header files needed later in the build

let them produce executables. The new MOPS back end parser first invokes the MOPS parser to produce a CFG, and then invokes the GCC parser to produce assembly code. The CFG and the assembly code files are named relatedly (e.g. they are named `foo.cfg` and `foo.s` respectively) so that the MOPS tools can locate them together. The new MOPS back end assembler invokes the GCC assembler on the input assembly code to produce machine code in an object file. The new MOPS linker first links all the CFG together, then invoke the model checker on the linked CFG, and finally invokes the GCC linker on the machine code in the object files to produce an executable.

In this approach, we had to handle the implementation issue of piping through standard input and output. In some circumstances, such as when invoked with the `-pipe` option, GCC sends input to receive output from back end tools via the standard input and output rather than through files. Since some of the MOPS back end tools invoke two tools, each of which read the input. For example, the MOPS back end parser invokes both the MOPS parser and the GCC parser. Both these parsers need to read the input, but the standard input cannot be read more than once. To solve this problem, the MOPS back end parser first saves the data from the standard input into a file, then invokes both the MOPS parser and the GCC parser with the standard input file descriptor redirected to the saved file.

Encapsulating CFGs into machine code file

The above approach, however, still fails on some packages. The approach relies on the weak linkage between a file containing assembly code or machine code and its corresponding file containing a CFG – the linkage is their names. For example, the approach assumes that the CFG file `foo.cfg` corresponds to the assembly code file `foo.s` and the machine code file `foo.o`. However, when the build process moves files that contain assembly code or machine code, renames these files, inserts them into archive files, or extract them from archive files, it breaks this weak linkage. We need strong linkage between the files that GCC produces, such as assembly and machine code files, and their corresponding files produced by MOPS, namely CFG files. Our solution is to attach CFGs to machine code by taking advantage of the versatile ELF format, in which machine code is stored. ELF format allows multiple sections. The improved MOPS back end assembler, after invoking the GCC assembler to produce an object file containing machine code, inserts the CFG into a new MOPS section of the object file. When the MOPS back end linker is invoked on the object file, the linker extracts the MOPS section to obtain the CFG. Note that this approach works even when object files are inserted into or extracted from archive files, since an archive file is just a collection of object files, and inserting into or extracting from an archive file does not modify the content of the object files. Unfortunately, we cannot use the same approach for attaching CFGs to assembly code files, since these files are not in the ELF format. Here our goal is to insert into assembly files information that helps locate their corresponding CFGs, but at the same time to keep the file a legitimate assembly file (i.e., the GCC assembler can parse the file successfully). Our solution takes advantage of the comment syntax of x86 assembly files, where a line beginning with `#` is treated as comment. The improved MOPS back end parser, after producing a CFG file and an assembly file, inserts the name of the CFG file as a comment at the beginning of the assembly file. When the MOPS assembler back end is invoked on the assembly file, it locates the corresponding CFG file from the inserted CFG file name.

Additional tool support

The above approach solved most problems that we encountered during build. We developed a front end tool that uses this approach to help users run MOPS. The user can run the tool in a few ways. He can ask for a new shell where all the environment variables necessary for interposing on GCC to run MOPS are set up. Then he can build the package in the normal way as required by the package, such as running `configure; make`; the build process will then invoke the MOPS tools automatically in the background. Alternatively, he can give the front end tool a source RPM package. Then the tool will extract the package, build the package, and run MOPS on the package.

3.3 User interface for error reporting

Errors—i.e., violations of the security property—are reported to the user in HTML format. The user is shown a brief explanation of what security rule was violated, accompanied by a set of error traces. Each error trace shows one path of execution through the program on which the property is violated, and thus explains why MOPS believes the program to be in error.

Our user interface (UI) displays error traces to the user concisely by highlighting FSA transitions and using an extended stack backtrace paradigm. Figure 3.3 shows the user interface for displaying error traces that MOPS found in checking race conditions in *zip*. Each statement in the trace where the property FSA changes states is likely to be particularly relevant to explaining the source of this error, so we always show each such transition. A full listing of all statements executed along this trace is usually far too long for a human to digest, so we summarize this information by showing a skeleton of the call stack. Any statement that does not cause an FSA transition is elided; any function whose body is entirely elided is itself elided; and this continues recursively. This leaves us with a tree in which each leaf is an FSA transition, and the path from the root to that leaf shows the stack backtrace at the time that FSA transition occurred.

An example may help clarify. Suppose we have a program where `main()` calls functions

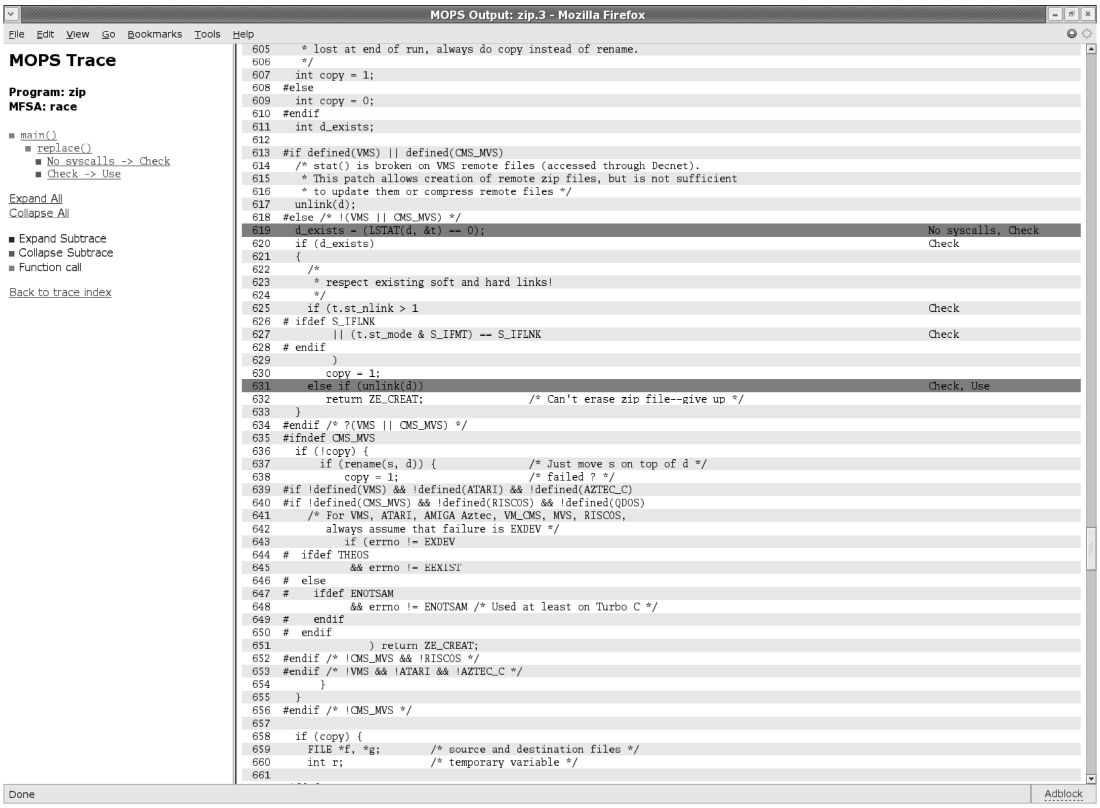


Figure 3.2: A user interface for displaying error traces

`f()`, `g()`, and `h()`, in that order; `f()` in turn calls `f0()` and then `f1()`; `g()` calls `g0()` and `g1()`; and similarly for `h()`. If the FSA transitions directly from the start state to the error state (in a single step) while executing the body of `g1()`, the error trace summary would look like this:

- `main()`
  - `g()`
    - `g1()`
      - Start → Error

Note that the calls to, and bodies of, `f()`, `f0()`, `f1()`, and `g0()` have been elided, because they had no effect on the state of the FSA. Alternatively, suppose that the FSA had changed state twice along this trace, once in `g1()` and then again later in `h0()`. In that case, a trace summary might look like this:

- `main()`
  - `g()`
    - `g1()`
      - Start → InProgress
  - `h()`
    - `h0()`
      - InProgress → Error

Trace summaries are interactive: clicking on one of the squares causes the body of that function to be expanded. This allows the user to see progressively more detailed error traces.

Our UI shows error traces in two panes, with a narrow left pane showing the trace summary and a right pane used to show the program source code. Clicking on any function call takes the right pane to the relevant call site in the source, and highlights that line of code. Clicking on an FSA transition goes to the statement that triggered the FSA to change states. Our experience is that if there is a bug in the program, it is most likely to be found where the last transition (the one that first enters the FSA's error state) occurs in some error trace. Therefore, by default our UI initially takes the user to the line of code matching the last FSA transition in this error trace.

Chapter 4

A Case Study of Security Models: Privileges in Unix Processes

In this chapter, we show a case study of how to create precise models for describing privileges in Unix processes.

4.1 Background

Access control in Unix systems is mainly based on user IDs. Each process has a set of user IDs and group IDs, which determine which system resources, such as files and network ports, the process can access[1]. Certain privileged user IDs and groups IDs allow a process to access restricted system resources. Particularly, the user ID zero represents the superuser *root*, and a process carrying the user ID zero can access all the system resources.

In some applications, a user process needs extra privileges, such as the permission to read

[1]In many Unix systems, a process has also a set of *supplementary group IDs*, which are not closely related to this topic and which will not be discussed.

the password file. By the principle of least privilege, the process should drop its privileges as soon as possible to minimize its risk to the system when it is compromised and execute malicious code. Unix systems offer a set of system calls, called the *uid-setting system calls*, for a process to raise and drop privileges. Unfortunately, for historical reasons, these calls are poorly designed, insufficiently documented, and widely misunderstood. “Many years after the inception of setuid programs, how to write them is still not well understood by the majority of people who write them” [57]. In short, the Unix setuid model is mysterious, and the resulting confusion has caused many security vulnerabilities.

To create models for describing privileges in Unix processes, we must resolve the setuid mystery. We proceed as follows. First, we study the semantics of the uid-setting system calls by reading kernel sources. While we compare and contrast the semantics among different Unix systems, we found that manual inspection is tedious and error-prone. This motivates us to construct a formal model to capture the behavior of the operating system and use it to guide our analysis. We will describe a new technique for building this formal model automatically. We have used the resulting formal model to understand more accurately the semantics of the uid-setting system calls, to uncover pitfalls in the Unix API of these calls, to identify differences in the semantics of these calls among various Unix systems, to detect inconsistency in the handling of user IDs within an OS kernel, and to check for the proper usage of user IDs in programs automatically.

4.2 The user ID model

Each user in a Unix system has a unique user ID. The user ID determines which system resources the user can access. In particular, the user ID zero represents the superuser *root*, who can access all the resources.

Each process has three user IDs: the *real user ID* (*real uid*, or *ruid*), the *effective user ID* (*effective uid*, or *euid*), and the *saved user ID* (*saved uid*, or *suid*). The real uid identifies the owner of the process, the effective uid is used in most access control decisions, and the saved uid stores a previous user ID so that it can be restored later. Similarly, a process has three group IDs: the *real group ID*, the *effective group ID*, and the *saved group ID*. In most cases, the properties of the group IDs parallel the properties of their user ID counterparts. For simplicity, we will focus on the user IDs and will mention the group IDs only when there is potential confusion. In Linux, each process has also an *fsuid* and an *fsgid* which are used for access control to the filesystem. The *fsuid* usually follows the value in the effective uid unless explicitly set by the *setfsuid* system call. Similarly, the *fsgid* usually follows the value in the effective gid unless explicitly set by the *setfsgid* system call. Since the *fsuid* and *fsgid* are Linux specific, we will not discuss them except when we point out an inconsistency in how the Linux kernel handles them.

When a process is created by *fork*, it inherits the three user IDs from its parent process. When a process executes a new file by invoking the *execv* family of system calls, it keeps its three user IDs unless the set-user-ID bit of the new file is set, in which case the system sets the effective uid and saved uid to the user ID of the file owner.

Since access control is based on the effective user ID, a process gains privilege by assigning a privileged user ID to its effective uid, and drops privilege by removing the privileged user ID from its effective uid. Privilege may be dropped either temporarily or permanently.

- To drop privilege temporarily, a process removes the privileged user ID from its effective uid but stores it in its saved uid. Later, the process may restore privilege by restoring the privileged user ID in its effective uid.

- To drop privilege permanently, a process removes the privileged user ID from all the three user IDs. Thereafter, the process can never restore privilege.

4.3 History

Bell Laboratories filed a patent application on Dennis Ritchie's invention of a bit to specify that a program should execute with the permissions of its owner, rather than invoker, in 1973. The patent was granted in 1979 [46]. Thus, setuid programs and related system calls have existed through most of Unix history.

4.3.1 Early Unix

In early Unix systems, a process had two user IDs: the real uid and the effective uid. Only one system call, *setuid*, modified them according to the following rule: if the effective uid was zero, *setuid* set both the real uid and effective uid; otherwise, *setuid* could only set the effective uid to the real uid [57]. This model had the problem that a process could not temporarily drop the root privilege in its effective uid and restore it later. As Unix diverged into System V and BSD, each system solved the problem in a different way.

4.3.2 System V

System V added a new user ID called the saved uid to each process. Also added was a new system call, *seteuid*, whose rules were:

- If the effective uid was zero, *seteuid* allowed the user to set the effective uid to any user ID.

- Otherwise, *seteuid* allowed the user to set the effective uid to only the real uid or saved uid.

seteuid did not change the real uid or saved uid. Furthermore, System V modified *setuid* so that if the effective uid was not zero, *setuid* functioned as *seteuid* (changing only the effective uid); otherwise, *setuid* set all three user IDs.

4.3.3 BSD

4.2 BSD kept the real uid and effective uid but changed the system call from *setuid* to *setreuid*. Processes could then directly control both their user IDs, under the following rules:

- If the effective uid was zero, the user might set the real uid and effective uid to any user ID.

- Otherwise, the user might set either the real uid or the effective uid to value of the other one.

Therefore, the *setreuid* system call enabled a process to swap the real uid and effective uid.

The POSIX standard [32] codified a new specification for the *setuid* call. In an attempt to be POSIX compliant, 4.4 BSD replaced 4.2 BSD's old *setreuid* model with the POSIX/System V style saved uid model. It modified *setuid* so that *setuid* set all three user IDs regardless of whether the effective uid of a process was zero, therefore allowing any process to permanently drop privileges.

4.3.4 Modern Unix

As System V and BSD influenced each other, both systems implemented *setuid*, *seteuid*, and *setreuid*, although with different semantics. None of these system calls, however, allowed the direct manipulation of the saved uid (although the saved uid could be modified indirectly through *setuid* and *setreuid*). Therefore, some modern Unix systems introduced a new call, *setresuid*, to allow the modification of each user ID directly.

4.4 Complexity of uid-setting system calls

A process modifies its user IDs by the uid-setting system calls: *setuid*, *seteuid*, *setreuid*, and in some systems, *setresuid*. Each of the system calls involves two steps. First, it checks if the process has permission to invoke the system call. If so, it then modifies the user IDs of the process according to certain rules.

In this section, we compare and contrast the semantics of the uid-setting system calls among Linux 2.4.18 [40], Solaris 8 [54], and FreeBSD 4.4 [27]. We discovered the behavior of the uid-setting system calls by a combination of manual inspection of the kernel source code and formal methods. We will defer discussion of the latter until Section 4.5.

The POSIX Specification To understand the semantics of the uid-setting system calls, we begin with the POSIX standard, which has influenced the design of the system calls in many systems. In particular, the behavior of *setuid(newuid)* is defined by the POSIX specification. See Figure 4.1 for the relevant text.

The POSIX standard refers repeatedly to the term *appropriate privileges*, which is defined in Section 2.3 of POSIX 1003.1-1988 as:

> An implementation-defined means of associating privileges with a process with regard to the function calls and function call options defined in this standard that need special privileges. There may be zero or more such means.

Essentially, the term *appropriate privilege* serves as a wildcard that allows compliant operating systems to use any policy whatsoever for deeming when a call to *setuid* should be allowed. The conditional flag {`_POSIX_SAVED_IDS`} parametrizes the specification, allowing POSIX-compatible operating systems to use either of two schemes (as described in Figure 4.1). We will see how dif-

If {`_POSIX_SAVED_IDS`} is defined:

1. If the process has *appropriate privileges*, the *setuid*() function sets the real user ID, effective user ID, and the [saved user ID] to *newuid*.

2. If the process does not have *appropriate privileges*, but *newuid* is equal to the real user ID or the [saved user ID], the *setuid*() function sets the effective user ID to *newuid*; the real user ID and [saved user ID] remain unchanged by this function call.

Otherwise:

1. If the process has *appropriate privileges*, the *setuid*() function sets the real user ID and effective user ID to *newuid*.

2. If the process does not have *appropriate privileges*, but *newuid* is equal to the real user ID, the *setuid*() function sets the effective user ID to *newuid*; the real user ID remains unchanged by this function call.

(POSIX 1003.1-1988, Section 4.2.2.2)

Figure 4.1: An excerpt from the POSIX specification [32] covering the behavior of the *setuid* system call.

ferent interpretations of the term *appropriate privilege* have led to considerable differences in the behavior of the uid-setting system calls between operating systems.

4.4.1 Operating system-specific differences

Much of the confusion is caused by different interpretations of *appropriate privileges* among Unix systems.

Solaris In Solaris 8, a System V based system, a process is considered to have *appropriate privileges* if its effective uid is zero (root). Also, Solaris defines {`_POSIX_SAVED_IDS`}. Consequently, calling *setuid(newuid)* sets all three user IDs to *newuid* if the effective uid is zero, but otherwise sets only the effective uid to *newuid* (if the *setuid* call is permitted).

FreeBSD FreeBSD 4.4 interprets *appropriate privileges* differently, as noted in Appendix B4.2.2 of POSIX:

> The behavior of 4.2BSD and 4.3BSD that allows setting the real ID to the effective ID is viewed as a value-dependent special case of *appropriate privilege*.

This means that a process is deemed to have *appropriate privileges* when it calls *setuid(newuid)* with *newuid=geteuid()*, in addition to when its effective uid is zero. Also in contrast to Solaris, FreeBSD does not define {`_POSIX_SAVED_IDS`}, although every FreeBSD process does have a saved uid. Therefore, by calling *setuid(newuid)*, a process sets both its real uid and effective uid to *newuid* if the system call is permitted, in agreement with POSIX. FreeBSD also sets the saved uid in all permitted *setuid* calls.

Linux Linux introduces a capability[2] model for finer-grained control of privileges. Instead of a single level of privilege determined by the effective uid (i.e., root or non-root), there are a number of capability bits each of which is used to determine access control to certain resources[3]. One of them, the *SETUID* capability, carries the POSIX *appropriate privileges*. To make the new capability model compatible with the traditional user ID model where *appropriate privileges* are carried by a zero effective uid, the Linux *SETUID* capability tracks the effective uid during all uid-setting system calls: Whenever the effective uid becomes zero, the *SETUID* capability is set; whenever the effective uid becomes non-zero, the *SETUID* capability is cleared.

However, the *SETUID* capability can be modified outside the uid-setting system calls. A process can clear its *SETUID* capability, and a process with the *SETPCAP* capability can remove the

[2] Beware: the word “capability” is a bit of a misnomer. In this context, it refers to special privileges that a process can possess, and not to the usual meaning in the security literature of an unforgeable reference. Regrettably, the former usage comes from the POSIX standard and seems to be in common use, and so we follow their convention.

[3] More accurately, a Linux process has three sets of capabilities, but only the set of *effective capabilities* determine access control. All references to *capabilities* in this dissertation refer to the effective capabilities.

SETUID capability of other processes (but note that in Linux 2.4.18, no process has or can acquire the *SETPCAP* capability, a change that was made to close a security hole; see Section 4.6.1 for details). Therefore, explicitly setting or clearing the *SETUID* capability changes the properties of uid-setting systems calls.

4.4.2 Comparison among uid-setting system calls

Next we compare the uid-setting system calls and point out several unexpected properties and an inconsistency in how the Linux kernel handles *fsuid*.

setresuid(newruid, neweuid, newsuid) *setresuid* has the clearest semantics among the four uid-setting system calls. The permission check for *setresuid()* is intuitive and common to all OSs: either the euid of the process must be root, or each of the three parameters must be equal to one of the three user IDs of the process. As each of the real uid, effective uid, and saved uid is set directly by *setresuid*, the programmer knows clearly what to expect after the call. Moreover, the *setresuid* call is guaranteed to have an all-or-nothing effect: if it succeeds, all the user IDs are changed; if it fails, none are. It will not fail after having changed some but not all of the user IDs.

Note that while FreeBSD and Linux offer *setresuid*, Solaris does not. However, Solaris does offer equivalent functionality via the `/proc` filesystem. Any process can examine its three user IDs, and a superuser process can set any of them, in line with the traditional System V notion of *appropriate privilege*.

seteuid(neweuid) *seteuid* has also a clear semantics. It sets the effective uid while leaving the real uid and saved uid unchanged. However, when the current effective uid is not zero, there is a slight difference in the permission required by *seteuid* among Unix systems. While Solaris and Linux allow the parameter *neweuid* to be equal to any of the three user IDs, FreeBSD only allows *neweuid* to be equal to either the real uid or saved uid; in FreeBSD, the effective uid is not used in the decision. As a surprising result, *seteuid(geteuid())*, which a programmer might intuitively expect to be always permitted, can fail in FreeBSD, e.g., when *ruid*=100, *euid*=200, and *suid*=100.

setreuid(newruid, neweuid) The semantics of *setreuid* is confusing. It modifies the real uid and effective uid, and in some cases, the saved uid. The rule by which the saved uid is modified is complicated. Furthermore, the permission required for *setreuid* differs among the three operating systems. In Solaris and Linux, a process can always swap the real uid and effective uid by calling *setreuid(geteuid(), getuid())*. In FreeBSD, however, *setreuid(geteuid(), getuid())* sometimes fails, e.g., when *ruid*=100, *euid*=200, and *suid*=100.

setuid(newuid) *setuid* is the only uid-setting system call standardized in POSIX 1003.1-1988, but ironically it is also the most confusing one. First, the required permission differs among Unix systems. Both Linux and Solaris require the parameter *newuid* to be equal to either the real uid or saved uid if the effective uid is not zero. As a surprising result, *setuid(geteuid())*, which a programmer might reasonably expect to be always permitted, can fail in some cases, e.g., when *ruid*=100, *euid*=200, and *suid*=100. On the other hand, *setuid(geteuid())* always succeeds in FreeBSD. Second, the action of *setuid* differs not only among different operating systems but also between privileged and unprivileged processes. In Solaris and Linux, if the effective uid is zero, a successful *setuid(newuid)* call sets all three user IDs to *newuid*; otherwise, it sets only the effective user ID to *newuid*. On the other hand, in FreeBSD a successful *setuid(newuid)* call sets all three user IDs to *newuid* regardless of the effective uid.

setfsuid(newfsuid) In Linux, each process has also an *fsuid* in addition to its real uid, effective uid, and saved uid. The *fsuid* is used for access control to the filesystem. It normally follows the effective uid unless it is explicitly set by the *setfsuid* system call. The Linux kernel tries to maintain the invariant that the *fsuid* is zero only if at least one of the real uid, effective uid, or saved uid is zero, as manifested in the comment in a source files. The rationale is that once a process has dropped the root privilege in each of its real uid, effective uid, and saved uid, the process cannot have any leftover root privilege in the *fsuid*. Since the *fsuid* is Linux specific, this invariant allows a cross-platform application that is unaware of the *fsuid* to drop all privileges securely.

