Model Checking One Million Lines of C Code*

Hao Chen	Drew Dean	David Wagner
UC Berkeley	SRI International	UC Berkeley
hchen@cs.berkeley.edu	ddean@csl.sri.com	daw@cs.berkeley.edu

Abstract

Implementation bugs in security-critical software are pervasive. Several authors have previously suggested model checking as a promising means to detect improper use of system interfaces and thereby detect a broad class of security vulnerabilities. In this paper, we report on our practical experience using MOPS, a tool for software model checking security-critical applications. As examples of security vulnerabilities that can be analyzed using model checking, we pick five important classes of vulnerabilities and show how to codify them as temporal safety properties, and then we describe the results of checking them on several significant Unix applications using MOPS. After analyzing over one million lines of code, we found more than a dozen new security weaknesses in important, widely-deployed applications. This demonstrates for the first time that model checking is practical and useful for detecting security weaknesses at large scale in real, legacy systems.

1. Introduction

From the dawn of the computing era, software has performed in ways surprising to its authors. In short: software has bugs. Right from the beginning, computer security researchers have recognized that assurance would be a critical challenge for the field, and much work has gone into formal methods for verifying programs. While there have been some successes in the national security arena, formal methods have yet to make it into mainstream software development, be it commercial or open source. And so, software in the 21st century still has bugs, and this seems likely to remain true for the foreseeable future.

All hope, however, is not lost. For instance, a variety of techniques have recently emerged for dealing with the lack of memory safety in the C programming language. Static analysis techniques, with some limitations, can now handle real world programs. And, model checking has provided an alternative paradigm to theorem proving in formal methods.

Model checking has several important advantages. For one, model checking is particularly useful because it tells exactly why a property isn't satisfied, rather than leaving one scratching one's head trying to figure out if the formal proof is failing because the theorem is false, or one is simply insufficiently clever to prove it in the formal system being used. Also, the advent of model checking has introduced a new spin on formal methods: beyond their applications for verification of systems, model checkers are even more useful for finding bugs in systems. Other researchers have reported significant advantages to focusing on bug-checking, at least when looking for non-security bugs [10], and so the time seems ripe to examine how model checking might improve software security.

In this paper, we initiate an empirical study of model checking as a means to improve the security of our software. Our aim is to find problems in software written with benign intent; verifying the absence of bugs is explicitly less important. In this sense, we fit squarely into Jackson's "lightweight formal methods" [13]. Hence, although we take a principled approach (model checking), we are driven more by the pragmatics of checking large software packages for security issues than by theoretical arguments.

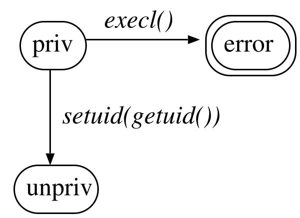
Many security problems in Unix programs are violations of folk rules for the construction of secure programs, especially for `setuid` programs. Some of these rules are amenable to simple local checks, such as those performed by ITS4 [20] and RATS [15]. Others, alas, require searching through all possible control flow paths through the program. Such analysis is painful and error-prone if done by hand. We've built a tool, called MOPS, which was designed and implemented to meet the need for automation to support the security analyst. In this paper, we report on experience with using MOPS to check six large, well-known, frequently used, open source packages (Apache HTTPD, BIND, OpenSSH, Postfix, Samba, and Sendmail) and two small setuid applications (At and VixieCron) in Red Hat Linux 9.

*This work was supported by the Office of Naval Research under contract N00014-02-1-0109, DARPA under contract N66001-00-C-8015, NSF CCR-0093337, and generous donations from Microsoft and Intel.

We have found previously unknown weaknesses in five out of the eight programs that we checked. Our experience has demonstrated that model checking security properties in large, real programs is practical and useful. Though our experiments focus on MOPS, MOPS is just one example of a whole class of model checking tools, and the lessons we learned are likely to be generally applicable to other model checking systems, as well.

2. Overview of MOPS

MOPS is a static (compile-time) analysis tool [6]. Given a program and a security property, MOPS checks whether the program can violate the security property. The security properties that MOPS checks are *temporal safety properties*, i.e., properties requiring that programs perform certain security-relevant operations in certain orders. For example, MOPS might be used to check the following property: a setuid-*root* program had better drop *root* privilege before executing an untrusted program; otherwise, the untrusted program may execute with *root* privilege and therefore compromise the system (see Section 3.1.1. for details on this property). The MOPS user describes a security property by a Finite State Automaton (FSA). Figure 1(a) shows a simplified FSA describing the above property[1] and Figure 1(b) shows a program that violates this property.