Unfortunately, we discovered that a bug in the kernel up to version 2.4.18 violates this invariant. The bug is that while every successful *setuid* and *setreuid* call sets the *fsuid* to the effective uid, a successful *setresuid* call will fail to do the same if the effective uid does not change during the call [4]. This causes the call sequence in Figure 4.2 to violate the invariant. We reported and helped fix this bug in the kernel. Section 4.5.4 will describe how we discovered this bug using a formal model.

setgid() and relatives There are also a set of calls for manipulating group IDs, namely, *setgid*, *setegid*, *setregid*, and *setresgid*. They behave much like their setuid counterparts, with only one minor exception (the permission check in *setregid* differs slightly from *setreuid* in Solaris). However, the *appropriate privileges* are always carried by the *euid* in both setuid-like and setgid-like calls. Thus, an effective group ID of zero does not accord any special privileges to change groups. This is

[4] The *seteuid(euid)* call in Linux is implemented as *setreuid(-1, euid)* or *setresuid(-1, euid, -1)*, depending on the version of the C library. Hence, the *seteuid* system call might or might not set the *fsuid* reliably, depending on the C library version.

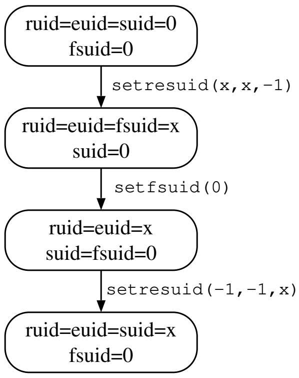


Figure 4.2: The call sequence shows that the Linux kernel violates the invariant: *the fsuid is zero only if at least one of the ruid, euid, or suid is zero*. In the figure, x represents a non-zero user ID.

a potential source of confusion: it is tempting to assume incorrectly that since *appropriate privileges* are carried by the *euid* in the setuid-like calls, they will be carried by the *egid* in the setgid-like calls, but this is not how it actually works. This misconception caused a mistake in the manual page of *setgid* in Redhat Linux 7.2 (Section 4.5.4).

In many Unix systems, a process has also a set of *supplementary group IDs* which are modified by the *setgroups* and *initgroups* calls. Since they are used less frequently and are less prone to misuse, we will not discuss them.

4.5 Formal models

We initially began to develop the summary in the previous section by manually reading operating system source code. Although reading kernel sources is a natural method to study the semantics of the uid-setting system calls, it has many serious limitations. First, it is laborious, especially when various Unix systems implement the system calls differently. Second, since our findings are based on current kernel sources, they may become invalid should the implementation change in the future. Third, we cannot prove that our findings are correct. Finally, informal specifications are not well-suited to programmatic use. MOPS needs formal specifications for checking proper usage of the uid-setting system calls. These problems with manual source code analysis motivate us to build a formal model of the uid-setting system calls.

4.5.1 Building a formal model

Our model of the uid-setting system calls is based on finite state automata. The operating system maintains per-process state (e.g., the real, effective, and saved uids) to track privilege levels, so it is natural to view the operating system as implementing a finite state automaton (FSA). A state of the FSA contains all the relevant information about the process, e.g., the three uids. Each uid-setting system call leads to a number of possible transitions; we label each transition with the system call that it comes from.

We construct the FSA in two steps: (1) determine its states by reading kernel sources; (2) determine its transitions by simulation. In the first step, we determine the states in the FSA by identifying kernel variables that affect the behavior of the uid-setting system calls. For example, if only the real uid, effective uid, and saved uid can affect the uid-setting system calls, then each state of the FSA is of the form (r, e, s), representing the values of the real, effective, and saved user IDs respectively.

This is a natural approach. However, an immediately problem is state space explosion in the FSA: in Linux, uids are 32-bit values, and so there are $(2^{32})^3 = 2^{96}$ possible states. Obviously, manipulating an FSA of such size is infeasible. Therefore, we need to somehow abstract away inessential details to reduce the size of the FSA substantially.

Fortunately, we note that the FSA has much symmetry. For each non-root user ID, the operating system behaves essentially independent of the actual value of this user ID; it behaves depending only on the fact that it is non-zero. For example, the states $(ruid, euid, suid) = (100, 100, 100)$ and $(200, 200, 200)$ are isomorphic up to a substitution of the value 100 by the value 200, since the OS will behave similarly in both cases (e.g., calling *setuid(0)* will fail in both cases). In general, we consider two states equivalent when each can be mutated into the other by a consistent substitution on non-root user IDs. By identifying equivalent states, we can shrink the size of the FSA considerably.

Now that we know that there must exist some reasonable FSA model, the next problem is how to compute it. Here we use *simulation*: if we simulate a pseudo-application that tries every possible system call to observe the state transitions performed by the operating system in response to these system calls, we can infer how the operating system behaves when real applications make these calls. Once we identify equivalent states, the state space will be small enough that we can exhaustively explore the entire state space of the operating system. To make this idea concrete, Figure 4.3 shows an algorithm to construct an FSA model using these techniques.

Note that by using simulation to create a model of the uid-setting system calls, we assume that while a process is executing such a call, the user IDs of the process cannot be modified outside the call. In other words, there is no race on the user IDs between a uid-setting system call and other parts of the kernel. This requirement might not hold in multi-threaded programs if multiple threads share the same user IDs.

Implementation Our implementation follows Figure 4.3 closely. (Note that the simulator must run as root.) We considered several optimizations and extensions to this basic algorithm.

One simple optimization is to use a depth-first search to explore only the reachable states. In our case, the state space is small enough that the improvement is probably unimportant, so we did

GETSTATE():
1. Call `getresuid(&r, &e, &s)`.
2. Return (r, e, s).

SETSTATE(r, e, s):
1. Call `setresuid(r, e, s)`.
2. Check for error.

GETALLSTATES():
1. Pick n arbitrary uids $u_1, \ldots, u_n$.
2. Let $U := \{u_1, \ldots, u_n\}$.
3. Let $S := \{(r, e, s) : r, e, s \in U\}$.
4. Let $C := \{$`setuid(x)`, `setreuid(x, y)`, `setresuid(x, y, z)`, $\cdots$
$\qquad : x, y, z \in U \cup \{-1\}\}$.
5. Return (S, C).

BUILDMODEL():
1. Let $(S, C) :=$ GETALLSTATES().
2. Create an empty FSA with statespace S.
3. For each $s \in S$, do:
4.   For each $c \in C$, do:
5.   Fork a child process, and within the child, do:
6.   Call SETSTATE(s), and then invoke c.
7.   Finally, let $s' :=$ GETSTATE(),
                pass s' to the parent process, and exit.
8.   Add the transition $s \xrightarrow{c} s'$ to the FSA.
9. Return the newly-constructed FSA as the model.

Figure 4.3: The model-extraction algorithm.

not implement this optimization. A more dangerous optimization would be to emulate the behavior of the operating system from user-level by cutting-and-pasting the source code of the setuid system calls from the kernel into our simulation engine. This would speed up model construction, but the performance improvement comes at a severe price: it is hard to ensure that our emulation of the OS is completely faithful. In any case, our unoptimized implementation already takes only a few seconds to generate the model. For these reasons, we do *not* apply this optimization in our implementation.

To ensure maximum confidence in the correctness of our results, we check in two different ways that the call to *setresuid* in line 1 of SETSTATE() succeeds. First, we check the return value from the operating system. Second, we call *getresuid* and check that all three user IDs have been set as desired (see Section 4.7.1).

On Solaris, there are no *getresuid* and *setresuid* system calls. Fortunately, we can simulate them using the `/proc` filesystem. We read the three user IDs of a process from its *cred* file, and we modify the user IDs by writing to its *ctl* file (see *proc(4)* for details).

On Linux, we also model the *SETUID* capability bit by adding a fourth dimension to the state tuple. Thus, states are of the form (r, e, s, b) where the bit b is true whenever the *SETUID* capability is enabled. This allows us to model accurately the case where an application explicitly clears or sets its *SETUID* capability bit; though we are unaware of any real application that does this, if we ever do encounter such an application our model will still remain valid.

On all operating systems, we extend our model further to deal with system calls that fail (i.e., when invoking call c in line 6 of BUILDMODEL()). It is sometimes useful to be able to reason about whether a system call has succeeded or failed, and one way is to add a bit to the state denoting whether the previous system call returned successfully or not.

Also, on all operating systems we extend our model to include group IDs. This adds three additional dimensions to the state: real gid, effective gid, and saved gid[5]. In this way, we can model the semantics of the gid-setting system calls. On Linux, we also add a bit to indicate whether the *SETGID* capability is enabled.

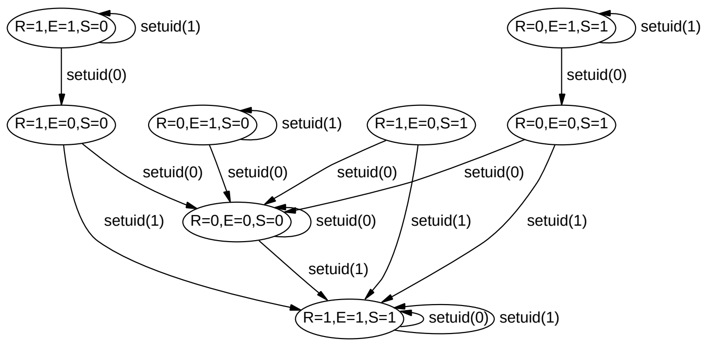
4.5.2 Examples of formal models

In this section, we show a series of formal models of the uid-setting system calls created using the algorithm in Figure 4.3. These models differ in their set of user ID values. In other words, they differ in the user ID values picked in step 1 of $\textsc{GetAllStates}()$ subroutine in Figure 4.3.

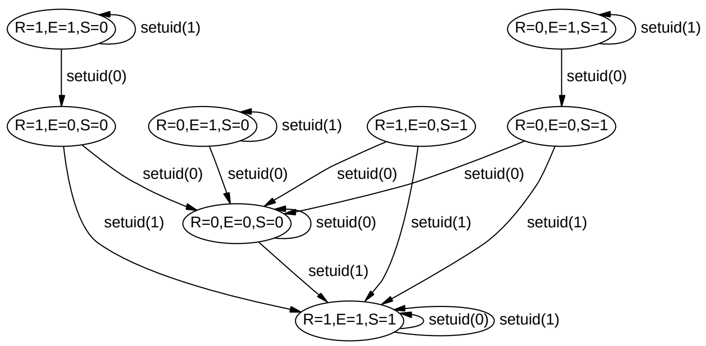
We start with a simple model where the set of user ID values is $\{0, x\}$ where x is a non-root user ID. Although simple, this model is accurate for many applications that manipulate at most one non-root user ID at a time. For instance, a state like $(100, 200, 100)$ will never appear in such an application. Each state in this simple FSA has three bits, each representing whether the real uid, effective uid, or saved uid is root or not. All together there are eight states in the FSA. Figure 4.4 shows graphically the models that we obtains in this way for the *setuid* call on Linux, Solaris, and FreeBSD. Note that the models on Solaris and Linux are equivalent, but they differ from the model on FreeBSD. Figure 4.5 shows the models for the *seteuid*, *setreuid*, and *setresuid* calls on Linux.

Figure 4.6 shows a variation of the previous models where the set of user ID values is $\{x, y\}$ where x and y are distinct non-root user ID values. This model is appropriate for applications that switch between two non-root user IDs (rather than between the root and a non-root user ID).

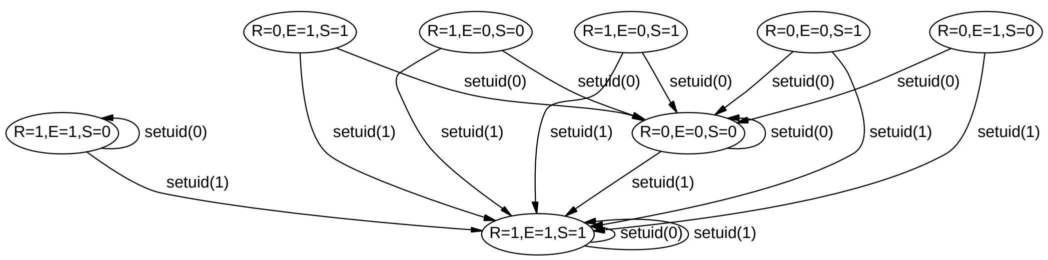
[5]We don't currently model supplemental groups. Note that this omission does not invalidate our model, as supplemental groups are only used in access control checks and never affect the behavior of the uid-setting system calls.



(a) An FSA describing *setuid* in Linux 2.4.18

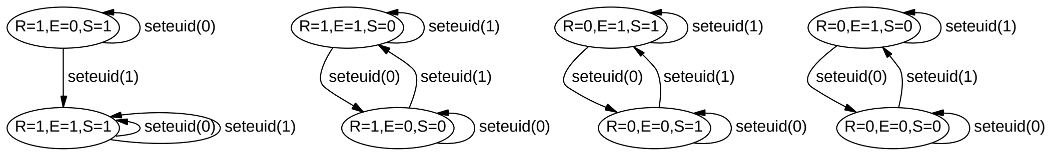


(b) An FSA describing *setuid* in Solaris 8

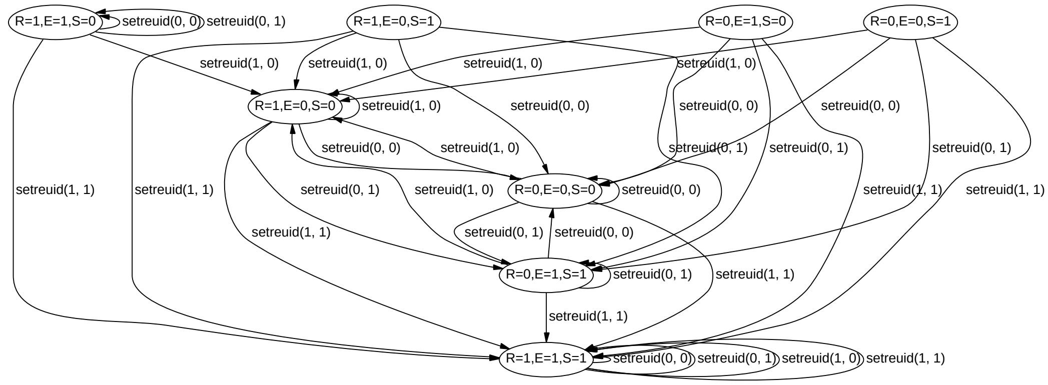


(c) An FSA describing *setuid* in FreeBSD 4.4

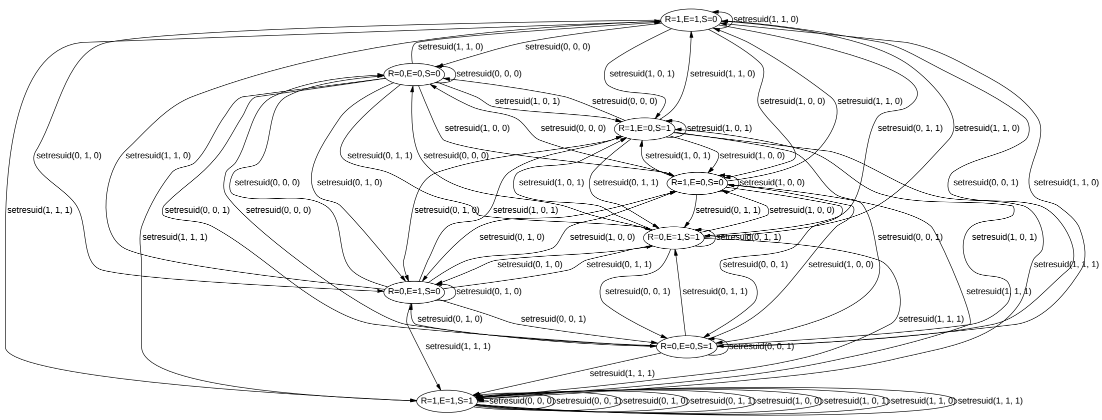
Figure 4.4: Three finite state automata describing the *setuid* system call in Linux, Solaris, and FreeBSD, respectively. Ellipses represent states of the FSA, where a notation like "R=1,E=0,S=1" indicates that $euid = 0$ and $ruid = suid \neq 0$. Each transition is labeled with the system call it corresponds to. To avoid cluttering the diagram, we omit the error states and (in Linux) the capability bits that otherwise would appear in our deduced model.



(a) An FSA describing *seteuid* in Linux



(b) An FSA describing *setreuid* in Linux



(c) An FSA describing *setresuid* in Linux

Figure 4.5: Three finite state automata describing the *seteuid*, *setreuid*, *setresuid* system calls in Linux respectively. Ellipses represent states of the FSA, where a notation like “R=1,E=0,S=1” indicates that $euid = 0$ and $ruid = suid \neq 0$. Each transition is labeled with the system call it corresponds to.

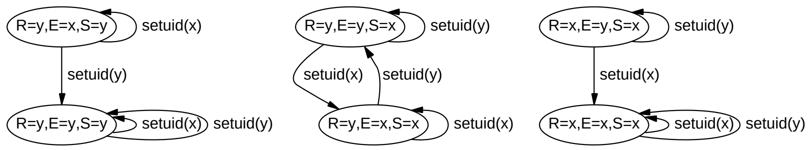


Figure 4.6: A finite state automaton describing the *setuid* system call in Linux. This FSA considers only two distinct non-root user ID values x and y. Ellipses represent states of the FSA, where a notation like "R=x,E=y,S=x" indicates that *euid* $= y$ and *ruid* $=$ *suid* $= x$. Each transition is labeled with the system call it corresponds to.

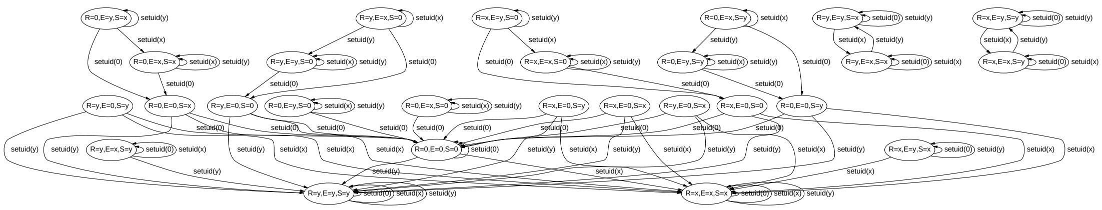


Figure 4.7: A finite state automaton describing the *setuid* system call in Linux. This FSA considers three user ID values: the root user ID and two distinct non-root user ID values x and y. Ellipses represent states of the FSA, where a notation like "R=0,E=x,S=y" indicates that *ruid* $= 0$, *euid* $= x$ and *suid* $= y$. Each transition is labeled with the system call it corresponds to.

This model is appropriate for analyzing BSD games [7] run under the dungeon master. Foley's work [24] offers a more serious use of this model.

We can easily extend the simple models to include more user ID values, which are appropriate for applications that use more than two user ID values. Figure 4.7 shows a model where the set of user ID values is $\{0, x, y\}$ where x and y are distinct non-root user ID values. This is the fully general model of Unix user IDs.

4.5.3 Correctness

Our model-extraction algorithm (Figure 4.3) is an instance of a more general schema for inferring finite-state models, specialized by including application-dependent implementations of the GETSTATE(), SETSTATE(), and GETALLSTATES() subroutines. We argue that our algorithm is correct by arguing that the general version is correct.

We frame our theoretical discussion in terms of equivalence relations. Let $\mathcal{S}$ denote the set of concrete states (e.g., triples of 32-bit uids) and $\mathcal{C}$ the set of concrete system calls. Write $s \overset{c}{\rightsquigarrow} t$ if the operating system will always transition from state s to t upon invocation of c. We will need equivalence relations $\equiv_{\mathcal{S}}$ on $\mathcal{S}$ and $\equiv_{\text{OS}}$ on $\mathcal{S} \times \mathcal{C}$ that are respected by the operating system: in other words, if $s \overset{c}{\rightsquigarrow} t$ and $s \equiv_{\mathcal{S}} s'$, then there is some state t' and some call c' so that $(s, c) \equiv_{\text{OS}} (s', c')$, $t \equiv_{\mathcal{S}} t'$, and $s' \overset{c'}{\rightsquigarrow} t'$. The intuition is that calling c from s is somehow isomorphic to calling c' from s'. Also, we require that whenever $(s, c) \equiv_{\text{OS}} (s', c')$ holds, then $s \equiv_{\mathcal{S}} s'$ does, too.

A critical requirement is that the operating system must behave *deterministically* given the equivalence class of the current state. More precisely, if $s \overset{c}{\rightsquigarrow} t$ and $s' \overset{c'}{\rightsquigarrow} u$ where $(s, c) \equiv_{\text{OS}} (s', c')$, then we require $t \equiv_{\mathcal{S}} u$. The intuition is that the behavior of the operating system will depend only on which equivalence class we are in, and not on any other information about the state. For instance, the behavior of the operating system cannot depend on any global variables that don't appear in the state s; if it does, these global variables must be included into the state space $\mathcal{S}$. As another example, a system call implementation that attempts to allocate memory and returned an error code if this allocation fails will violate our requirement, because the success or failure of the memory allocation introduces non-determinism, which is prohibited. We can see that this requirement is non-trivial, and it must be verified by manual inspection of the source code before our algorithm in Figure 4.3 can be safely applied; we will return to this issue later.

Next, there are three requirements on the instantiation of the GETSTATE(), SETSTATE(), and GETALLSTATES() subroutines. First, the GETSTATE() routine must return (a representative for) the equivalence class of the current state of the operating system. Note that it is natural to represent equivalence classes internally by singling out a unique representative for each equivalence class and using this value. Second, the SETSTATE() procedure with parameter s must somehow cause the operating system to enter a state s' in the same equivalence class as s (the implementation may freely choose one). Finally, the GETALLSTATES() function must return a pair (S, C) so that S contains at least one representative from each equivalence class of $\equiv_{\mathcal{S}}$ and so that every equivalence class of $\equiv_{\mathrm{OS}}$ contains some element (s, c) with $c \in C$.

When these general requirements are satisfied, the BUILDMODEL() algorithm from Figure 4.3 will correctly infer a valid finite-state model for the underlying operating system. The proof is easy. We will write $[x]$ for the equivalence class containing x, e.g., $[s] = \{t \in \mathcal{S} : s \equiv_{\mathcal{S}} t\}$. If $s \xrightarrow{c} t$ appears in the final FSA output by BUILDMODEL(), then there must have been a step at which, for some $s' \in [s]$, $t' \in [t]$, and c' with $(s, c) \equiv_{\mathrm{OS}} (s', c')$, we executed c' in state s' at line 6 and transitioned to state t'. (This follows from the correctness of SETSTATE() and GETSTATE().) The latter means that $s' \overset{c'}{\rightsquigarrow} t'$, from which it follows that $s \overset{c}{\rightsquigarrow} t''$ for some $t'' \in [t]$, since the OS respects $\equiv_{\mathrm{OS}}$. Conversely, if $s' \overset{c'}{\rightsquigarrow} t'$ for some s', c', t', then by the correctness of GETALLSTATES(), there will be some s and c satisfying $(s, c) \equiv_{\mathrm{OS}} (s', c')$ so that we enter line 6 with s, c, and thanks to the deterministic nature of the operating system we will discover the transition $s \xrightarrow{c} t$ for some $t \equiv_{\mathcal{S}} t'$. Thus, the FSA output by BUILDMODEL() is exactly what it should be. Consequently, all that remains is to check that these requirements are satisfied by our instantiation of the schema.

We argue this next for the implementation shown in Figure 4.3. Let $\mathcal{U}$ denote the set of concrete uids (e.g., all 32-bit values), so that $\mathcal{S} = \mathcal{U} \times \mathcal{U} \times \mathcal{U}$. Say that a map $\sigma : \mathcal{U} \to \mathcal{U}$ is a *valid substitution* if it is bijective and fixes 0, i.e., $\sigma(0) = 0$. Each such substitution can be extended to one on $\mathcal{S}$ by working component-wise, i.e., $\sigma(r, e, s) = (\sigma(r), \sigma(e), \sigma(s))$, and we can extend it to work on system calls by applying the substitution to the arguments of the system call, e.g., $\sigma(\texttt{setreuid}(r, e)) = \texttt{setreuid}(\sigma(r), \sigma(e))$. We define our equivalence relation $\equiv_{\mathcal{S}}$ on $\mathcal{S}$ as follows: two states $s, s' \in \mathcal{S}$ are equivalent if there is a valid substitution σ such that $\sigma(s) = s'$. Similarly, $(s, c) \equiv_{\text{OS}} (s', c')$ holds if there is some valid substitution σ so that $\sigma(s) = s'$ and $\sigma(c) = c'$.

The correctness of GETSTATE() and SETSTATE() is immediate. Also, so long as $n \geq 6$, GETALLSTATES() is correct since the choice of uids $u_1, \ldots, u_n$ is immaterial: every pair $(s, c) \in \mathcal{S} \times \mathcal{C}$ is equivalent to some pair $(s', c') \in S \times C$, since we can simply map the first six non-zero uids in (s, c) to $u_1, \ldots, u_6$ respectively, and there can be at most six non-zero uids in (s, c). Actually, we can see that the algorithm in Figure 4.3 comes from a finer partition than that given by $\equiv_{\text{OS}}$: for example, (u_1, u_1, u_1) and (u_2, u_2, u_2) are unnecessarily distinguished. This causes no harm to the correctness of the result, and only unnecessarily increases the size of the resulting FSA. We gave the variant shown in Figure 4.3 because it is simpler to present, but in practice our implementation does use the coarser relation $\equiv_{\mathcal{S}}$.

All that remains to check is that the operating system respects and behaves deterministically with respect to this equivalence class. We verify this by manual inspection of the kernel sources, which shows that in Linux, FreeBSD, and Solaris the only operations that the uid-setting system calls perform on user IDs are equality testing of two user IDs, comparison to zero, copying one user ID to another, and setting a user ID to zero. Moreover, the operating system behavior does not depend on anything else, with one exception: Linux depends on whether the *SETUID* capability is enabled for the process, so on Linux we add an extra bit to each state indicating whether this capability is enabled. Thus, our verification task amounts to checking that user IDs are treated as an abstract data type with only four operations (equality testing, comparison to zero, and so on) and that the side effects and results of the system call do not depend on anything outside the state S. In our experience, verifying that the operating system satisfies these conditions is much easier than fully understanding its behavior, as the former is an almost purely mechanical process.

This completes our justification for the correctness of our method for extracting a formal model to capture the behavior of the operating system.

4.5.4 Applications

Although our main purpose for building the formal model of the uid-setting system calls is to check whether programs use these calls properly, this model has many other applications as well. We have already discussed in Section 4.4 the semantics of the setuid system calls and pointed out pitfalls; this relied heavily on the FSA formal model. Next, we will discuss several additional applications: verifying documentation and checking conformance with informal specifications; identifying cross-platform semantic differences that might indicate potential portability issues; detecting inconsistency in the handling of user IDs within an OS kernel.

Verifying the accuracy of manual pages

Manual pages are the primary source of information for Unix programmers, but unfortunately they are often incomplete or wrong. FSAs are useful in verifying the accuracy of manual pages of uid-setting system calls. For each call, if its FSA is small and its description in manual pages is simple, we check if each transition in the FSA agrees with the description by hand. Otherwise, we build another FSA based on the description and compare this FSA to the original FSA built by simulation. Differences between the two FSAs indicate discrepancies between the behavior of the system call and its description in manual pages.

The following are a few examples of problematic documentation that we have found using our formal model:

- The man page of *setuid* in Redhat Linux 7.2 fails to mention the *SETUID* capability, which affects the behavior of *setuid*.
- The man page of *setreuid* in FreeBSD 4.4 says:

  > Unprivileged users may change the real user ID to the effective user ID and vice-versa; only the super-user may make other changes.

  However, this is incorrect. Swapping the real uid and effective uid does not always succeed, such as when ruid=100, euid=200, suid=100, contrary to what the man page suggests. The correct description should be "Unprivileged users may change the real user ID to the real uid or saved uid, and change the effective uid to the real uid, effective uid, or saved uid."
- The man page of *setgid* in Redhat Linux 7.2 says

  > The setgid function checks the effective gid of the caller and if it is the superuser, all process related group ID's are set to gid.

  In reality, the function checks the effective **uid** instead of the effective **gid**.

Identifying implementation differences

Since various Unix systems implement the uid-setting system calls differently, it is difficult to identify their semantic differences via reading kernel sources. Instead, we can create an FSA of these calls in each Unix system and contrasting the FSAs. For example, Figure 4.4 shows clearly that the semantics of *setuid* on Solaris is different from that on FreeBSD and Linux.

We can further formalize this approach by taking the symmetric difference of FSAs. In particular, if M, M' are two FSAs for two Unix platforms with the same state space, we can find portability issues as follows. Compute the parallel composition $M \times M'$, whose states are pairs (s, s') with s a state from M and s' a state from M'. Then, mark as an accepting state of $M \times M'$ any pair (s, s') where $s \neq s'$. Now any execution trace that starts at a non-accepting state and eventually reaches an accepting state indicates a sequence of system calls whose semantics is not the same on both operating systems. This indicates a potential portability issue, and all such differences can be computed via a simple reachability computation (e.g., depth-first search).

Detecting inconsistency within an OS kernel

An OS kernel maintains many invariants that both the kernel itself and many application programs depend on, so violation of these invariants may cause vulnerabilities in both the OS and applications.

The Linux kernel documents the invariant that the *fsuid* is zero only if at least one of the real uid, effective uid, or saved uid is zero. To verify this invariant, we extended the formal model of user IDs with the *fsuid* and created an FSA using the algorithm in Figure refbuildmodel. From the FSA, we discovered that the invariant does not always hold, because the state where *fsuid* $= 0$ and *ruid* $\neq 0$, *euid* $\neq 0$, *suid* $\neq 0$ is reachable. For example, the call sequence in Figure 4.2 will violate the invariant. The problem results from an inconsistency in how the kernel handles the *fsuid* in the uid-setting system calls. While every successful *setuid* and *setreuid* call sets the *fsuid* to the effective uid, a successful *setresuid* call will fail to do the same if the effective uid does not change during the call. We contributed a patch to the Linux kernel version 2.4.19 and later.

4.6 Case studies of security vulnerability

Misuses of uid-setting system calls have caused many security vulnerabilities, which are good lessons in learning the proper usage of the system calls. We will analyze two such incidents in older versions of sendmail.

Sendmail [33] is a commonly used Mail Transmission Agent(MTA). It runs in two modes: (1) as a daemon that listens on port 25 (SMTP), and (2) via a Mail User Agent to submit mail to the mail queue. In the first case, all three user IDs of the sendmail process are typically zero, as it is run by the superuser *root* in the boot process. In the second case, however, sendmail is run by an ordinary user. As the mail queue is not world writable, sendmail requires privilege to access the mail queue.

4.6.1 Misuse of setuid

Next we describe a vulnerability that was caused by a misuse of setuid [34]. Sendmail 8.10.1 installed the *sendmail* binary as a setuid-root executable. When it was executed by a non-root user, the real uid of the process was the non-root user while both the effective uid and saved uid were zero. This gave *sendmail* permission to write to the mail queue since its effective uid was zero. To minimize risks in the event that an attacker takes over *sendmail* and executes malicious code with root privilege, *sendmail* permanently dropped root privilege before doing potentially dangerous operations requested by an user. This was done by calling *setuid(getuid())*, which sets all three user IDs to the non-root user.

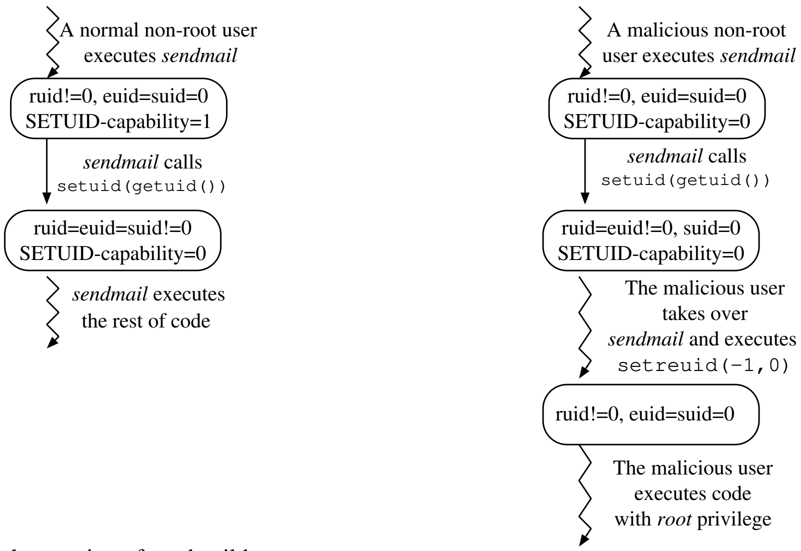
POSIX specifies that if a process has *appropriate privileges*, *setuid(newuid)* sets all three user IDs to *newuid*; otherwise, *setuid(newuid)* only sets the effective uid to *newuid* (if *newuid* is equal to the real uid or saved uid). In Linux, *appropriate privileges* are carried by the *SETUID* capability. Furthermore, after any uid-setting system call, the Linux kernel sets or clears the *SETUID* capability bit, if necessary, to establish a simple post-condition: the *SETUID* capability should be set if and only if the effective uid is zero.

However, prior to version 2.2.16 of Linux, there was a bug in the kernel that made it possible for a process to clear its *SETUID* capability bit even when its effective uid was zero. In this case, calling *setuid(getuid())* only modified the effective uid, and under these conditions, *sendmail* would only drop root privilege from its effective uid but not its saved uid. Consequently, any malicious local user who could take over *sendmail* (e.g., with a buffer overrun attack) could restore root privilege in the effective uid by calling *setreuid(-1, 0)*. In other words, an attacker could ensure *sendmail*'s attempt to drop all privileges would fail, thereby raising the risk of a root attack on sendmail. Figure 4.8 illustrates the vulnerability.

The vulnerability was caused by the overloaded semantics of *setuid*. Depending on whether a process has the *SETUID* capability, setuid sets one user ID or all three user IDs, but it returns a success code in both cases. The vulnerability can be avoided by replacing *setuid(newuid)* with *setresuid(newuid, newuid, newuid)* if available, or with *setreuid(newuid, newuid)* otherwise.

4.6.2 Interaction between user IDs and group IDs

Another vulnerability in Sendmail was caused by an interaction between the user IDs and the group IDs [62]. To further reduce the risk from a malicious user taking over *sendmail*, since version 8.12.0 Sendmail no longer installed *sendmail* as a setuid-root program. To give *sendmail* permission to write to the mail queue, the mail queue was configured to be writable by group *smmsp*, and *sendmail* was installed as setgid-*smmsp*. Therefore, when *sendmail* was executed by a non-root



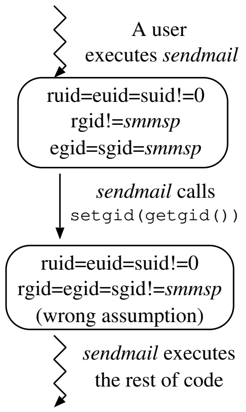
(a) A normal execution of sendmail by a non-root user

(b) An execution of sendmail by an attacker

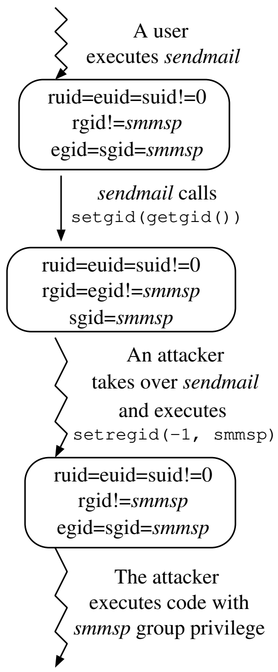
Figure 4.8: A vulnerability in sendmail due to a misuse of *setuid*. Note the failure: the programmer assumed that *setuid(getuid())* would always succeed in dropping all privilege, but by disabling the *SETUID* capability, the attacker is able to violate that expectation.

user, the real gid of the process was the primary group of the user, but the effective gid and saved gid were *smmsp*.

For the same reason that it permanently dropped root privilege in previous versions, now *sendmail* permanently dropped *smmsp* group privilege before executing potentially malicious directives from a user. Similar to the use of *setuid(getuid())* to permanently drop root privilege, *sendmail* called *setgid(getgid())* to permanently drop *smmsp* group privilege. However, since *sendmail* no longer had *appropriate privileges* because its effective uid was not zero anymore, *setgid(getgid())* only dropped the privileged group ID *smmsp* from the effective gid but left it in the saved gid. Consequently, any malicious user who found some way to take over sendmail (e.g., by a buffer overrun) could restore the *smmsp* group privilege in the effective gid by calling *setgid(-1, smmsp)*. This is illustrated in Figure 4.9.



(a) The programmer's mental model of an expected execution trace



(b) Real execution of sendmail by a malicious user

Figure 4.9: A vulnerability in sendmail due to interaction between user IDs and group IDs. The failure occurs because the programmer has overlooked that she has already dropped root privilege and hence no longer has the *appropriate privileges* to drop all group privileges in the *setgid* call.

The vulnerability was caused by an interaction between the user IDs and group IDs since changing user IDs may affect the property of *setgid*. To avoid the vulnerability, we can replace *setgid(newgid)* with *setresgid(newgid, newgid, newgid)* if available, or *setregid(newgid, newgid)* otherwise. The vulnerability also shows that if both user IDs and group IDs are to be modified, the modification should follow a specific order (Section 4.7.1).

4.7 Guidelines

We provide guidelines on the proper usage of the uid-setting system calls. First, we discuss general guidelines that apply to all setuid programs. Then, we focus on applications that use the uid-setting system calls in a specific way. We propose a high-level API for these applications to manage their privileges. The API is easier to understand and to use than the Unix API.

4.7.1 General guidelines

Selecting an appropriate system call

Since *setresuid* has a clear semantics and is able to set each user ID individually, programs should always use it if it is available. Otherwise, programs should use *seteuid(new_euid)* to set only the effective uid, and use *setreuid(new_uid, new_uid)* to set all three user IDs.

Programs should avoid *setuid* because its overloaded semantics and inconsistent implementation in different Unix systems may cause confusion and security vulnerabilities for the unwary programmer. As described in Section 4.4.2, in Linux or Solaris, if the effective user ID is zero, *setuid(newuid)* sets all three user IDs to *newuid*; otherwise, it sets only the effective user ID to *newuid*. On the other hand, in FreeBSD *setuid(newuid)* sets all three user IDs to *newuid* regardless of the effective user ID. We envision the following scenarios where *setuid* may be misused:

- If a setuid-root program temporarily drops root privilege with *seteuid(getuid())* and later calls *setuid(getuid())* with the intention of permanently dropping all root privileges, the program does not get the intended behavior on Linux or Solaris, because the saved user ID remains root. (However, the program does receive the intended behavior on FreeBSD.)

- Also on Linux or Solaris, in a setuid-root program, calling *setuid(getuid())* permanently drops root privileges; however, in a setuid-non-root program (e.g., a program that is setuid-Alice where Alice is a non-root user), calling *setuid(getuid())* will not permanently drop Alice's privileges, because the saved user ID remains Alice. This is particularly confusing, because the way setuid-root programs permanently drop privileges does not work in setuid-non-root programs on Linux or Solaris.

Obeying the proper order of system calls

The POSIX-defined *appropriate privileges* affect the actions of both system calls that set user IDs and that set group IDs. Since often *appropriate privileges* are carried by the effective uid, a program should drop group privileges before dropping user privileges permanently. Otherwise, after permanently dropping user privileges, the program may be unable to permanently drop group privileges. For example, the program in Figure 4.10(a) is able to permanently drop both user and group privileges because it calls *setgid* before *setuid*. In contrast, the program in Figure 4.10(b) calls *setuid* before *setgid*, so it fails to drop group privileges permanently.

(a) A program correctly drops both user and group privileges permanently by calling *setgid(getgid())* before *setuid(getuid)*

(b) A program fails to drop group privileges permanently because it calls *setuid(getuid())* before *setgid(getgid())*

Figure 4.10: Proper order of dropping user and group privileges. Figure (a), on the left, shows proper usage; figure (b) shows what can go wrong if one gets the order backwards.

Verifying proper execution of system calls

Since the semantics of the uid-setting system calls may change, e.g., when the kernel changes or when an application is ported to a different Unix system, it is imperative to verify successful execution of these system calls.

Checking return codes The uid-setting system calls return zero on success and non-zero on failure, so a process should check the return codes to verify the successful execution of these calls. This is especially important when a process permanently drops privilege, since such an action usually precedes operations that, if executed with privilege, may compromise the system.

Be aware that the Linux-specific *setfsuid* system call returns the previous *fsuid* from before the call and does not return any error message to the caller on failure. This is one motivation for our next guideline.

Verifying user IDs However, checking return codes may be insufficient for uid-setting system calls. For example, in Linux and Solaris, depending on the effective uid, *setuid(newuid)* may either (1) set all three user IDs (if the effective uid is zero), or (2) set only the effective uid (if it is non-zero), but the system call returns the same success code in both cases. The return code does not indicate which of the two cases has happened, so checking return codes is insufficient to guarantee successful completion of the uid operation in some cases. Moreover, the return code of the *setfsuid* call does not indicate whether the operation has succeeded: it returns the *fsuid* before the call.

Therefore, after each uid-setting system call, a program should verify that all its user IDs are as expected. On many Unix systems, a program can query all its three user ID values. For example, on Linux and FreeBSD the program may call *getresuid*; on Solaris the program may use the */proc* filesystem. If a Unix system does not allow programs to query all the three user ID values, a program may call *getuid* and *geteuid* to check the real uid and effective uid. A Linux process may examine its *fsuid* via the `/proc` filesystem, since Linux does not offer a *getfsuid* call.

Verifying failures Once an attacker has controlled a process, the attacker may insert arbitrary code into the process. Therefore, to assure security further, the process should ensure that all undesirable uid-setting system calls will fail. For example, after dropping privilege permanently, the process should verify that attempts to restore privilege will fail. This is shown in Figure 4.11.

4.7.2 An improved API for privilege management

Although the general guidelines in Section 4.7.1 can help programmers to use the uid-setting system calls more securely, programmers still have to grapple with the complex semantics of the uid-setting system calls and their differences among Unix systems. The complexity is partly

```
// drop privilege
setuid(getuid());
// verify that the process cannot restore privilege
if (setreuid(-1, 0) == 0)
  return ERROR;
```

Figure 4.11: An example of a program that verifies that it has properly dropped root privileges. The verification is achieved by checking that forbidden uid-setting system calls will fail. Note that a full implementation should also check the return code from *setuid* and verify that all three user IDs are as expected after the call to *setuid*.

due to a mismatch between the low-level semantics of the system calls, which describes how to modify the user IDs, and the high-level goals of the programmer, which represent a policy for when the application should run with privilege. We propose to resolve this tension by introducing an API that better matches the needs of application programmers.

Proposed API

In many applications, privilege management can typically be broken down into the following tasks:

- Drop privilege temporarily, in a way that allows the privilege to be restored later.
- Drop privilege permanently, so that it can never be restored.
- Restore privilege.

We propose a new API that offers the ability to perform each of these tasks directly and easily. The API contains three functions:

- *drop_priv_temp(new_uid)*: Drop privilege temporarily. Move the privileged user ID from the effective uid to the saved uid. Assign *new_uid* to the effective uid.

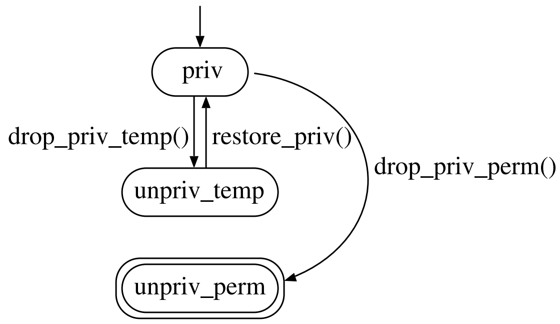


Figure 4.12: An FSA showing the behavior of a process when calling the functions of the new API.

- *drop_priv_perm(new_uid)*: Drop privilege permanently. Assign *new_uid* to all the real uid, effective uid, and saved uid.

- *restore_priv*: Restore privilege. Copy the privileged user ID from the saved uid to the effective uid.

By raising the level of abstraction, we allow programmers to think more about their desired security policy and less about the mechanism of implementing this policy. Figure 4.12 illustrates the action of these functions pictorially with a simple FSA.

Implementation

We implement the new API as wrapper functions to the uid-setting system calls. The implementation uses *setresuid* if available since it is able to set each of the user IDs independently, as shown in Figure 4.13. If *setresuid* or its equivalent is not available, the implementation uses *seteuid* and *setreuid*, as shown in Figure 4.14.

To use this implementation, an application must meet the following requirements:

- When the process starts, its effective uid contains the privileged user ID. This is true in most circumstances. When a process is run by a privileged user, all three user IDs contain the

```
int drop_priv_temp(uid_t new_uid)
{
  if (setresuid(-1, new_uid, geteuid()) < 0)
    return ERROR_SYSCALL;
  if (geteuid() != new_uid)
    return ERROR_SYSCALL;
  return 0;
}

int drop_priv_perm(uid_t new_uid)
{
  uid_t ruid, euid, suid;
  if (setresuid(new_uid, new_uid, new_uid) < 0)
    return ERROR_SYSCALL;
  if (getresuid(&ruid, &euid, &suid) < 0)
    return ERROR_SYSCALL;
  if (ruid != new_uid || euid != new_uid ||
      suid != new_uid)
    return ERROR_SYSCALL;
  return 0;
}

int restore_priv()
{
  int ruid, euid, suid;
  if (getresuid(&ruid, &euid, &suid) < 0)
    return ERROR_SYSCALL;
  if (setresuid(-1, suid, -1) < 0)
    return ERROR_SYSCALL;
  if (geteuid() != suid)
    return ERROR_SYSCALL;
  return 0;
}
```

Figure 4.13: A possible implementation of the high-level API for systems with *setresuid*.

```
uid_t priv_uid;

int drop_priv_temp(uid_t new_uid)
{
  int old_euid = geteuid();

  // copy euid to suid
  if (setreuid(getuid(), old_euid) < 0)
    return ERROR_SYSCALL;
  // set euid as new_uid
  if (seteuid(new_uid) < 0)
    return ERROR_SYSCALL;
  if (geteuid() != new_uid)
    return ERROR_SYSCALL;
  priv_uid = old_euid;
  return 0;
}

int drop_priv_perm(uid_t new_uid)
{
  uid_t suid;
  if (setreuid(new_uid, new_uid) < 0)
    return ERROR_SYSCALL;
  // OS specific way of reading suid
  suid = read_suid_from_proc_filesystem();
  if (getuid() != new_uid ||
      geteuid() != new_uid ||
      suid != new_uid)
    return ERROR_SYSCALL;
  return 0;
}

int restore_priv()
{
  if (seteuid(priv_uid) < 0)
    return ERROR_SYSCALL;
  if (geteuid() != priv_uid)
    return ERROR_SYSCALL;
  return 0;
}
```

Figure 4.14: A possible implementation of the high-level API for systems without *setresuid*.

privileged user ID. If the process is run as a privileged user, i.e., its executable is setuid'ed to the privileged user and is run by an unprivileged user, both the effective uid and saved uid of the process contain the privilege user ID.

- If the privileged user ID is not zero, then the unprivileged user ID must be stored in the real uid when the process starts. This requirement enables the process to replace the privileged user ID in the effective uid with the unprivileged user ID in *drop_priv_temp* and *drop_priv_perm*. This is the case when a non-root user executes an executable that is setuid'ed to another non-root user. On the other hand, if the privileged user ID is zero, then there is no such requirement, since the process can set its user IDs to arbitrary values.

- The process does not make any uid-setting system calls that change any of the three user IDs. Such a call may cause the process to enter a state not covered by the FSA in Figure 4.12, on which the high-level API and the implementation are based.

The implementation has the following beneficial properties:

- It does not affect the real uid.

- It guarantees that all transitions in Figure 4.12 succeed.

- It verifies that the user IDs are as expected after each uid-setting system call.

- It does the right thing even in cases where root is not involved, i.e., where the privileged user ID is not the superuser.

We can extend this basic implementation to include stronger safeguards against programming errors or OS inconsistency. To prevent a program from restoring a wrong privilege, we can let the function *restore_priv* take a parameter and check that the parameter matches the privilege stored in the saved user ID (Figure 4.13) or in the variable *priv_uid* (Figure 4.14). Another improvement is to let the function *drop_priv_perm* verify that an attempt to regain privilege will fail, as described in Section 4.7.1.

Chapter 5

Experience with Open Source Applications

To evaluate MOPS's effectiveness as a practical tool for checking real world programs, we used MOPS to check both large application programs and kernel code. This chapter describes our experience with application programs, and Chapter 6 will describe our experience with kernel code. In this chapter, Section 5.1 describes our experiment with eight open source packages with a total over one million lines of code, where we found 13 bugs and weaknesses. Section 5.2 describes our group effort to run MOPS on all the packages in RedHat Linux 9 containing over 60 million lines of code, where we found 79 bugs and weaknesses. These experiments demonstrate that MOPS has achieved its goal of becoming a practical tool for check large, real world programs.

5.1 Experiments with one million lines of code

5.1.1 Security properties

We selected five important security properties that are non-trivial and that have been violated repeatedly in the past. We will show how we expressed them in a formal language suitable for input to MOPS.

Drop privileges properly

Access control in Unix systems is mostly based on user IDs (*uid*s). On most Unix systems, each process has three user IDs: the real user ID (*ruid*), the effective user ID (*euid*), and the saved user ID (*suid*). The real uid identifies the owner of the process, the effective uid represents the privilege of the process such as its permission to access the file system, and the saved uid is used by the process to swap between a privileged *uid* and an unprivileged one. Different user ID values carry different privileges. The *uid* 0, reserved for the superuser *root*, carries full privileges: it gives the process complete control over the machine. Some non-zero user IDs carry privileges as well. For example, the user ID *daemon* grants the process the privilege of accessing the spools used by *atd*. Most user processes have no privileges[1]. However, a class of programs called *setuid* programs allow the user to run a process with extra privileges. For example, *passwd* is a program that allows a user to change his password, so the program needs extra privilege to write to the password file. This program is setuid-*root*, which means that when a user runs the program, the real uid of the process is still the user, but both the effective and saved uid of the process are *root*, a privileged user ID.

A process modifies its user IDs by a set of system calls, such as `setuid`, `seteuid`,

[1]Since each process always has the privilege of its owner, this privilege is uninteresting from the security point of view and will not be considered as a special privilege further on.

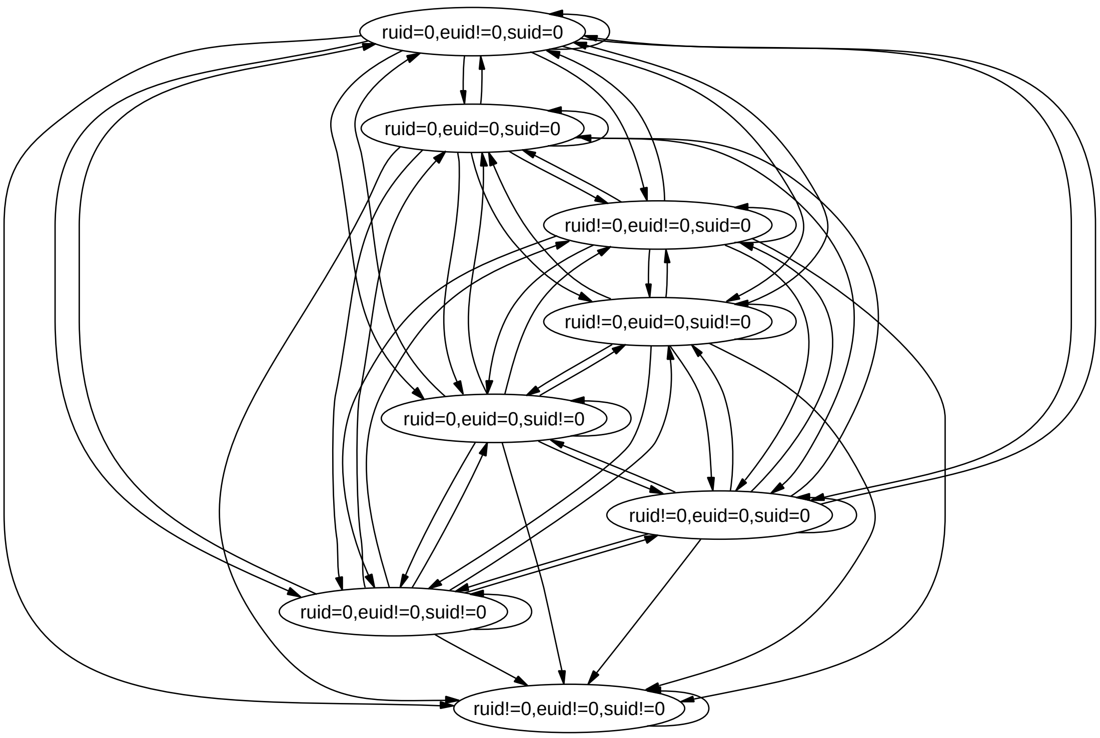
setreuid, and setresuid. By the principle of least privilege, if the process starts with a privileged user ID, it should drop the privilege permanently — by removing the privileged user ID from its real uid, effective uid, and saved uid — as soon as it no longer needs the privilege. Otherwise, if a malicious user takes control over the process, e.g. by a buffer overrun, he can restore the privileged user ID into the effective uid and thus regain the privilege. Section 4 discussed this topic in details.

Since where to drop privilege permanently is application specific, it is difficult for an automated tool like MOPS to check this property automatically without knowing the design of each application. Nevertheless, a process should permanently drop privileges before making certain system calls, such as execl, popen, and system, unless the process has verified that the arguments to these calls are safe. So we check the following property, which is a good approximation of the desired behavior (see Figure 5.1):

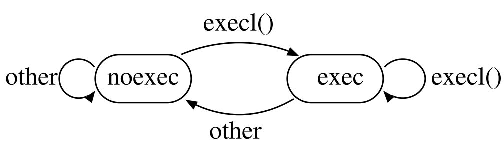
Property 1 *A process should drop privilege from all its user IDs before calling execl, popen, system, or any of their relatives.*

We decompose this property into two FSAs. The first one describes which user IDs of the process carry privilege (Figure 5.1(a)) and the second one describes whether the process has called the execl, popen, and system (Figure 5.1(b)[2]). Decomposition makes each FSA simpler and also allows the user to reuse FSAs (if the user wants to describe another property that involves privilege, he can reuse the FSA in Figure 5.1(a)). MOPS automatically computes the the parallel composition of the two FSAs, which represents the integrated property.

[2]Although the transition from the state "*after exec*" to the state "*before exec*" is optional for this property, it enables MOPS to find all the risky system calls if a trace contains multiple such calls; without this transition MOPS can only find the first call on the trace.



(a) An FSA describing which user IDs carry the *root* privilege. For clarity, we do not show the labels on the transitions.



(b) An FSA describing whether the process has called any of `execl`, `execv`, `system`, `popen`, etc.

Figure 5.1: Two FSAs describing Property 1 ("*A process should drop privilege from all its user IDs before calling* `execl`*,* `execv`*,* `popen`*, or* `system`"). In these FSAs the privileged user ID is *root* and the unprivileged user ID is non-*root*. In other situations, however, we may need to use an alternative FSA where the privileged user ID is also non-*root*.

Create chroot jails securely

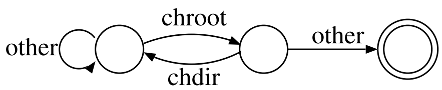
`chroot` is a system call that allows a process to confine itself to a sub-filesystem, called a *chroot jail*. To create a jail securely, the process should observe the following property (see Figure 5.2(a)):

Property 2 *After calling* `chroot`*, a process should immediately call* `chdir("/")` *to change its working directory to the root of the sub-filesystem.*

The program in Figure 5.2(b) violates this property because 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 2 encodes a safe style of programming that avoids some of these traps.

Avoid race conditions when accessing the file system

When a process accesses a file by its name, the process should avoid race conditions, which are potential security vulnerabilities [6]. As an example, consider a privileged process that



(a) An FSA describing Property 2

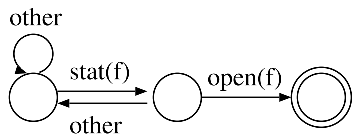
```
chroot("/var/ftp/pub");
filename = read_from_network();
fd = open(filename, O_RDONLY);
```

(b) A program segment violating Property 2. Note that the program fails to call `chdir("/")` after `chroot()`, so if `filename` is *"../../etc/passwd"*, a security violation ensues.

Figure 5.2: An FSA illustrating Property 2 ("*`chroot()` must always be immediately followed by `chdir("/")`*") and a program violating it.

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 might use 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 5.3(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 race conditions in privileged processes may be exploited by an adversary to gain control over the system. Moreover, race conditions in unprivileged processes may also be exploited by an adversary to penetrate the account of the user that runs the vulnerable processes. A



(a) An FSA describing Property 3

```
stat(logfile, &st);
if (st.st_uid != getuid())
    return -1;
open(logfile, O_RDWR);
```

(b) A program segment violating Property 3. 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 5.3: An FSA illustrating Property 3 ("*A program should not pass the same file name to two system calls on any path*") and a program violating it.

conservative approach for detecting race conditions is to check if the program passes the same file name to two system calls on any path, which we encode in Property 3:

Property 3 *A program should not pass the same file name to two system calls on any path*[3].

The system calls in Property 3 include: `chdir`, `chmod`, `chroot`, `creat`, `execve`, `lchown`, `link`, `lstat`, `mkdir`, `mknod`, `mount`, `open`, `pivot_root`, `quotactl`, `readlink`, `rename`, `rmdir`, `stat`, `statfs`, `symlink`, `truncate`, `umount`, `unlink`, `uselib`, `utime`, `utimes`.

Avoid attacks on standard file descriptors

Normally when a Unix process is created, it has three open file descriptors: 0 for standard input (*stdin*), 1 for standard output (*stdout*), and 2 for standard error (*stderr*). Many C programs and some library functions, such as `perror`, write error messages to *stderr*, assuming that *stderr*

[3]We could make this property stronger by including library functions that take file names as arguments.

```
// This is a setuid-root program
fd = open("/etc/passwd");
str = read_from_user();
fprintf(stderr, "The user entered:\n%s\n", str);
```

(a) victim.c: a program vulnerable to the *stderr* attack

```
int main()
{
    close(2);
    execl("victim", "victim", NULL);
}
```

(b) A program run by the adversary to attack the program in Figure 5.4(a)

Figure 5.4: A program vulnerable to the attack on standard file descriptors and an exploiting program.

is opened to a terminal (*tty*). If, however, *stderr* is opened to a sensitive file, then the messages will be written to the sensitive file, damaging the file or even allowing an adversary to take control over the system. For example, suppose the victim program in Figure 5.4(a) is installed setuid-*root*. The adversary runs the attack program in Figure 5.4(b), which closes its *stderr* and then executes the victim program. Then, when the victim program opens the password file */etc/passwd*, because Unix opens a file to the smallest closed file descriptor, the file will be opened to file descriptor 2, i.e., *stderr*. Later, when the victim program writes an error message to *stderr*, the message enters */etc/passwd*. Since the message contains a string coming from the adversary, the adversary can choose the string to be a valid entry in the password file that allows him to log in as *root*.

The avoid this attack, a prudent program should observe the following property:

Property 4 *Do not open a file in writing mode to* stdout *or* stderr*, no matter which file descriptors* *are open when the process starts*[4].

A popular defense to this vulnerability is to open */dev/null* three times at the beginning of the program so that no files can be opened to *stderr*.

Create temporary files securely

Many programs create temporary files in a shared directory such as */tmp*. The C library provides several functions for creating unique temporary files. Unfortunately, most of them are insecure because they make the program vulnerable to race condition attacks. For example, `mktemp` returns a unique file name, but if an adversary creates a file with the same name before the program does, the program will either open the adversary's file, if the `open` call does not specify the `O_EXCL` flag, or fail otherwise. Other insecure functions for making unique temporary files are `tmpnam`, `tempnam`, and `tmpfile`. The only secure function is `mkstemp`, which accepts a string parameter as the template for the temporary file, opens a unique file, and then returns the file descriptor. Additionally,

- To avoid race conditions, the program should not reuse the string parameter to `mkstemp` for any other system calls or library functions. This is because `mkstemp` writes the name of the unique temporary file to the string, but the binding from name to inode might change after `mkstemp` returns and before the next function executes.
- The program should call `umask(077)`[5] before calling `mkstemp`, because old versions of `mkstemp` create temporary files with the mode 0666, which is a security risk because all users can read the files.

[4] We could make the property stronger by ensuring that the program does not open any file in reading mode to *stdin*. We have not experimented with this modification.

[5] Here we are using the C convention that a leading 0 denotes an octal number.

Program	LOC	Description
Apache 2.0.40-21	229K	An HTTP server
At 3.1.8-33	6K	A program to queue jobs for later execution
BIND 9.2.1-16	279K	An implementation of the Domain Name System (DNS)
OpenSSH 3.5p1-6	59K	A client and a server for logging into a remote machine via the SSH protocol
Postfix 1.1.11-11	94K	A security-oriented Mail Transport Agent
Samba	254K	A Windows SMB/CIFS file server and client for Unix
Sendmail 8.12.8-4	222K	A popular Mail Transport Agent
VixieCron 3.0.1-74	4K	A program to maintain crontab files for individual user

Figure 5.5: Software packages that we checked using MOPS

We summarize the security property as the following:

Property 5 *A program should (1) never call* `mktemp`*,* `tmpnam`*,* `tempnam`*, or* `tmpfile`*; (2) never reuse the parameter* `x` *in* `mkstemp(x)`*; and (3) call* `umask(077)` *before* `mkstemp`*.*

5.1.2 Software programs

We selected six large, security-sensitive packages and two small setuid-*root* programs from Redhat Linux 9. See Figure 5.5 for a list of software packages we analyzed, their descriptions, and the number of lines of code analyzed (counted using "`wc -l *.c`").

5.1.3 Performance

We ran all the experiments on an 1.5GHz Pentium 4 single-CPU PC with 1GB memory that runs Redhat Linux 9. Figure 5.6 shows the time that MOPS spent on model checking each package and the number of real error traces and total error traces that MOPS found. Each table shows the results for one property, where each row shows the results for one package, including the number of programs checked, the running time, and the number of real error traces vs. total error traces. The following explains how we measured these results.

- The number of programs checked. Each package builds multiple executable programs, but some properties only apply to a subset of these programs. Property 1 and 2 only apply to either programs that are run by the *root* user or setuid/setgid programs, because the OS only allows these programs to change their user IDs or group IDs (Property 1) and to create root jails (Property 2). Property 4 applies only to setuid/setgid programs because an unprivileged adversary can only exploit the violation of this property by running vulnerable setuid/setgid programs, which grant him extra privilege. Property 3 and 5 apply to all the programs in each package.

- The running time. We calculated the running time by summing the user time and the system time as reported by the `time` command.

- The number of real error traces and total error traces. For each package, after collecting the error traces that MOPS reported on each program, we merged all the equivalent traces. We consider two traces from two different programs equivalent if both make transitions to an identical error state from an identical non-error state at an identical program point. This is similar to how we decide two equivalent traces within a program (Section 2.2.6). Then, we examine each trace manually to determine if it indicates a real bug. A real bug means that under certain conditions an adversary may exploit the bug to take control over the system or the user's account, to gain privileges, or to cause the program to fail. A trace reported by MOPS may not indicate a real bug due to the following reasons:

  - The trace is infeasible due to MOPS's imprecise program analysis.

  - Because the property is conservative, the trace does not indicate a bug even though it violates the property. For example, Property 1 requires that a privileged process should not call *execv* with untrusted argument. Since we cannot determine if an argument is trusted statically, we code the property to forbid a privileged process from calling *execv* entirely. Therefore, even if a privileged process calls *execv* with an trusted argument, MOPS still considers it a violation of the security property and provides an error trace; it is up to the human to notice that this error trace is not a real bug.

Therefore, the 5th column in each table counts only real bugs. The 6th column counts the total number of all error traces (whether they represent real bugs or not).

5.1.4 Findings

In this section we describe several security weaknesses in the programs that we found during our experiments.

Failure to drop privilege completely

In the past several programs have had vulnerabilities because they failed to drop privilege securely. This is partly due to the poor design of the system calls that set user IDs — they have confusing semantics, different behaviors on different platforms, and insufficient or even wrong documentation (Chapter 4). As a result, they are susceptible to misuse, which has caused numerous security vulnerabilities.

***ssh* in OpenSSH** *ssh* is a client that allows a user to log into a server via the SSH protocol. *ssh* is installed as a setuid-*root* program on some systems[6]. *ssh* may need to execute another program,

[6] *ssh* needs *root* privilege to read the local host key and to generate the digital signature required during the host-based authentication with SSH protocol version 2. A site can either install *ssh* setuid-*root* or configure it to use a setuid-*root* helper, *ssh-keysign*.

Package	LOC	Programs	Time (m:s)	Error Traces	
				Real	Total
Apache	229K	2	:45	1	4
At	6K	2	:05	0	0
BIND	279K	1	:53	0	1
OpenSSH	59K	3	:23	2	8
Postfix	94K	3	:17	0	2
Samba	254K	3	1:53	0	5
Sendmail	222K	1	:12	0	0
VixieCron	4K	2	:05	0	0

(a) Performance of MOPS on Property 1: "*A process should drop privilege from all its user IDs before calling* `execl`*,* `popen`*,* `system`*, or any of their relatives.*"

Package	LOC	Programs	Time (m:s)	Error Traces	
				Real	Total
Apache	229K	2	:09	0	0
At	6K	2	:05	0	0
BIND	279K	1	:03	0	0
OpenSSH	59K	3	:17	0	0
Postfix	94K	3	:12	0	0
Samba	254K	3	:56	0	0
Sendmail	222K	1	:22	0	0
VixieCron	4K	2	:05	0	0

(b) Performance of MOPS on Property 2: "*After calling* `chroot`*, a process should immediately call* `chdir("/")` *to change its working directory to the root of the sub-filesystem.*"

Package	LOC	Programs	Time (m:s)	Error Traces	
				Real	Total
Apache	229K	14	:43	0	1
At	6K	2	:06	0	6
BIND	279K	30	1:08	0	3
OpenSSH	59K	13	:50	0	12
Postfix	94K	33	2:18	0	3
Samba	254K	25	13:14	1	2
Sendmail	222K	24	1:53	0	8
VixieCron	4K	2	:07	1	2

(c) Performance of MOPS on Property 3: "*Avoid race conditions in file system access.*"

Package	LOC	Programs	Time (m:s)	Error Traces	
				Real	Total
Apache	229K	1	:14	1	1
At	6K	1	:04	1	1
BIND	279K	0	:00	0	0
OpenSSH	59K	2	:58	1	2
Postfix	94K	2	:46	0	1
Samba	254K	1	:52	1	1
Sendmail	222K	1	14:12	0	3
VixieCron	4K	1	:04	2	2

(d) Performance of MOPS on Property 4: "*Do not open a file in writing mode to* stdout *or* stderr*, no matter which file descriptors are open when the process starts.*"

Package	LOC	Programs	Time (m:s)	Error Traces	
				Real	Total
Apache	229K	14	:42	0	0
At	6K	2	:05	0	0
BIND	279K	30	1:11	0	0
OpenSSH	59K	13	1:01	2	2
Postfix	94K	33	3:20	0	0
Samba	254K	25	28:38	0	0
Sendmail	222K	24	1:55	0	0
VixieCron	4K	2	:06	0	0

(e) Performance of MOPS on Property 5: "*A program should: (1) never call* `mktemp`*,* `tmpnam`*,* `tempnam`*, or* `tmpfile`*; (2) never reuse the parameter* `x` *in* `mkstemp(x)`*; (3) call* `umask(0077)` *before* `mkstemp`*.*"

Figure 5.6: Running time of MOPS and number of error traces reported by MOPS on five properties

which is *ssh-askpass* by default but is user selectable, to read the passphrase of the user's private key. Since this program is not trusted, *ssh* needs to drop its *root* privilege permanently before executing the program. The code is "`seteuid(getuid()); setuid(getuid());`". This code fulfills the objective on BSD where OpenSSH was originally developed, but behaves unexpectedly on Linux. This is because the semantics of `setuid` differs between BSD and Linux. On BSD `setuid(new_uid)` sets all the real uid, effective uid, and saved uid to `new_uid`. On Linux, however, the behavior of this call depends on whether the effective uid is *root*: if so, the call sets all three user IDs to `new_uid`; otherwise, the call sets only the real and effective uid to `new_uid` but leaves the saved uid unchanged. Before *ssh* executes the above code, its real uid is non-*root* and its effective uid and saved uid are *root*. The first call, `seteuid(getuid())`, sets the effective uid to non-*root*. Therefore, the outcome of the second call, `setuid(getuid())`, depends on the OS. On BSD the call sets the saved uid to non-*root*, but on Linux the call keeps the *root* saved uid unchanged.

This weakness suggests that the programmer misunderstands how the `setuid`-like system calls work. In fact, if we remove the first call, *ssh* would behave as desired on both Linux and BSD. The extra `seteuid (getuid())` was introduced in recent versions of OpenSSH (an old version, 2.5.2, does not have it) and the programmers seem to think that it makes the program safer, but in fact it introduces a weakness. It would be very easy to overlook this subtle weakness in a manual audit, which demonstrates the utility of MOPS.

Although leaving a privileged user ID in the saved uid before executing an untrusted program does not result in an immediate exploit by itself (because the OS will set the saved uid to the unprivileged effective uid before executing the program), it is an indication of weakness in the program because the programmers have likely intended to drop privilege permanently. This may cause two problems. First, since the programmers think that they have permanently dropped privilege, they may freely do certain actions that are safe without privilege but risky with privilege. Second, if an adversary causes a buffer overrun in the program, he may inject code into the program to regain privilege in the saved uid (by calling `seteuid(saved_uid)`). The more code that runs after the failed attempt to drop privilege permanently, the more potential threat the program faces.

***ssh-keysign* in OpenSSH** *ssh-keysign* is a setuid-*root* program that accesses the local host keys and generates a digital signature. It starts with *root* privilege in its effective uid and saved uid because it needs it to access the local host keys, which are accessible only to *root*. After *ssh-keysign* opens the host key files, it intends to drop *root* privilege permanently before doing complicated cryptographic operations. Unfortunately, it fails to drop *root* privilege from the saved uid on Linux because it calls "`seteuid(getuid()); setuid(getuid());`", like *ssh* as discussed earlier. Therefore, the *ssh-keysign* process will execute complicated, possibly third-party cryptographic code with *root* privilege in the saved uid. If an adversary can cause a buffer overrun in the code, as happened to the RSAREF2 library in the past [49], he may take control of the process and then regain privilege.

***suexec* in Apache** *suexec* is a setuid-*root* program that executes another program using a given user ID and group ID. The calling convention is as follows.

```
suexec uid gid program args
```

An option in the Apache web server *httpd* lets the server execute CGI programs as the program owner (e.g. using the owner's user ID and group ID). Since *httpd* runs as the user *apache*, it cannot run any program as another user, so it asks *suexec* to do it. This option, however, is turned off by default, therefore by default *httpd* executes CGI programs as the user *apache*. To prevent non-*root* users from running *suexec* directly, *suexec* is executable by the user *root* and the group *apache* but not by anyone else.

However, a local adversary on the web server can circumvent this protection by running *suexec* from his CGI program because the CGI program runs as the group *apache* and so can run *suexec*. This seems very dangerous because now the adversary can run any program as any user. Fortunately, to ensure that it will not do any harm to the system, *suexec* performs many security checks, including:

- It checks that the current directory is within DOCROOT (e.g. public_html/cgi-bin).
- It checks that the requested user and group IDs own the current directory.
- It checks that the requested user and group IDs own the requested program.
- It checks that the last component in the path to the requested program is not a symbolic link. (However, it does not check if a directory name in the path is a symbolic link.)
- It checks that the command does not start with `/` or `../` and does not contain `/../`, to prevent the command from escaping DOCROOT.

Notwithstanding these paranoid checks, the adversary can still attack the system in at least the following two cases:

- For any target user on the system, the adversary can execute any program that is owned by the victim and is in a subdirectory of the victim's CGI directory. This may not seem a risk because the victim should assume that any program under his CGI directory can be invoked at any time by a web user. However, the threat is that this may break the victim's expectations in two different ways, as listed below, and this could lead to a security violation.

- First, the victim can no longer prevent other users from executing the programs in his CGI directory by setting file permission appropriately. For example, when the victim is experimenting with a CGI program that has not been audited for security, he may want to prevent other users from running the program by setting the file not world executable, but he might not realize that an adversary can still run his program via *suexec*.
- Second, the victim can no longer expect the command line arguments to his CGI program to come from the web server and therefore to be well-formed, because the adversary can pass arbitrary arguments to the CGI program via *suexec*. As the victim may not expect this, he may not have written his CGI programs to defend against malicious data in command line arguments.

This becomes more serious if the victim creates a symbolic link from his CGI directory to his home directory, as now the adversary can run every program in his home directory with command line arguments of his choice, and these programs runs with the victim's privileges. This may allow the adversary to gain control of the victim's account.

- On systems where a non-*root* user can change the ownership of his files, such as some systems derived from System V Unix, the attack becomes more severe. The adversary can create a directory within his DOCROOT, create a malicious program inside it, and change the ownership of both to the victim. Then the adversary lets *suexec* (via his CGI program) to run the malicious program as the victim.

Discovering both these security issues in *suexec* required some manual analysis, but MOPS pointed us to the right place to start and helped find “the needle in the haystack”. This example shows the value of looking for suspicious code, even when that code is not previously known for certain to be a security hole.

Standard file descriptor vulnerabilities

***at* in At** *at* is a setuid-*root* program. It reads the commands that the user wants to execute at a certain time from the standard input and writes them into a file. Later, the daemon *atd* executes the file. The file is placed in a directory that ordinary users cannot read from or write to.

at does not take care to ensure that the first three file descriptors are open before it opens any file for writing. Because the invoker controls the initial bindings of all file descriptors, an adversary can cause *at* to open a file in writing mode to *stderr*. For example, the adversary could close *stdout* and *stderr* and then invoke *at*. *at* first opens a lock file, which will be assigned to *stdout*, and then opens a task file to record the user’s commands, which will be assigned to *stderr*. From now on, all messages for *stderr* will enter the task file.

We are unaware of any way to exploit this bug. Though it is possible for an attacker to corrupt the task file, the attacker has little control over the contents written into the task file, since *at* only writes prompts to *stderr*. Nonetheless, this blemish may cause serious problems if future versions of *at* write user-supplied strings to *stderr*.

We also found similar problems in *Apache*, *OpenSSH*, *Samba*, and *VixieCron*.

Temporary file vulnerability

***sshd* in OpenSSH** *sshd* makes temporary files using the secure function `mkstemp`. It, however, forgets to call `umask(077)` before calling `mkstemp`. This may introduce a vulnerability when used with older versions of glibc, where `mkstemp` creates unique temporary files that are readable and writable by everyone (mode 0666).

5.2 Experiments with an entire RedHat Linux distribution

This section describes our experiments on using MOPS to check the entire RedHat Linux 9 distribution. This is joint work with Benjamin Schwarz, David Wagner, Geoff Morrison, Jacob West, Jeremy Lin, and Wei Tu. The text in this section is extracted from a draft paper jointly written by this group of people.

RedHat Linux 9 contains 839 packages totaling 60.0 million lines of code (counting total lines in all `.h`, `.c`, and `.cc` files). MOPS successfully analyzed 87% (732) of these packages; the remaining 107 packages could not be analyzed because MOPS's parser cannot parse some files in these packages[7]. To the best of our knowledge, our experiment is by far the largest reported security auditing of software using automated tools.

5.2.1 Properties and findings

We developed six security properties. Table 5.1 shows a summary of the bugs discovered. For each property, the table shows the number of warnings reported by MOPS and the number of real bugs. We will describe four properties in detail, explain the model checking results, and show

[7]The parser failed on 73 packages because they contain C++ code (the parser currently parses only C programs, although we are developing a new C++ parser), and on 34 packages because they contain certain C99 constructs.

Property	*Reported Warnings*	*Real Bugs*
TOCTTOU	790	41
Standard File Descriptors	30	0
Temporary Files	108	34
`strncpy`	668	>4
Chroot Jails	1	0
Format String	0	0
Total	1597	79

Table 5.1: Overview of Results

representative bugs and vulnerabilities that we discovered in this section.

Red Hat Linux 9 consists of 839 packages with 60.0 million lines of code, the majority of which are integer-style programs written in C. We ran our experiments an Intel IA-64 Itanium machine with dual 800 MHz processors. MOPS does not require any special features from the machine's 64-bit architecture; on the contrary, the machine's performance is significantly lower than typical 32-bit desktop machines. Although the machine has plenty of physical memory, MOPS typically uses less than 1GB.

TOCTTOU

Race condition attacks have perennially plagued secure systems. One common mistake is that a program checks the access permission of an object and, if the check succeeds, makes a privileged system call on the object. For example, one notorious error involves the `access()` and `open()` system calls. Consider a program running as root (e.g., setuid root, or executed by root) executing the following code fragment:

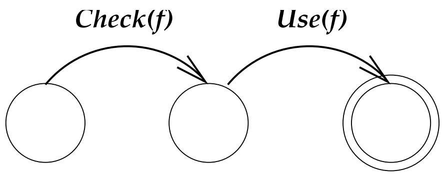
```
if (access(pathname, R_OK) == 0)
  fd = open(pathname, O_RDONLY);
```

The kernel does not execute this sequence of system calls atomically, so if there is a context switch between the two calls or if the program is running on a multiprocessor system, another program may change the permission of the object. When the above program resumes its execution, it then blindly performs `open()` even though the user should no longer have access permission to the object.

Another example comes from UNIX folklore. It is well known that the root user should not recursively remove files inside directories that may be writable by other users. For example, "`rm -rf /tmp`" is a dangerous command, even if root has verified that the directory */tmp* contains no symlink to other parts of the file system. The reason is that after `rm` verifies that a directory is not a symlink but before it enters the directory to delete the files within, an adversary may replace the directory with a symlink to another part of the file system, therefore tricking `rm` into deleting that part of the file system.

Many of the vulnerabilities that we found are exploitable when two users share access to some portion of the file system, and one user is operating on the shared portion while the other mounts an attack by replacing symbolic links. Although programs commonly attempt to ensure that they do not follow symbolic links before doing dangerous operations like *unlink*, they often check it incorrectly and are susceptible to TOCTTOU attacks as a result.

We have experimented with several different FSAs to capture these types of vulnerabilities. In our first attempt, we chose an FSA that had two transitions—one from the start state to an intermediate state, and the other from the intermediate state to the accepting state. Both transitions were defined on the union of the file system calls that access the file system using a pathname argument. However, we found that this naive property results in too many false positives—for example, `chdir(".")` followed by a `chdir(".")` would trigger a false positive. Typically these situa-



Check(f) = {`stat(f), lstat(f), access(f), readlink(f), statfs(f)`}.
Use(f) = {`basename(f), bindtextdomain(f), catopen(f), chown(f), dirname(f), dlopen(f), freopen(f), ftw(f), mkfifo(f), nftw(f), opendir(f), pathconf(f), realpath(f), setmntent(f), utmpname(f), chdir(f), chmod(f), chroot(f), creat(f), execv(f), execve(f), execl(f), execlp(f), execvp(f), execle(f), lchown(f), mkdir(f), fopen(f), remove(f), tempnam(f), mknod(f), open(f), quotactl(f), rmdir(f), truncate(f), umount(f), unlink(f), uselib(f), utime(f), utimes(f), link(f), mount(f), rename(f), symlink(f)`}.
Non-filename arguments are omitted.

Figure 5.7: A refined FSA for finding TOCTTOU (file system race condition) vulnerabilities.

tions arise when a single system call is located inside of a loop, and both transitions in the FSA are made as a result of executing the same line of code. Since these are obviously not security holes, we decided to separate out the file system calls that can be classified as “checks” from those that are “uses”.

We refined the property by dividing the file system calls into two distinct sets. Figure 5.7 shows the general structure of the FSA. We assume here, and in subsequent illustrations, that there is an implicit *other* transition from every state back to itself; if none of the normal outgoing transitions match, the *other* transition is taken, and the FSA stays in the same state. The intuition is as follows: a call to *Check(f)* is probably intended to establish an invariant (e.g., “*f* is not a symlink”), and a call to *Use(f)* might rely on this invariant. Of course, these invariants might be violated in an attack, so *Check(f)* followed by *Use(f)* may indicate a TOCTTOU vulnerability. This refined property is much more manageable and finds most of the bugs we are interested in. However, the more general property is capable of finding some bugs which the narrower TOCTTOU cannot—for example, `creat(f)` followed by `chmod(f)`.

The types of vulnerabilities we have found can be classified under the following categories:

1. [*Access Checks*] A check is performed—often by a program running as root—on file access permissions. The result of the check is then used to determine whether a resource can be used. The `access(f)` and `open(f)` race at the beginning of this section illustrates this class of bugs.

2. [*Ownership Stealing*] A program may `stat()` a file to make sure it does not exist, then open the file for writing. If the *exclusive* flag is not used, an attacker may create the file after the `stat()` is performed, and the program will then write to a file owned by the attacker. We consider this a vulnerability, because the program may disclose sensitive information.

3. [*Symlinks*] Vulnerabilities due to symbolic links arise when two users share the same portion of the file system. One user can change a file to a symlink to trick the other user to mistakenly operate on an unexpected file. The method of such an attack depends on whether the system call follows symlinks. Broadly, there are two classes of system calls:

   (a) [*Syscalls that follow symlinks*] Many system calls will follow symbolic links that occur anywhere in the pathname passed to them. These present no barrier to attack.

   (b) [*Syscalls that don't follow symlinks*] Other system calls avoid following symbolic links if they occur in the last component of their pathname argument. For instance, if `c` is a symbolic link to `d`, calling `unlink("/a/b/c")` will delete the symbolic link itself rather than the target of the link: it deletes `/a/b/c`, not `/a/b/d`. However, many programmers do not realize that these calls will gladly follow any symlinks that occur in earlier components of the pathname. For example, if `b` is a symlink to `../etc`, then `unlink("/a/b/passwd")` will delete the password file `/etc/passwd`. Consequently, to attack this second class of system calls, it suffices for the attacker to tamper with one of the intermediate components of the path.

Table 5.3 in Appendix 5.2.2 shows all the TOCTTOU bugs we discovered. The third column shows the number of traces that MOPS thought violated the security property, and the last column shows the number of those traces that were actual bugs. We successfully mounted attacks on several programs to verify that the bugs were real. Many bugs we found were not previously known. Some were previously reported (but apparently not yet fixed and not known to us at the time of our experiments). To illustrate the kinds of bugs we found with MOPS, we will show three representative examples of TOCTTOU bugs.

```
# binutils-2.13.90.0.18-9 :: ar #
exists = lstat (to, &s) == 0;
/* Use rename only if TO is not a symbolic link and has only one hard link.  */
if (! exists || (!S_ISLNK (s.st_mode)
  && s.st_nlink == 1)) {
  ret = rename (from, to);
  if (ret == 0) {
    if (exists) {
      chmod (to, s.st_mode & 0777);
      if (chown (to, s.st_uid,
           s.st_gid) >= 0) {
        chmod (to, s.st_mode & 07777);
      }
  ...
```

In our first example, the program *ar* executes the code fragment above to replace an archive file with one of the same name. It calls `lstat` on the destination file and then checks if the file is a symbolic link. If it is not, *ar* calls `rename` on the file and then sets the mode of the file. This code, however, is unsafe. An adversary may change the file to be symbolic link after *ar* checks for symbolic links. Then, *ar* will happily change the mode of whatever the symbolic link points to—assuming the user running *ar* has permission to do so. The attack is applicable when two users have write access to the directory of the archive file.

```
# initscripts-7.14-1 :: minilogd #
/* Get stat info on /dev/log so we can later check to make
   sure we still own it... */
if (stat(_PATH_LOG,&s1) != 0) {
   memset(&s1, '\0', sizeof(struct stat));
}
...
if ( (stat(_PATH_LOG,&s2)!=0) || (s1.st_ino != s2.st_ino ) ||
      (s1.st_ctime != s2.st_ctime) ||
      (s1.st_mtime != s2.st_mtime) ||
      (s1.st_atime != s2.st_atime) ) {
         done = 1;
         we_own_log = 0;
}
/* If we own the log, unlink it before trying to free our buffer.
   Otherwise, sending the buffer to /dev/log doesn't make much
   sense ... */
if (we_own_log) {
   perror("wol");
   unlink(_PATH_LOG);
}
```

The second code fragment is taken from the program *minilogd*, which is run by root. This program may unlink `_PATH_LOG` (which is defined to be `/dev/log` by default) if it thinks it exclusively owns the file. It compares the timestamps on the file at two different times in the execution of the program and, if they equal, decides that it exclusively owns the file and then removes the file. However, even if another program modifies the file after *minilogd* checks the timestampes, *minilogd* will still unlink it, possibly corrupting other programs. An additional vulnerability exists when user programs can write to the log file; for instance, if `_PATH_LOG` is defined as something like `/home/alice/log` instead. In this case, Alice can trick *minilogd* into removing anything on the file system. We have found many vulnerabilities that are similar to the latter case. It is important that these filename constants be checked very carefully when these programs are built, since it may not be obvious to users that defining `_PATH_LOG` to a user-writable file can result in a total compromise of the file system.

```
# zip-2.3-16 :: zip #
  d_exists = (LSTAT(d, &t) == 0);
  if (d_exists) {
    /* respect existing soft and hard links! */
    if (t.st_nlink > 1 || (t.st_mode&S_IFMT)==S_IFLNK )
       copy = 1;
    else if (unlink(d))
       return ZE_CREAT;
  }
```

The final code snippet comes from the widely-used program *zip*. If the destination file already exists, *zip* will move the file to a new location, unlink the old copy, and write the new copy. The program verifies that the file is not a link before calling `unlink` on it. The attack is applicable when two users share a portion of the file system and one user is running *zip* to write a new file to the shared space. If the other user is malicious, after *zip* calls `stat`, the user can change the file to be a symbolic link that points to another part of the file system. Since `unlink` will not follow the last component of a pathname, the attacker would have to change one of the components in the middle of the pathname to a symbolic link. For instance, if Alice is using *zip* to write a file to `/shared/alice/afile`[8], Bob can change `/shared/alice` to be a symbolic link that points to `/home/alice`. Then the *zip* program running on behalf of Alice will remove `/home/alice/afile`. Most users will not be aware that using a shared directory enables such attacks, so it seems unfair to blame Alice for doing so. In this case, *zip* does try to do the right thing

[8]Also, the suggested scenario requires that the sticky bit is not set. The sticky bit prevents deletion of files and directories for anyone except the creator, even if others have write access. Generally, `/tmp` has the sticky bit set.

by checking for symbolic links; it just happens to get the check wrong.

The best way we know to avoid TOCTTOU errors is to `stat` a file, then perform access checks or whatever else needs to be done, then open the file, then `fstat` the resulting file descriptor to make sure it is as expected. Note that this still allows an adversary to open an arbitrary file, and this may have undesirable effects—such as causing a tape drive to rewind. However, it is unclear how to correctly `unlink` files from the file system in this way. Alternatively, one possible solution would be to use the probabilistic solution proposed by Dean and Hu [16].

A Standard File Descriptor Attack

The first three file descriptors of each Unix process are called *standard file descriptors*: FD 0 for the standard input (*stdin*), FD 1 for the standard output (*stdout*), and FD 2 for the standard error (*stderr*). Several commonly used C standard library functions read from or write to these standard file descriptors; e.g., `fgets()` reads from *stdin*, `printf()` writes to *stdout*, and `perror()` writes to *stderr*. Customarily, a program starts with its standard file descriptors opened to terminal devices. However, since the kernel does not enforce this convention, an attacker can force a standard file descriptor of a victim program to be opened to a sensitive file, so that he may discover confidential information from the sensitive file or modify the sensitive file.

For example, suppose a victim program is setuid-root[9] and executes the following code:

```
/* victim.c */
fd = open("/etc/passwd", O_RDWR);
if (!process_ok(argv[0])) { perror(argv[0]); }
```

Then the adversary can run the following attack program to exploit the standard file descriptor vulnerability in the victim program:

[9] A setuid-root program runs with root privileges, even if it is executed by an unprivileged user.

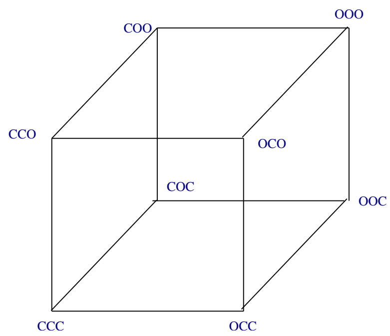


Figure 5.8: The structure of an FSA for finding file descriptor vulnerabilities. This FSA tracks the state of the three standard file descriptors across `open()` calls.

```
/* attack.c */
int main(void) {
  close(2);
  execl("victim", "foo:<pw>:0:1:Super-User-2:...", NULL);
}
```

This attack works as follows. First, the attack program closes its FD 2 and executes the victim program. Since a spawned process inherits its file descriptors from its parent process, the victim program starts with its FD 2 closed. Then, when the victim program opens the password file */etc/passwd*, the file is opened to FD 2, since Unix systems open a new file to the smallest available file descriptor. Later, when the victim program writes an error message by calling `perror`, which writes to FD 2, the message enters */etc/passwd*. Since the attacker has chosen the message to be a valid entry in the password file, he may now log in with the superuser privilege.

We designed a property that detects this type of attacks on standard file descriptors. Since each of the three standard file descriptors may be independently open or closed, the FSA contains eight states each of which describes a unique combination of the states of the three file descriptors (Figure 5.8). For example, the state `OCC` represents that FD 0 is open, but FD 1 and FD 2 are closed. The FSA may start in all the seven states where at least one of the three standard FDs is closed;

the case where all of the standard FDs are initially open is the usual one, and not of interest to an attacker. The FSA transitions only along the edges of the cube, as there are no system calls that can change the status of multiple standard file descriptors at once. Additionally, we add an accepting state (the error state, not shown in Figure 5.8) to the FSA .

The FSA transitions on three types of events:

- *UnsafeOpen*: any `open()` that opens an "unsafe" file in non-read-only mode. We consider any file that is neither */dev/null* nor */dev/tty* to be unsafe. When the FSA encounters an *UnsafeOpen*, it transitions to the error state, indicating that an unsafe file may have been opened on a standard FD.
- *SafeOpen*: any `open()` call that is not a member of *UnsafeOpen*. When the FSA encounters a *SafeOpen*, it transitions to the state representing the new open/closed configuration of the standard FDs. For example, if the current state is `COC` and a *SafeOpen* is encountered, then the new state is `OOC`, since the new file is opened to FD 0.
- *DupOrPipe*: any `dup()` or `dup2()` call where the new FD is a standard FD. Though the `open()` FSA is primarily intended to track `open()` calls, including these additional cases allows more accurate modeling of which standard FDs are open or closed. The transitions used here are straightforward. For example, if the current state is `OCC`, then encountering a `dup()` would result in state `OOC`, and a `dup2(x,2)`, state `OCO`.

We ran this property on all programs installed setuid or setgid on Red Hat 9. We found one potential vulnerability in the *gnuchess* program, which is installed setgid-games. In principle, this vulnerability could allow the shared gamebook to be corrupted; however, it appears not to be widely exploitable. One qualification is that it appears that the *gnuchess* binary being installed setgid-games is the result of Red Hat customizations, and is not the default for the original source distribution.

Secure Temporary Files

Applications often use temporary files as a means for passing data to another application, writing log information, or storing temporary data. Often times on a Unix system, the files will be created in the `/tmp` directory, which is world writable and readable. For example, the GNU C compiler creates temporary files when it is compiling programs, and later passes them to the linker. Many of the functions to create temporary files that are found in the C standard library are insecure. The reason is that they do not return a file descriptor, but rather a file name. An adversary that is able to predict the filename can thus create the file before the application has a chance to open or create it. This attack can give the adversary ownership of the temporary file, which is undesirable[10].

We identified the set of insecure functions: *mktemp*, *tmpnam*, *tempnam*, and *tmpfile*. These functions should never be used. We uphold the same principle in detecting misuses of the one function that can be used securely: *mkstemp*. Security requires that the filename retrieved from *mkstemp* is never subsequently used in another system call: *mkstemp* returns both a file descriptor and a filename, but a secure program should not use the filename. Figure 5.9 illustrates our automaton.

In some situations a program may use an insecure function but still not have a vulnerability. An example of such a situation is a program that uses *tempnam* to create a temporary filename in `/tmp`, then attempts to open that file using the UNIX `O_CREAT` and `O_EXCL` flags. Combined, the flags ensure that the file does not already exist. The program can then check the return value from

[10]In the *gcc* example, an adversary could insert malicious code into a user's program by replacing the temporary file with the desired code.

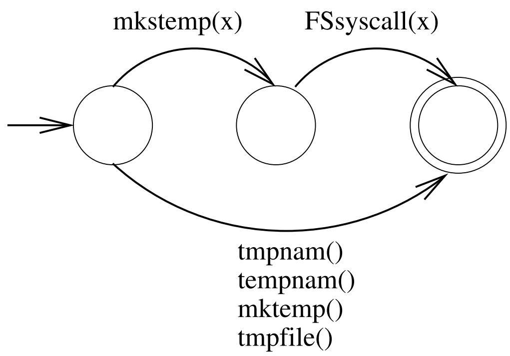


Figure 5.9: An FSA to detect insecure uses of temporary files.

open to verify that an attacker did not first create the file. It is possible to modify the simple FSA presented in Figure 5.9 to detect these situations and not report them as false positives, however it makes the FSA much larger. In practice, it seems prudent to manually examine these situations anyway. One can imagine examples in which a program uses the flags but does not take the correct action based on the return value from *open*.

Table 5.4 shows the applications that violated the property by using one of the insecure calls. Table 5.5 shows the subset of bugs due to inappropriate use of the filename returned by *mkstemp*. These tables may be found in Appendix 5.2.2.

Below we show a representative example of a program that violates the clause of our property that finds re-uses of the parameter passed to *mkstemp*. Not only is this the most complicated example presented thus far, but it shows how the whole-program inter-procedural analysis was effective. The code will be presented in several fragments as they occur temporally while executing the program. The example comes from the program *yacc* from the package `byacc-1.9-25`.

```
static void open_files() {
  int fd;
  create_file_names();
  if (input_file == 0) {
    input_file = fopen(input_file_name, "r");
    if (input_file == 0)
```

```
        open_error(input_file_name);
      fd = mkstemp(action_file_name);
      if (fd < 0 || (action_file = fdopen(fd, "w")) == NULL) {
        if (fd >= 0)
          close(fd);
        open_error(action_file_name);
      }
}
```

Before the above program fragment executes, there is some setup code that sets the value of the variable *action_file_name*. Specifically, it is a string whose first component is a pathname to a temporary directory (by default, it chooses `/tmp`, but this can be changed by defining an environment variable), and whose second component is a temporary file template[11]. The above code alone should be of concern to us. Recall that *mkstemp* returns a file descriptor that can be safely used, but the template passed to *mkstemp* is not safe to re-use. In this case, we see the template being passed to another function called *open_error*:

```
void open_error(char *filename) {
  warnx("f - cannot open \"%s\"", filename);
  done(2);
}
```

From the above fragment it looks like the function *warnx* may be a good candidate for inspection because it is the recipient of the filename we are interested in tracking. Strangely enough, MOPS directs us to the function *done*:

```
void done(int k) {
  if (action_file)
    fclose(action_file);
  if (action_file_name[0])
    unlink(action_file_name);
```

Here we find the bug. The variable *action_file_name*, which is the template passed to *mkstemp*, is re-used as an argument to the *unlink* system call. This is unsafe. By the time we call *unlink*, the

[11] A template is a partial filename with a number of placeholders denoted by a special character *X* that will be filled in with random numbers by the function creating the temporary filename.

filename may no longer point to the location we think it does. Recall that the directory in which the file is being created may be world writable. An attacker that has write access to the file can change it to a symbolic link, and cause the program to unlink other unexpected files on the system. Unfortunately there does not appear to be a good resolution to the problem.

In addition to these two types of vulnerabilities, we have also investigated a third class that can arise on systems with old versions of *glibc*. In previous versions, the library functions for creating temporary files automatically set the permissions on the file to be world writable and readable by default. Therefore, programs must use the *umask* system call to change default file permissions before creating temporary files; if they do not, they will be vulnerable to attack. We used MOPS to find programs that use temporary files and do not first issue a *umask* call. Formally, if there exists a path from the program entry to a function call that creates a temporary file, and no *umask* call exists on the path, then MOPS will report an error.

We found that the overwhelming majority (greater than 90%) of programs do not use *umask* and thus would be vulnerable if used with old versions of *glibc*. However, the severity of these bugs is very low, and they are large in number, so we have not enumerated them here.

Attacks Against `strncpy`

There are several common attacks against programs that misuse the standard library function `strncpy`. `strncpy(d,s,n)` copies a string of characters pointed to by *s* into the memory region designated by *d*. If *s* contains more than *n* characters, `strncpy` only copies the first *n* characters. If *s* contains fewer than *n* characters, `strncpy` copies all the characters in *s* and then fills *d* with null characters until the length of *d* reaches *n*.

`strncpy` is easy to misuse for two reasons. First, it encourages off-by-one errors if the programmer is not careful to compute the value of `n` precisely. Off-by-one errors can often cause the program to write past the end of an array bounds, which can in turn lead to buffer overrun attacks against the program. In particular, consider a case where the string buffer in question is allocated on the runtime stack (as it will be when the buffer is a an array local to a function in C), and the user of the program is able to control the contents of the source of `s`. If the program writes past the end of the buffer, a malicious user may construct a special string `s`, such that when the program writes past the array bounds, it writes special code into the stack frame that corrupts the program. Secondly, because the function does not automatically null-terminate a string in all cases (for instance, when the size of the source string is larger than `n`), it is a common mistake for a program to create unterminated strings during its execution.

We have constructed an FSA to try and catch both scenarios as described above. In the interest of space, and due to its complicated and non-intuitive structure, we have not presented a figure here. The intuition is that we identify several idioms that are correct ways to null terminate a string, and raise an alarm when one of these idioms is not used. For example, a common idiom is the following code sequence that is safe:

```
buf[sizeof(buf)-1] = '\0';  strncpy(buf, ..., sizeof(buf)-1);
```

In the above case, the buffer will also be terminated. However, the following two cases show a common misuse of the `strncpy`:

- `buf[sizeof(buf)-1] = '\0';  strncpy(buf, ..., sizeof(buf));`
- `memset(buf, 0, sizeof(buf)-1);  strncpy(buf, ..., sizeof(buf)-1);`

In the first unsafe example, the string is null-terminated before the `strncpy`, and the execution of the function subsequently may overwrite the null-terminating character. In the second unsafe example, `memset` is used to zero-out the destination buffer; unfortunately, it is misused—the third argument needs to be the size of the entire buffer. Our FSA attempts to detect patterns that appear fishy, and alerts the user to their presence. Pattern variables are used judiciously to make MOPS precisely match the null-terminating code to the `strncpy` code that uses the same buffers. The property requires more manual inspection than other properties, because we have chosen an approach where we identify correct behavior, then raise an alarm at code that does not match our expectations. Moreover, our property does not attempt to find all `strncpy` bugs, focusing instead on patterns that are particularly suspicious.

Our `strncpy` FSA has alerted us to a number of bugs. Below we show one of the most interesting examples. It comes from the program *xloadimage*:

```
void dumpImage(Image *image, char *type, char *filename,
               int verbose) {
  int a;
  char typename[32];
  char *optptr;
  optptr = index(type, ',');
  if (optptr) {
    strncpy(typename, type, optptr - type);
    typename[optptr - type] = '\0';
  ...
```

In the above code fragment, MOPS identifies an idiom that does not appear to be safe. The character buffer `typename` is declared to be 32 bytes long, but the length passed to `strncpy` is computed entirely based on the second argument to the function `dumpImage`. We must verify that this string cannot be constructed in such a way that when `optptr - type` is computed, the result is longer than 32 bytes. MOPS is able to direct us to the call site of this function, in which we see the following (abbreviated):

```
newopt->info.dump.type= argv[++a];
...
dumpImage(dispimage, dump->info.dump.type, dump->info.dump.file,
          verbose);
```

Shockingly, the contents of the string come from the command line arguments, and can be set entirely by the user. Consequently, a malicious user can supply a carefully crafted argument to the function which causes the function `dumpImage` to write past the end of an array, causing a possible buffer overrun exploitation.

We are in the process of auditing all of the packages to tell, for each warning from MOPS, whether it is a real bug or a false positive. Our initial experiments suggest `strncpy` bugs are widespread, and we are optimistic that the property will be useful and feasible for bug-finding.

Correct Use of `chroot()`

The `chroot()` system call is tricky to use properly, and historically some programs have been rendered vulnerable by a failure to follow safe usage patterns. To give some background, calling `chroot(dir)` is intended to confine the program within the portion of the filesystem under `dir`, the *jail*. This is implemented by the kernel as follows: after `chroot(dir)`, the program's accesses to `/` are mapped to `dir` in the underlying filesystem (so that `dir` acts as the root of the filesystem for this one process), and as a security check the program's accesses to `..` from within that directory are re-directed so that the program cannot escape from its jail. So long as the program starts within this jail, the `chroot()` call is irrevocable: even if the program is later penetrated through, say, a buffer overrun, the attacker will not be able to escape this jail or affect any part of the filesystem outside of the jail. Consequently, many security-sensitive programs use the `chroot()`

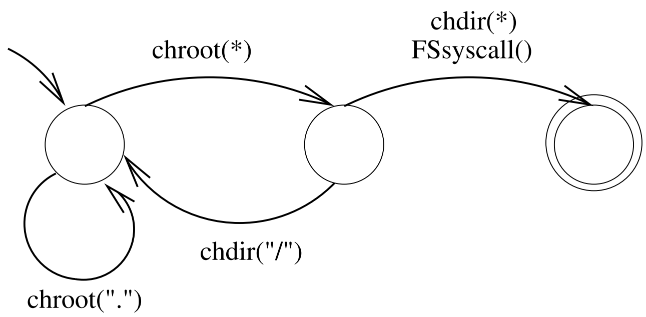


Figure 5.10: Our preliminary FSA to detect incorrect uses of `chroot()`. This FSA leads to a few false positives and is refined in Figure 5.11.

system call to voluntarily give up unneeded privileges and thereby reduce the consequences from any unexpected security compromises.

However, there is a subtle pitfall here. Due to the way that most Unix kernels implement this functionality, if the program's current working directory is outside the jail at the time of the `chroot()` call, it is possible to escape the jail. Consider a program whose current directory is initially `/home/joe`, and that calls `chroot("/var/jail")` but never changes its current directory. Suppose there is a vulnerability that allows an attacker to subsequently take control of the program, and the attacker executes `chdir("../..")`. This will succeed in changing the current directory to `/` (the real root of the filesystem, not the jail): directory traversal has never attempted to look up the name `..` from within the directory `/var/jail`, so the security checks are never triggered. At this point the attacker can modify the entire filesystem at will simply by using relative paths to refer to files outside the jail. In effect, this exploits a deficiency in the kernel's implementation of the `chroot()` system call.

One might think it would be easy to modify the kernel to eliminate this vulnerability, and indeed it would be, if not for one difficulty: this dangerous behavior has been enshrined in the

POSIX standard as required for POSIX compliance. Consequently, most[12] Unix kernels retain this dangerous functionality.

This, then, is a nasty pitfall for the unwary programmer; if the programmer is not careful, her intent to confine an untrusted program can be foiled. To prevent this risk, the safe way to use `chroot()` is to make sure that the program's current working directory[13] is within the jail. The simplest pattern is to simply use

```
chroot(dir);
chdir("/");
```

in place of `chroot(dir)`.

Accordingly, we built a property to look for deviations from this safe pattern. Any program that calls `chroot()` and subsequently performs any filesystem-related call without first executing `chdir("/")` is declared in violation. We quickly discovered that we needed to modify the property to recognize error-handling code, like the following, as safe:

```
rv = chroot(dir);
if (rv < 0) {
  perror("chroot");
  exit(1);
}
chdir("/");
```

Therefore, as a special case, we modified the property so that any program trace that exits immediately after a `chroot()` is not considered an error, even if it fails to call `chdir("/")` before exiting. See Figure 5.10 for a graphical depiction of the corresponding FSA. We define *FSsyscall* to be any system call that interacts explicitly or implicitly with the file system. There are over 100, so we do not list them here.

[12] Not all Unices have allowed this dangerous behavior to remain. OpenBSD has decided to forsake standards compliance in favor of reducing the risk of attack.

[13] Similar care must be taken with any open file descriptors the program might retain across the `chroot()` call. Our property does not attempt to check this requirement.

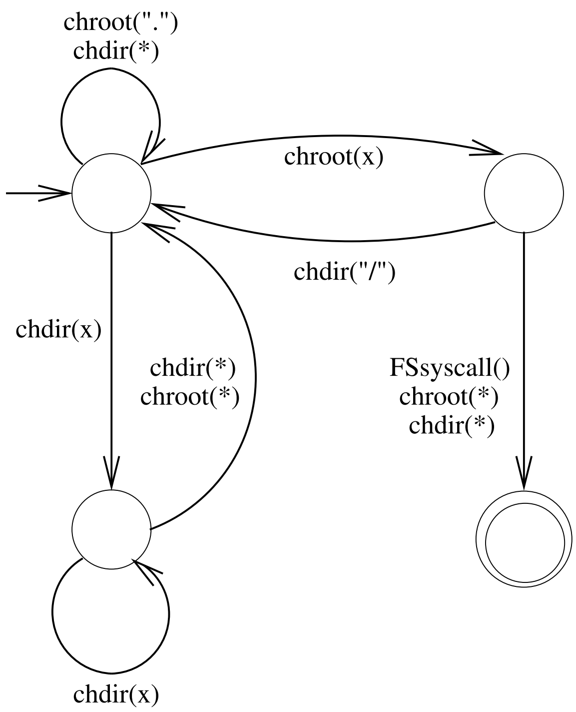


Figure 5.11: Our refined FSA for detecting `chroot()` vulnerabilities. This improves on the FSA in Figure 5.10 by reducing false positives.

After running this property across all Red Hat packages, we discovered several other common idioms for using `chroot()` that are safe. Because these idioms were not included in the aforementioned property, they led to 5 false positives. In particular, one safe idiom is

```
chdir(dir);
chroot(dir);
```

Similarly, another safe pattern is to move to the desired directory and then execute

```
chroot(".");
```

The obvious next step is to refine the FSA to allow all three of these patterns of usage. This refinement step turned out to be rather tricky, partially because `chdir(X); chroot(X)` is safe but `chdir(X); chroot(Y)` is not. We implemented this with a judicious use of pattern variables. See Figure 5.11.

With the aid of the refined property, we were able to confirm that none of the Red Hat packages contain unsafe usages of `chroot()`. There was only one false positive; see Figure 5.12. Arguably, the offending code segment should have been written more clearly, though the programmer has added a comment to clarify what is going on here. For comparison, there were 51 uses of `chroot()` spread across 30 different packages in the Red Hat sources. The low false positive rate shows that our refined property could reasonably be applied as part of standard development and release processes.

Preventing Format String Vulnerabilities

Some library functions are passed arguments containing format strings. For example, `printf()`'s first argument is a format string, and special sequences in it, such as `%d`, `%s`, etc., are replaced by values located on the stack. However, if the program passes an untrusted format

```
if (guest || restricted) {
    if (chroot(pw->pw_dir) < 0)
        goto bad;
}
if (krb5_seteuid((uid_t)pw->pw_uid) < 0)
    goto bad;
if (guest) {
    /*
     * We MUST do a chdir() after the chroot. Otherwise
     * the old current directory will be accessible as "."
     * outside the new root!
     */
    if (chdir("/") < 0)
        goto bad;
} else {
    if (chdir(restricted ? "/" : pw->pw_dir) < 0) {
        if (chdir("/") < 0)
            goto bad;
    }
}
```

Figure 5.12: The only false positive for the `chroot()` property. This code, taken from Kerberos 5's ftpd program, is safe; however, MOPS is unable to prove its safety, since MOPS does not perform path-sensitive data flow analysis.

string to `printf()`, a problem arises: the program will blindly attempt to retrieve values from the stack. This can lead to a compromise of security, as the user can specify format strings that cause `printf()` to write to arbitrary locations in memory. For example, suppose `buf` is a character buffer supplied by the user; then the call `printf(buf)` will introduce a security vulnerability. The correct usage is `printf("%s", buf)`. This type of vulnerability has been explored in previous work [50].

We detect potential format string vulnerabilities in a crude fashion: we identify all places in the code where format string functions may be supplied with non-constant character strings. Note that this introduces many false positives: for instance, `printf(buf)` might be safe if `buf` is a buffer known not to contain any format string sequences (for instance, it may have been initialized elsewhere to a value not under user control), but MOPS has no way of recognizing its safety in this case. Nonetheless, despite the presence of false positives, this simple property does represent one way to find all potential format string bugs in C code. We have built a FSA that identifies all format string functions and checks for non-constant arguments. Our FSA understands 61 different format string-related functions, and thereby generalizes and extends the set of functions checked for in Flawfinder, RATS, ITS4, and PScan.

After some experimentation, we soon encountered one important idiom with non-constant arguments to `printf` that is nonetheless recognizably safe. For instance, `printf(gettext(s))` is safe if `s` is a constant string, since we know the argument to `printf` will not contain a format string. Here we are assuming that the internationalization table is not maliciously constructed, hence the value `gettext(s)` can be trusted. We extended our FSA to recognize these additional safe programming idioms, thus reducing the number of false positives. With these improvements, we expect that our property will find at least as many bugs as previous tools, like RATS, ITS4, Flawfinder, and PScan, but with somewhat fewer false positives.

Nonetheless, when applying this property on all Redhat 9 packages, the number of false positives still remains very large. We are currently in the process of auditing our results and refining the FSA to reduce the number of false positives. Because of the large number of warnings that need to be checked, this process will take some time.

5.2.2 Results

In this section, we give further details on the results of our analysis of Redhat 9 packages. For each package, we show the program affected, the number of warnings (counted in terms of the number of trace groups), and the number of real security bugs. For space reasons, our summary for the TOCTTOU property omits those packages with no real bugs.

5.3 Conclusions

This chapter described our experiments of running MOPS on extremely large amount of code in open source applications. The experiments demonstrated both MOPS's power and its usability. As evidence for its power, MOPS can check a variety of security vulnerabilities, as diverse as privilege escalation, race conditions, standard file descriptor vulnerabilities, and temporary file vulnerabilities. MOPS is effective in discovering bugs: it helped us find dozens of bugs in our experiments. MOPS is scalable: it can check one million lines of C code within an hour. As evidence for its usability, MOPS provides its users with full error traces to help the users locate the bugs. Moreover, its error trace consolidation algorithm helps reduce substantially the number of

Package	*Program*	*Reported Warnings*	*Bugs*
SysVinit-2.84-13	wall	1	0
gnome-libs-1.4.1.2.90-32	gnome-pty-helper	2	0
gnuchess-5.02-11	gnuchess	6	0
initscripts-7.14-1	netreport	1	0
inn-2.3.4-2	rnews	1	0
isdn4k-utils-3.1-62	userisdncntrl	1	0
kon2-0.3.9b-16	kon	1	0
lockdev-1.0.0-23	lockdev	1	0
mc-4.6.0-4	cons.saver	1	0
mgetty-1.1.30-2	faxq-helper	4	0
ncpfs-2.2.1-1	ncplogin, ncpmap	2	0
rsh-0.17-14	rcp	1	0
slocate-2.6-8	slocate	1	0
slrn-0.9.7.4-9	slrnpull	3	0
sudo-1.6.6-3	sudo	2	0
util-linux-2.11y-9	chfn, chsh	1	0
vte-0.10.25-1	gnome-pty-helper	1	0
Total		30	0

Table 5.2: Results: the standard file descriptor property.

traces that the user has to review manually. These evidence demonstrates that MOPS is a viable, practical tool for discovering vulnerabilities in real world programs.

Package	*Program*	*Reported Warnings*	*Bugs*
binutils-2.13.90.0.18-9	ar	2	1
coreutils-4.5.3-19	chown	3	2
coreutils-4.5.3-19	chmod	2	1
coreutils-4.5.3-19	cp	2	1
dos2unix-3.1-15	dos2unix	4	2
ftpcopy-0.5.2-2	ftpcopy	8	3
gaim-0.59.8-0	gaim	3	3
joe-2.9.7-12	joe	1	1
jpilot-0.99.2-1	jpilot	2	1
initscripts-7.14-1	minilogd	1	1
inn-2.3.4-2	fastrm	1	1
isdn4k-utils-3.1-62	idsnlog	4	1
lrzsz-0.12.20-16	lsz	4	1
LPRng-3.8.19-3	checkpc	8	1
make-3.79.1-17	make	1	1
mc-4.6.0-4	mc	5	1
ncompress-4.2.4-33	compress	2	1
nmap-3.00-4	nmap	2	1
nmh-1.0.4-18	ali	1	1
pam-0.75-48	pam_console	5	1
postfix-1.1.11-11	postlock	2	1
postgresql-7.3.2-3	psql	4	1
rdist-6.1.5-26	rdist	18	3
rsh-0.17-14	rcp	4	2
strace-4.4.95-2	strace	1	1
symlinks-1.2-18	symlinks	4	2
talk-0.17-20	talkd	1	1
unix2dos-2.2-19	unix2dos	4	2
util-linux-2.11y-9	sln	2	1
zip-2.3-16	zip	7	1
Total		108	41

Table 5.3: Results: the TOCTTOU property.

Package	*Program*	*Reported Warnings*	*Bugs*
anacron-2.3-25	anacron	1	0
binutils-2.13.90.0.18-9	ar	2	2
compat-gcc-7.3-2.96.118	texindex	1	0
cproto-4.6-15	cproto	1	0
inn-2.3.4-2	actsync	1	0
inn-2.3.4-2	ctlinnd	1	0
inn-2.3.4-2	innxmit	1	0
inn-2.3.4-2	rnews	1	0
inn-2.3.4-2	shrinkfile	1	0
imap-2001a-18	impad	1	1
isdn4k-utils-3.1-62	vbox	2	0
krb5-1.2.7-10	kprop	1	0
lha-1.14i-9	lha	1	0
libungif-4.1.0-15	gifinto	1	1
lrzsz-0.12.20-16	lrz	1	1
lv-4.49.4-9	lv	2	0
m4-1.4.1-13	m4	1	0
mailx-8.1.1-28	mail	1	0
mikmod-3.1.6-20	mikmod	1	1
mpage-2.5.3-3	mpage	1	0
ncurses4-5.0-11	tic	1	1
nmh-1.0.4-18	comp	2	0
patch-2.5.4-16	patch	1	0
patchutils-0.2.19-1	interdiff	1	0
pine-4.44-18	impad	1	0
pinfo-0.6.6-4	pinfo	1	0
psutils-1.17-19	psbook	1	0
rcs-5.7-20	co	2	0
rdist-6.1.5-26	rdistd	1	1
rdist-6.1.5-26	rdist	1	1
util-linux-2.11y-9	pg	2	0
vnc-3.3.3r2-47	imake	2	0
x3270-3.2.19-4	mkfb	1	0
Total		40	9

Table 5.4: Results: insecure creation of temporary files (excluding *mkstemp* template reuse; see Table 5.5).

Package	*Program*	*Reported Warnings*	*Bugs*
byacc-1.9-25	yacc	5	3
chkconfig-1.3.8-1	ntsysv	1	0
dos2unix-3.1-15	dos2unix	4	2
dump-0.4b28-7	restore	1	0
file-3.39-9	file	1	1
gftp-2.0.14-2	gftp-gtk	1	1
ggv-1.99.97-2	ggv	1	1
gnome-libs-1.4.1..	gconfigger	2	1
gpm-1.19.3-27	gpm	1	0
isdn4k-utils-3.1-62	idsnrep	2	0
mars-nwe-0.99pl20-12	nwbind	2	0
mktemp-1.5-18	mktemp	1	1
mpage-2.5.3-3	mpage	1	1
nautilus-cd-burn..	nautilus-cd..	1	1
nkf-2.01-2	nks	1	1
ntp-4.1.2-0.rc1.2	ntp-genkeys	1	1
nvi-m17n-1.79-20..	nvi	2	2
openssh-3.5p1-6	sshd	1	0
pax-3.0-6	pax	2	2
quota-3.06-9	edquota	4	4
redhat-config-pr..	traditional	1	0
rwall-0.17-17	rwall	1	1
shadow-utils-4.0..	useradd	3	0
skkinput-2.06.3-3	skkinput	2	0
sndconfig-0.70-2	sndcondig	1	0
sylpheed-0.8.9-3	slypheed	1	0
xfig-3.2.3d-12	xfig	2	1
zebra-0.93b-1	bgpd	2	1
Total		48	25

Table 5.5: Results: re-use of the *mkstemp* template.

Chapter 6

Experience on EROS

Chapter 5 described our experience with running MOPS on open source application programs. This chapter describes our experience with running MOPS on the EROS kernel.

6.1 Overview of EROS

EROS is a capability-based operating system that runs on commodity hardware. Applications invoke user-implemented objects and kernel services by invoking kernel-protected capabilities. Our focus of attention in this paper is the EROS microkernel, and specifically the invariants of the microkernel. A more complete discussion of the EROS design can be found in [53, 51]

One view of a microkernel is that it implements an extended virtual machine that is derived from and built upon the underlying hardware architecture. In microkernels, this extended virtualization generally provides a primitive interval timer, multitasking, virtual memory, and exception encapsulation. In addition, microkernels provide a small set of system calls necessary to direct and manipulate the virtualization performed – to define memory maps, set scheduling policy, and so forth. Allowing for some variations in individual implementations, there are either three or four substantive entry points in a typical microkernel:

1. The interval timer interrupt, whose arrival conditionally results in a scheduling preemption.

2. The page fault interrupt, which signals to the microkernel that it either needs to translate virtualized mappings into hardware mappings or request application-level action.

3. The system call trap, which directs the performance of a microkernel service, possibly an interprocess operation.

4. The major preemption entry point, which is invoked when the current kernel thread of control has become blocked and a new thread of control must be selected. This entry point only exists in interrupt-style kernels, including EROS.

While other hardware exceptions have associated entry points, microkernels generally implement these as "fast path" entry points whose effect is to record fault information in the process control block of the currently executing process and then invoke the major preemption entry point. We will not consider exceptions of this kind further in this paper.

Three aspects of the EROS kernel design are particularly relevant to model checking: the fact that it is an interrupt style kernel, the single level store, and the state caching design of the system.

Interrupt-style kernel An interrupt-style kernel [25] is one in which processes do not retain a kernel stack while blocked. On wakeup, the process resumes by restarting the system call that originally caused it to sleep.

This style of kernel design imposes a transaction-like structure on the execution of system calls. Kernel service invocations proceed by testing preconditions, reaching a “commit point,” and then performing the requested operation. No externally visible modifications are permitted prior to the commit point, and after the commit point the service invocation is required to run to completion successfully. If a precondition cannot be satisfied, the requesting process is placed on a queue and is required to restart its invocation from scratch. One way to think of this is that every system service invocation carries an implicit “catch block,” and that any cause of blocking involves a “throw” that is caught at the system call boundary.

The transactional style of execution naturally results in temporal safety invariants that must be observed. Procedures that cause externally observable state changes must not be called prior to the commit point. Procedures that might remove objects from memory must not be called after the commit point has been reached. Procedures that check preconditions (e.g. Is the object we are mutating writable?) may be viewed as causing typestate transitions [56, 55] in a temporal safety property. Many of the key temporal safety invariants of interrupt-style kernels can be expressed as finite state machines.

Single-Level Store The EROS system implements a transparent single-level store built on a global snapshot and checkpointing system. One consequence of this design is that an object descriptor (a capability) refers to an object that can be either in memory or on disk. Descriptor usage can conditionally induce object faults, and transient pinning is used to ensure that objects required in a particular operation remain in memory for the duration of the operation. This induces restrictions on what procedures can be permissibly called at what points in the kernel control flow. After a commit point is reached, no operation is permitted that might result in an object removal. Prior to a commit point, no externally visible modifications to kernel state are permitted.

Caching Design EROS implements a caching approach to managing system state [12]. Process and memory map state is stored in user-allocated, kernel-protected data structures called *nodes*. Hardware-manipulable page tables and process control blocks are managed as a software-managed write-back cache of the state in these nodes [51]. This software-management strategy demands data structures for dependency tracking. EROS requires that the code that maintains these data structures must follow certain control flow invariants, many of which can be checked by a model checker. For example, one invariant requires that the code should not modify any page table entry without a preceding call to create an entry in the data structures for dependency tracking.

Collectively, the state machines associated with these properties govern the majority of the EROS kernel's function. If we can establish through formal techniques that the transition rules of these state machines are satisfied, considerable confidence emerges in the global structure of the system. Better still, these formal checks serve as a guard against inadvertent error during maintenance. Model checking will not allow us to find local errors in the code, but may significantly reduce exposure to violations of global design rules. The aggregate constraint behavior of a large, complex system results from complex interactions between relatively simple individual state machines that guard particular aspects of overall consistency. By model checking these state machines individually, with some attention to the interaction points between them, the satisfaction of complex overall constraints can be assured.

6.2 Properties and experiments

To evaluate the effectiveness of MOPS in establishing assurance, we modeled five properties of the EROS implementation. The first three can be expressed purely as control flow invariants, and fall directly within the space of properties that MOPS was designed to check. The remaining properties are typestate properties [56, 55]. We were interested in part to determine whether we could adequately approximate these using MOPS pattern variables, and also to determine whether it would be worthwhile to extend MOPS or some related tool with support for typestate.

It should be emphasized that EROS is in many respects an ideal subject system for this sort of analysis, and in consequence a frustrating one. The history of "design by invariant" in EROS and its predecessor KeyKOS collectively extends back over 30 years. This has the effect of making the system friendly to the type of analysis reported here. In the eyes of the checker, however, it is also somewhat frustrating: invariant discipline is very effective in resisting bugs, and there were relatively few to find. To safeguard MOPS against implementation errors that cause MOPS to miss bugs, for each property we artificially inserted bugs into the EROS kernel and ensured that MOPS found them.

6.2.1 Interrupt enable and disable, yield

The simplest property we attempted was ensuring that interrupt enables and disables are properly bracketed. Every disable should be balanced by a corresponding enable, and there should not be redundant enables. In addition, the `Yield()` function (which abandons the current system call) should not be called while interrupts are disabled. This is because `Yield()` abandons the current kernel control flow and invokes a kernel system call, but all kernel system calls are expected

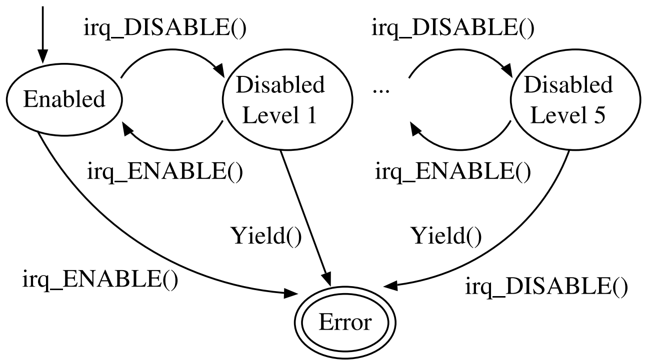


Figure 6.1: An FSA describing the property: *(1) each `irq_DISALBE()` should be balanced by an `irq_ENABLE()`; (2) there should be no redundant `irq_ENABLE()`; (3) `Yield()` should not be called while interrupts are disabled.*

to be invoked with interrupts enabled. The FSA in Figure 6.1 describes this property.

A problem with checking interrupt enable and disable is that it requires a stack or a counter. Finite state automata are not powerful enough to implement counters, but in practice the enable/disable depth of a well-structured kernel is not deep. To cover the current behavior of the kernel, we implemented five distinct disabled states, and used these to simulate up to five levels of nesting. Additionally, we used the error state as a guard state to validate that five levels were enough. The need for fixing a maximum number of nesting levels came as an unpleasant surprise; one of the ancillary outcomes of this analysis was a decision to review the use or interrupt nesting to reduce this depth.

Consistent with our expectations, we did not find any violations of this property. We *did* modify the EROS source code to convert a table-driven capability handler dispatch mechanism into an equivalent switch-driven mechanism so that the checker would be able to see that these routines are reachable. The resulting code is functionally equivalent and probably more efficient, as the compiler now has the option to implement the dispatch using a vector of branch instructions. We also modified one place to rewrite a correctly conditionalized disable/enable pair so that the static

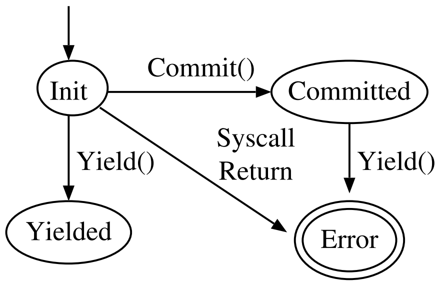


Figure 6.2: An FSA describe the property: *(1) every system call control path should invoke exactly one of* `Yield()` *or* `Commit()`*; (2) following* `Commit()`*, the kernel should not call* `Yield()`.

checker would not report a false positive error trace. This was a false positive that could readily be resolved through conditional liveness analysis [55].

6.2.2 Yield, commit

As has previously been mentioned, EROS is an interrupt-style kernel. The flow of operation proceeds by first checking preconditions and then performing the operation itself. These two phases are separated by what we call a "commit point." A kernel invocation can yield before the commit point, but not after. This constraint leads to two rules:

- Every system call control path should invoke exactly one of `Yield()` or `Commit()`.
- Following a call to `Commit()`, it is a bug to subsequently call `Yield()`.

The `Yield()` function does not return, so we do not check for `Commit()` after `Yield()` or `Yield()` after `Yield()`. The FSA in Figure 6.2 describes these rules.

The "yield or commit" design rule is fundamental to the EROS security model. The consistency of our security model is predicated on an argument about atomic, stepwise correct evolution of the protection graph [52]. The yield or commit design rule is the rule that guarantees the atomicity of the protection graph transformations. This was in fact the property that we attempted first, because we were struck by how direct this property corresponds to the previous formal verification work on the EROS confinement mechanism [52].

We found a bug in the course of this analysis. After a commit point, when the EROS kernel is midway through a transition to more dynamic heap management, one system service call used by user-mode drivers allocates heap memory from a finite kernel pool by calling `malloc()`. Should this allocation fail, `malloc()` calls `Yield()`. This violation of the "yield or commit" design rule revealed that we had entirely failed to consider this challenge when we decided to introduce `malloc()` into the kernel. The correct repair in the context of the current EROS design is to allocate all required memory during a "dry run" phase prior to the commit point. Fortunately, this is the *only* system service in the EROS kernel that invokes `malloc()` after startup initialization has completed, so we do not anticipate great difficulties in correcting this flaw.

6.2.3 Sleep and yield

One very effective way to create problems in a kernel is to mishandle process sleep and wakeup. Kernels have a large number of stall queues, and mishandling of these queues can easily lead to lost processes or data corruption. Because stall queue manipulation occurs with interrupts disabled, these errors are difficult to debug. Sleep handling in EROS differs from widely-used kernels in two significant regards:

- A process can be asleep on at most one stall queue at a time. There is no equivalent to the UNIX `select()` operation in the EROS kernel.
- The `Sleep()` and `Yield()` operations are not atomically joined. In an interrupt-style kernel, it is sometimes convenient to place the process to sleep on the appropriate stall queue

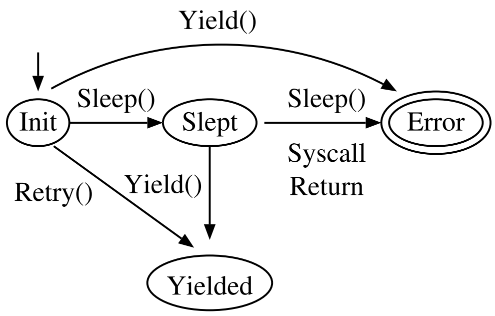


Figure 6.3: An FSA describing the property: *(1) no path should call `Sleep()` more than once; (2) after a `Sleep()` call, the kernel must call `Yield()`; (3) before a `Yield()` call, the kernel must call `Sleep()`, unless it has called `Retry()`.*

and to perform additional processing using the kernel thread of control before relinquishing the processor.

The first property implies that no path through the kernel should call `Sleep()` more than once, and the second implies that every call to `Sleep()` should be followed at some later point by a call to `Yield()`. The FSA in Figure 6.3 describes these properties.

There is a third property that is almost but not universally true: nearly every call to `Yield()` follows a `Sleep()` of some sort. Conceptually, this is because a `Yield()` generally means that there was some reason that an operation could not be completed immediately, and all of the operations that might eliminate such an impediment signal their completion by waking up a stall queue somewhere. Given this, one would expect that a yielding process should be on a stall queue somewhere, which is achieved by calling `Sleep()`.

When we originally specified this property as above, we neglected an important exceptional case. There are certain cases where the kernel *may* be in an inconsistent state but does not know for sure. For example, a system call or a page fault may rely on the continued existence of a page table entry, but may need to bring an object into memory, which may invoke the aging logic to evict something to make room. The eviction may invalidate one or more page table entries, and these page table entries may turn out to be the ones that the current operation is relying on. It is not cost-effective to remember which page table entries are specifically implicated by the current call; instead, the kernel keeps a global boolean flag `PteZapped`, which is cleared on entry to the kernel and is set whenever a page table entry is invalidated. This approach is conservative, but effective.

In defensive cases such as the one described above, it is appropriate for the current operation to perform a `Yield()` voluntarily. Because the process has not gone to sleep, it has not relinquished its ownership of the CPU, and will retry the current operation. Assuming that there were no real impediments to completion, the system call will complete successfully during the second pass.

To capture both the property and the intent correctly, we modified the source code to call `Retry()` in these special cases (which is simply a wrapper for `Yield()`), and we modified the property to stop in a successful state whenever `Retry()` is called, as shown in Figure 6.3. With this modification, no further errors were found in the kernel.

6.2.4 Prepare before GetRegs

The next property validates one of the caching requirements of the EROS kernel. Certain process-related operations, such as `proc_GetRegs32()`, operate by fetching values from the process control block. Because the process may be inactive when registers are requested, it is necessary to encache the process first by calling `proc_Prepare()`. The property that we would like to check here is a typestate property: any call to `proc_GetRegs32(`p`)` requires that the typestate of process p is "cached." `proc_Prepare(p)` changes the typestate of process p from "unknown" to "cached."

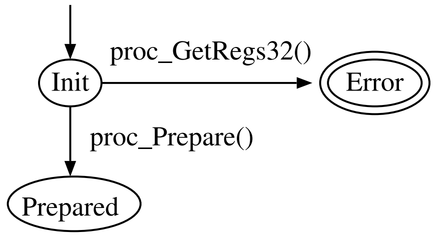


Figure 6.4: An FSA describing the property: *before a `proc_GetRegs32()` call, the kernel must call `proc_Prepare()`.*

MOPS does not implement typestate checking, but it *does* provide a feature called "pattern variables." Using pattern variables, we can statically verify that given a code sequence of the form

```
proc_Prepare(param1)
...
proc_GetRegs32(param2, &regset)
```

the arguments passed in as *`param1`* and *`param2`* are textually identical (by which we mean that their abstract syntax trees match). This is a much weaker check than the typestate check, but it is pragmatically sufficient for our purposes. The EROS kernel does, in fact, use the same variable in both calls, and we know from the implementation that any attempt to decache the process must ultimately call either `proc_NeedRevalidate()` or `proc_Unload()`. As long as neither of these functions is called, a cached process will remain cached. Therefore, so long as the value of *`param1`* has not changed before the call to `proc_GetRegs32`, we know that the process remains cached. The FSA in Figure 6.4 describes this property.

Note that because of the possibility of an intervening decache, the property we are trying to test is truly a typestate property; static types are insufficient to describe the control flow constraints we have described. The point of our trial was to understand how, in practice, typestate analysis would be helpful in validating properties about this sort of kernel. Our conclusion is that typestate is an essential tool for validation of this sort, but we note that in an interrupt kernel a surprising number of typestate properties can be checked without risk of ambiguities that might arise from failures of alias analysis. In this case, it is sufficient to know (a) that the argument process p was cached, (b) that this value of p reached the call to `proc_GetRegs()`, and that (c) there was no intervening decaching operation on *any* process.

We found no error in checking this property.

6.2.5 PTE dependency tracking

Our final check attempted to validate the consistency of the EROS memory management subsystem. EROS applications specify their address space structures using a tree of fixed-size capability lists (c-lists). The kernel traverses these structures on demand to construct page table entries. If a capability in one of the translated c-lists is subsequently overwritten, the derived page table entries must be invalidated. To ensure this, EROS maintains a dependency tracking data structure known as a *depend table* that maps from capability addresses (the address of the traversed slot in the c-list) to page table entry addresses. Whenever a c-list entry is overwritten, the depend table is used to invalidate the appropriate page table entries.

The design rule that we wanted to check is that no page table entry should be modified without a preceding call to create a depend table entry. This test is approximate. Since MOPS does not provide either typestate or value tracing, we did not attempt to check that the depend table entry was associated with the *right* page table entry. The overall goal of the algorithm is to ensure that every path in the hardware mapping table tree (Figure 6.6(b)) corresponds to some path in the node tree. (Figure 6.6(a)) The FSA in Figure 6.5 describes this design rule.

The check of this property revealed five places where page table entries were set without

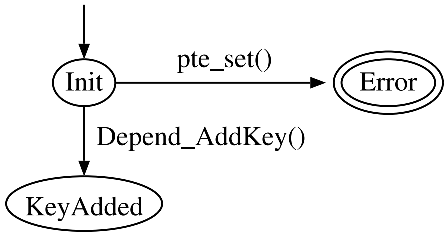
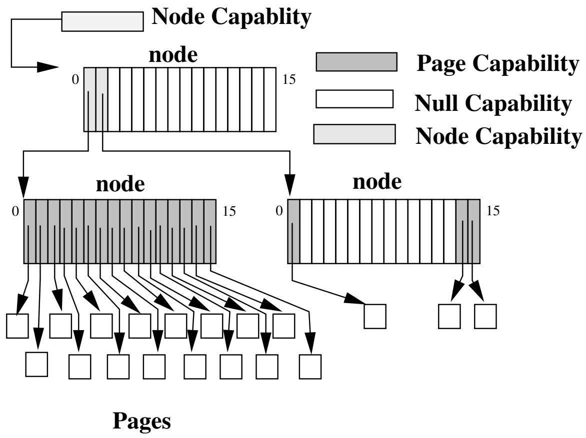


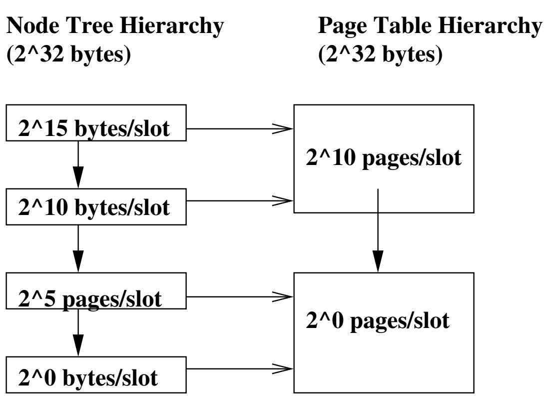
Figure 6.5: An FSA describing the property: *before a `pte_set()` call, the kernel must call `Depend_AddKey()`.*

any dependency entry construction that MOPS could detect. Examination of the cases was sufficient to show that all five statements were correct, and that all five in fact *did* obey the property, but that this could not be detected statically. The cases arose because of an optimization in which a restarted page fault could bypass translation steps that are known to have been previously performed, because the required dependency table entries have been constructed in the previous pass. It is not entirely surprising that this check fails. The optimization in question is based on a correspondence between two trees of different arity in which the second tree (the mapping table hierarchy) is a lazily generated partial projection (Figure 6.6) of the first (the node tree) [51]. Because of the difference in tree arity, the correctness of the correspondence – never mind the implementation – is not immediately obvious, and the correct implementation of this projection by the code is less so.

Matters are further complicated by the fact that the algorithm takes advantage of reverse correspondences between the two trees to avoid traversing the more expensive mapping structure when possible. The algorithm relies heavily on checks of values produced by previous passes through the translation code to determine the correct translation strategy. It is not clear whether the temporal safety properties of this code can be validated even with typestate support, because the automaton is unable to consider the cumulative effect of successive invocations of the address translation subsystem.



(a) Address mapping expressed as a node tree.



(b) Projection from nodes to page tables.

Figure 6.6: Correspondence between node and page table trees.

In our view, there are two lessons to take away from the failure of this check. The obvious one is that not every invariant can be reduced easily to temporal safety properties. The less obvious one is that *most* invariants can: of the many invariants that we attempted to check in the EROS kernel, this is the only one that failed entirely. Others would have required typestate analysis, but clearly fall within the scope of what model checking technique can do.

6.3 Evaluation

We argue that usability and performance are two key criteria for evaluating the feasibility of integrating a static analysis tool into the kernel development process. For usability, we measured how easy it is for non-tool developers to write domain specific properties. For performance, we measured how much overhead the static checking adds to the build process.

6.3.1 Usability

The five properties that we checked in Section 6.2 are design invariants in the EROS system. Historically, EROS has a long track record of specifying invariants before implementing them. While these specifications have not been collected in a single place (but ought to be), they have been published in email discussions, design notes that are part of the source tree, and the original KeyKOS design document [37].

In the course of this experiment, our use of MOPS was iterative. It took us just a few minutes to write the initial property specification, but we would check this property and found that it produced errors because the specification was in some way incomplete. Then we would add new states and transitions to the specification to address the deficiency and would re-run the property.

Two or three iterations of this were generally sufficient to capture the property accurately.

In general, we found few false traces with the refined FSAs. Most of these were due to failures of conditional liveness analysis, and were resolved by making minor modifications to the EROS code. In contrast to some other tools we have used, the modifications yield more readable and more maintainable code. In one or two cases we were unable to eliminate the last one or two false positives within a reasonable amount of effort. These cases and their explanations are now documented in the source code.

6.3.2 Performance

From the developer perspective, a key concern with MOPS was the run time cost of generating and building the CFG files. The EROS kernel has approximately 25,704 lines of code, of which 4,153 are header lines that are widely included. Building the default configuration of the EROS kernel without MOPS checking takes 12.13 seconds. When running the current version of MOPS on EROS, building and linking the same configuration takes 31.34 seconds, the majority of which is spent in Java I/O. Model checking all the five properties in Section 6.2 takes 100.13 seconds. This is a large enough addition of time that we tend to omit it during the "edit compile debug" loop, but small enough that we can make it a routine part of our major builds without noticeable cost.

Earlier versions of MOPS suffered from naively implemented I/O. The I/O code was easily improved, and the actual processing time used for managing the CFG was never a source of concern. However, the Java stream I/O package is difficult to use efficiently, and garbage collection is not well suited to applications that retain large graph structures for extended processing. Our sense is that an implementation crafted in a more conventional systems programming language might well yield a factor of ten improvement in the performance of the checking phase, and reduce the CFG construction times down to something less than the normal compile phase. This would be sufficient to let us run MOPS in every build we do.

6.4 Conclusions

Working with the EROS developers provided us with a unique opportunity to evaluate the feasibility and the value of integrating MOPS into the development cycle of security-critical software. Our experiments affirmed the potentials of both. MOPS is fast enough to become a routine part of EROS's major builds without noticeable cost (and could be made faster to become a part of every build), and MOPS can be easily embedded in the EROS build process. Perhaps the most important value of MOPS is its soundness: because MOPS is sound under certain conditions, it not only can find bugs but also can prevent bugs from being re-introduced.

Chapter 7

Related Work

7.1 Full program verification

Formal verification is an approach to demonstrate that a software implementation satisfies its design. In this approach, the user specifies the behavior of a program, and the verification tool checks if the program satisfies the specification. LARCH [29] is such a tool. Another example is the B-Method [48], which provides a notation, a method, and a toolkit for requirement modeling, software interface specification, software design, implementation and maintenance. Although these tools can lead to astonishingly strong code, the work to create the specification is often heavy, and the skills needed are beyond most software programmers.

Perhaps the best documented effort to verify completely properties about a substantial software system is the verification work on PSOS [22, 44]. The PSOS verification method relies on a hierarchical structuring of the system design and its implementation followed by a theorem-proving verification of properties across the layers of the design. While this method generates justifiably higher confidence than the one used by MOPS, the effort required is substantially larger and the confidence of full verification is not required in all applications. At some threshold, bounding the maintenance cost of assurance over the course of the software life cycle becomes more important than achieving a higher degree of assurance. In such situations, lighter methods such as MOPS may provide a more effective cost/benefit trade-off.

7.2 Lightweight program verification

Light-weight formal verification has been proposed to overcome the complexity and scalability problems in full formal verification. Instead of creating a precise but complex specification of the program, the light-weight verification user selects a set of simple properties that are easy to specify and computationally cheap to verify. As such, light-weight verification tools can afford to use relatively simple and scalable machinery. Splint [20, 21] is a tool for statically checking C programs for a pre-defined set of common coding errors. Since it checks only those properties that were envisioned by the tool developer, it cannot be used to check application-specific properties.

SLAM [3, 2] uses software model checking to verify user-specifiable temporal safety properties in programs. It iteratively finds error traces using a model checker, rejects infeasible traces using a theorem prover, and refines a boolean abstraction of the source program. BLAST [30, 31], a software model checker similar to SLAM, uses lazy abstraction to reduce unnecessary abstraction refinement. ESP [15] is a tool for verifying temporal safety properties in C/C++ programs. It uses a global context sensitive, control flow insensitive analysis in the first phase and an inter-procedural, context sensitive dataflow analysis in the second phase. Among these tools, SLAM/BLAST are the most precise and least scalable, MOPS is the most scalable and least precise, and ESP is between SLAM/BLAST and MOPS in precision and scalability. CMC [41] model checks a system for erratic behaviors. It models the system as a collection of interacting concurrent processes. The state of the system is the union of the states of all its processes along with the contents of the shared memory, and the state of each process consists of its global variables, heap, stack, and context registers. CMC handles its state explosion problem by using hash tables and heuristics, which effectively compromises soundness. Jensen et al. model checked a special class of security properties in Java using only control flow analysis [35, 4]. Its algorithm, however, requires that one specifies a fixed, finite bound on the size of the program stack. MOPS has no such limitation.

Rather than focus on temporal safety properties, some tools are concerned mainly with data flow properties. Koved et al. used context sensitive, flow sensitive, inter-procedural data flow analysis to compute access rights requirement in Java with optimizations to keep the analysis tractable [38]. CQUAL [26] is a type-based analysis tool that provides a mechanism for specifying and checking properties of C programs. It has been used to detect format string vulnerabilities [50] and to verify authorization hook placement in the Linux Security Model framework [63], which are examples of the development of sound analysis for verification of particular security properties. Recently, Johnson and Wagner [36] use CQUAL to detect user/kernel pointer bugs. CQUAL and MOPS complement each other: MOPS is control flow sensitive but data flow insensitive, while CQUAL is data flow sensitive but control flow insensitive. By combining their analyses, we envision a more powerful tool that can check a wider range of properties and report more precise errors. Extended Static Checker for Java (ESC/Java) [23] uses verification-condition generation and automatic theorem-proving techniques to find common programming errors. ESC/Java requires the user to annotate programs to express their design decisions, which makes it harder to use and maintain in the software development process. MOPS does not require source code annotation.

The tools that we discussed above can check a variety of properties. In comparison, there are also tools are designed to check specific properties, mostly because these properties require distinctive analyses. A prominent example is the work by Wagner et. al. on finding buffer overruns using integer range analysis [59]. It complements MOPS in the sense that the properties that MOPS checks does not overlap the properties that it checks.

7.3 Bug finding tools

Besides light-weight verification tools, there are light-weight bug finding tools. They do not attempt to verify the absence of bugs in a program; rather they find bugs in a program at best-effort. Metal, a ground-breaking bug finding tool, which is similar in concept to a model checker, has been very successful at finding a broad variety of rule violations in operating systems [1, 18]. Metal has been augmented with Z-ranking, a powerful technique for reducing the number of false positives, and used to find many bugs in large applications [39]. Metal is unsound, because it neither guarantees to find all bugs for certain properties nor specifies the conditions where it is sound.

Today, the state of the art for most practitioners involves tools that scan program source code for simple syntactic patterns [42, 61]. For instance, RATS [47], ITS4 [58], Flawfinder [60], and PScan [17] are all based on analyzing the stream of tokens produced by the lexer to look for things like function calls to certain error-prone library calls. MOPS is able to look more deeply, both because of its ability to match on complex AST patterns and because of its ability to look for certain sequences of operations by using control-flow analysis. As a result, all of the properties described in this paper go beyond what those lexical tools can find. Nonetheless, the popularity of even very simple tools demonstrates a growing interest in security auditing tools.

7.4 Security properties

We learned security vulnerabilities mainly from Unix manual pages, bug tracking mailing lists, secure programming tutorials, and folklore in the security community. Based on the knowledge from these sources, we designed security properties that can detect these vulnerabilities.

Bishop discussed security vulnerabilities in setuid programs [5]. His focus is on potential vulnerabilities that a process may be susceptible to once it gains privilege, while our focus is on how to gain and drop privilege confidently and securely. Unix systems have evolved and diversified a great deal since Bishop's work in 1987, and a big problem today is how to port setuid programs securely to various Unix systems.

File system race conditions have been extensively studied in the computer security literature. Bishop and Dilger first articulated the vulnerability pattern and developed a syntactic pattern-matching analysis for detecting TOCTTOU vulnerabilities in C code [6]; however, because their analysis is not semantically based, it is unable to find many of the vulnerabilities found in this work. Dean and Hu propose a probabilistic defense that makes it very difficult for adversaries to win races [16]; however, their approach requires re-writing existing code. Also, some authors have proposed runtime program analysis methods to detect TOCTTOU bugs by monitoring program executions and preventing their exploitation [14]; our work differs by trying to find TOCTTOU bugs at compile time, rather than at runtime.

Chapter 8

Conclusions

In this dissertation, we have shown that using lightweight model checking for improving software security is effective and practical. We started by proposing a formal framework for discovering security bugs in software. Then, we implemented the framework in the tool MOPS, and designed new algorithms to improve MOPS's usability considerably, including reporting concise, comprehensible error traces and integrating MOPS into the build process of software packages. Finally, we used MOPS to audit both application and kernel code on a large scale. For application code, we checked five properties on over one million lines of code in open source software and discovered more than a dozen bugs and weaknesses. For kernel code, we verified five design invariants on the EROS kernel.

In order to be effective as a tool for discovering or preventing temporal safety errors, a verification technique must meet several requirements:

- It must be sufficiently sound (and must document clearly the conditions under which it may become unsound). The tool must not fail silently.

- Its specifications must be expressed in a form that typical programmers can write.
- It must not require invasive changes to the code base.
- It must be efficient enough to be incorporated into the normal build process.

MOPS satisfies all of these properties.

The value of this dissertation goes beyond MOPS. As we discussed in Chapter 7, MOPS is an example of several lightweight static code auditing tools. Some of these tools complement MOPS in several aspects. For example, some tools are more precise (but at the expense of less scalability), and some tools can check properties that MOPS cannot check. Our successful experience with MOPS signifies potential exciting opportunities and wide impact for these tools. Furthermore, our engineering efforts for integrating MOPS into the build process of software packages apply to many these tools as well.

Chapter 9

Future Work

Our strong results in using lightweight program verification to detect and prevent security bugs open many avenues for future work. One obvious direction is to enrich MOPS's analysis. Currently, MOPS's lack of data flow analysis not only restricts the variety of security properties that it can check but also increases false positives in its error reports. We must, however, proceed with caution, because MOPS's success attributes to its lightweight analysis, so we must carefully evaluate the overhead of candidate data flow analysis techniques. An alternative is to combine MOPS with another tool that does data flow analysis. CQUAL is a good candidate, because it complements MOPS's analysis nicely. A combined MOPS/CQUAL tool might first run CQUAL to annotate the AST of the program with data flow information, and then run MOPS, which can use the data flow information provided by CQUAL, to check security properties.

MOPS is one of many static analysis tools that audit programs for bugs. Many of these tools share certain problems, among which usability is a prominent one. In order to promote wide spread use of code auditing tools, we need to answer several usability questions, such as: How much do these tools improve code auditing? What do users like and dislike about these tools? How can we improve users experience? We are in the process of setting up a user study for answering these questions.

We developed MOPS for finding security vulnerabilities in source programs. Beyond source programs, security vulnerabilities may exist in many other parts of the system, so it would be valuable if we can apply MOPS's techniques to these vulnerabilities. Particularly, we think that MOPS is promising in addressing configuration security. Experience shows that misconfiguration has caused many security vulnerabilities and will continue causing more. This is because many configurations are complex and error prone, but typical system administrators are under-qualified to configure their software securely. As more software are becoming more user configurable, we expect misconfiguration to become a bigger problem in the future. Just as lightweight static analysis has helped improve code security, we anticipate that it could help improve configuration security as well. One approach might be to ask the developers of each software to supply a specification for checking insecure configurations; then, system administrators can use a MOPS-like tool to check their customized configurations against the specification to discover potential vulnerabilities in the configuration.

Bibliography

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

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

[3] Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL '02: Proceedings of the ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, 2002.

[4] Frederic Besson, Thomas Jensen, Daniel Le Metayer, and Tommy Thorn. Model checking security properties of control flow graphs. *Journal of Computer Security*, 9:217–250, 2001.

[5] Matt Bishop. How to write a setuid program. *;login:*, 12(1):5–11, 1987.

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

[7] BSD. dm(8). 4.4 BSD System Manager's Manual.

[8] Hao Chen, Drew Dean, and David Wagner. Model checking one million lines of C code. In

Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS), pages 171–185, San Diego, CA, February 4–6, 2004.

[9] Hao Chen and Jonathan Shapiro. Using build-integrated static checking to preserve correctness invariants. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, November 25–29, 2004.

[10] Hao Chen and David Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, pages 235–244, Washington, DC, November 18–22, 2002.

[11] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *Proceedings of the 11th USENIX Security Symposium*, pages 171–190, San Francisco, CA, August 5–9, 2002.

[12] David R. Cheriton and Kenneth J. Duda. A caching model of operating system kernel functionality. In *Proc. USENIX Symposium on Operating Systems Design and Implementation*, pages 179–193. USENIX Association, November 1994.

[13] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. McGraw-Hill, 1989.

[14] C. Cowan, S. Beattie, C. Wright, and G. Kroah-Hartman. RaceGuard: Kernel protection from temporary file race vulnerabilities. In *Proceedings of the Tenth USENIX Security Symposium*, 2001.

[15] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[16] Drew Dean and Alan Hu. Fixing races for fun and profit. In *Proceedings of the Thirteenth Usenix Security Symposium*, August 2004.

[17] Alan DeKok. PScan: A limited problem scanner for C source files. Available at `http://www.striker.ottawa.on.ca/aland/pscan`.

[18] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, 2000.

[19] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Lectures Nodes in Computer Science*, volume 1855, pages 232–247, 2000.

[20] David Evans, John V. Guttag, James J. Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Symposium on the Foundations of Software Engineering*, December 1994.

[21] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1), January 2002.

[22] R. Feiertag and P. Neumann. The foundations of a provably secure operating system (PSOS). In *Proc. 1979 National Computer Conference*, pages 329–334, 1979.

[23] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, 2002.

[24] Simon N. Foley. Implementing Chinese walls in Unix. *Computers and Security Journal*, 16(6):551–563, December 1997.

[25] Bryan Ford, Mike Hibler, Jay Lepreau, Roland McGrath, and Patrick Tullmann. Interface and execution models in the fluke kernel. In *Proc. 3rd Symposium on Operating System Design and Implementation*, pages 101–115, February 1999.

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

[27] FreeBSD. `http://www.freebsd.org`.

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

[29] John V. Guttag, James J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *LARCH: Languages and Tools for Formal Specification*. Springer-Verlag, New York, NY, 1993.

[30] Thomas A. Henziger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Gregoire Sutre, and Westley Weimer. Temporal-safety proofs for systems code. In *Proc. 14th International Conference on Computer-Aided Verification*, pages 526–538, 2002.

[31] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *POPL '02: Proceedings of the ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, 2002.

[32] IEEE. *IEEE Standard 1003.1-1998: IEEE standard portable operating system interface for computer environments*. Institute of Electrical and Electronics Engineers, 1988.

[33] Sendmail Inc. `http://www.sendmail.org/`.

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

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

[36] Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *Proceedings of 13th USENIX Security Symposium*, San Diego, CA, August 2004.

[37] Key Logic, Inc. *GNOSIS Design Documentation*, 1990.

[38] Larry Koved, Marco Pistoia, and Aaron Kershenbaum. Access rights analysis for Java. In *Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2002.

[39] Ted Kremenek and Dawson Engler. Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. In *Proceedings of the 2003 Static Analysis Symposium*, 2003.

[40] Linux. `http://www.kernel.org`.

[41] Madanlal Musuvathi, David Y.W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. In *OSDI '02: Proceedings of the*

5th Symposium on Operating Systems Design and Implementation, Boston, MA, December 2002.

[42] Jose Nazario. Source code scanners for better code, January 2002. Available at `http://www.linuxjournal.com//article.php?sid=5673`.

[43] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proofs. Technical Report Report CSL-116, Computer Science Laboratory, may 1980.

[44] P.G. Neumann and R.J. Feiertag. PSOS revisited. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003), Classic Papers section*, Las Vegas, Nevada, December 2003.

[45] T. Reps, S. Schwoon, and S. Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *Proceedings of the 10th International Static Analysis Symposium*, San Diego, CA, June 11–13 2003.

[46] Dennis M. Ritchie. Protection of data file contents. United States Patent #4,135,240. Available from `http://www.uspto.gov`.

[47] Secure Software, Inc. RATS. Available at `http://www.securesoftware.com/download_rats.htm`.

[48] Emil Sekerinski and Kaisa Sere, editors. *Program Development by Refinement*. Springer, 1999.

[49] Sendmail Inc. CERT advisory CA-1999-15 buffer overflows in SSH daemon and RSAREF2 library. http://www.cert.org/advisories/CA-1999-15.htm.

[50] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[51] J. S. Shapiro. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, Philadelphia, PA 19104, 1999.

[52] J. S. Shapiro and S. Weber. Verifying the EROS confinement mechanism. In *Proc. 2000 IEEE Symposium on Security and Privacy*, pages 166–176, Oakland, CA, USA, 2000.

[53] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *Proc. 17th ACM Symposium on Operating Systems Principles*, pages 170–185, Kiawah Island Resort, near Charleston, SC, USA, December 1999. ACM.

[54] Solaris. http://www.sun.com/software/solaris/.

[55] Robert E. Strom and Daniel M. Yellin. Extending typestate checking using conditional liveness analysis. *IEEE Trans. on Software Engineering*, (5):478–485, May 1993.

[56] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. on Software Engineering*, (1):157–171, January 1986.

[57] Chris Torek and Casper H.S. Dik. Setuid mess. http://yarchive.net/comp/setuid_mess.html.

[58] John Viega, J.T. Bloch, Tadayoshi Kohno, and Gary McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *Proceedings of ACSAC 2000*, December 2000.

[59] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2000.

[60] David Wheeler. Flawfinder. Available at `http://www.dwheeler.com/flawfinder/`.

[61] John Wilander. A comparison of publicly available tools for static intrusion prevention. In *7th Nordic Workshop on Secure IT Systems*, November 2002.

[62] Michal Zalewski. Multiple local sendmail vulnerabilities. `http://razor.bindview.com/publish/advisories/adv_sm812.html`.

[63] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using CQUAL for static analysis of authorization hook placement. In *Proceedings of the Eleventh USENIX Security Symposium*, August 2002.