(a) An FSA describing this property.

```
// The program has root privilege
if ((passwd = getpwuid(getuid())) != NULL)
{
   fprintf(log, "drop priv for %s", passwd->pw_name);
   setuid(getuid()); // drop privilege
}
execl("/bin/sh", "/bin/sh", NULL); // risky syscall
```

(b) A setuid-*root* program that violates this property. One path of the program satisfies the property, but the other path violates it, giving the user a shell with full privilege.

Figure 1. An FSA describing the property that "*A setuid-root program should drop root privilege before executing an untrusted program***" and a program violating it.**

MOPS checks if a program may violate a temporal safety property using pushdown model checking. Model checking technique exhaustively searches the control flow graph of a program to check if any path may violate a safety property. Pushdown model checking enables searching inter-procedural paths in a context-sensitive manner. If MOPS finds violations, it reports *error traces*, program paths that cause such violations. In general, model checking tends to be best at checking properties that refer primarily to the control flow of the program; in contrast, model checking is usually less well suited for checking data-flow intensive properties.

2.1. Soundness

MOPS strives for soundness, precision, and scalability. A sound tool will not overlook any violations of the security property in the program[2]. A precise tool will have very few false alarms. However, it is challenging to achieve all three criteria at once. Since MOPS is designed to be a practical tool for checking lots of security properties on large programs, it tries to strike a balance between soundness, precision, and scalability. To strive for soundness, MOPS is path sensitive, i.e., it follows every path in the program (including arbitrary number of iterations of loops) except a few minor cases discussed below. MOPS is also context sensitive, i.e., MOPS can match each function return with its call site. To ensure scalability, MOPS takes the approach of sacrificing on the precision of its data-flow analysis rather than sacrificing on scalability. Since data-flow analysis presents many difficulties for scalability, MOPS chooses to be data-flow insensitive. In other words, MOPS ignores most data values in the program and assumes that each variable may take any value[3]. Therefore, MOPS assumes that both branches of a conditional statement may be taken and that a loop may execute anywhere from zero to infinite iterations. As such, MOPS is mostly suitable for properties that are control-flow centric.

MOPS is sound under the following assumptions:

- The program is single-threaded. In other words,

[1] The FSA depicted here considers only one call, `setuid(getuid())`, for dropping *root* privilege and only one call, `execl(...)`, for executing an untrusted program. Other calls are omitted for clarity.

[2] A tool that proves whether a program satisfies a property is *sound* if all the programs that it can prove to satisfy the property do in deed satisfy the property.

[3] MOPS implements limited data flow analysis so that it recognizes the same variable `x` in different expressions such as `x=open()` and `close(x)`.

MOPS is unsuitable for checking concurrent programs.

- The program is memory safe, e.g., no buffer overruns.
- The program is portable. For instance, MOPS does not understand inline assembly code.
- The program does not violate the soundness assumptions required by the property. 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 4. 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.

In addition, the current version of MOPS does not consider control flows that are not in the CFG, such as indirect calls via function pointer, signal handlers, long jumps (`setjmp()/longjmp()`), and libraries loaded at runtime (`dlopen()`). Although these may cause unsoundness, they are implementation limitations rather than intrinsic difficulties.

2.2. Completeness

Since any nontrivial property about the language recognized by a Turing machine is undecidable (Rice's Theorem[16]), no tool that checks a nontrivial property can be both sound and complete. Because MOPS strives to be sound, it is inevitably incomplete in that MOPS may generate *false positive traces*, i.e., program traces that are either infeasible or that do not violate the property. Unfortunately, large numbers of false positive traces would overwhelm the user easily; therefore, it is essential to avoid generating too many false positives. We will discuss how MOPS reduces false positive traces while not sacrificing soundness in Section 5.1.

3. Experiments

We designed the experiments to evaluate three objectives of MOPS. To be useful, MOPS ought to be able to (1) check a variety of security properties; (2) check large programs; and (3) be usable — it should run fast, require a moderate amount of memory, and generate a manageable number of false positive traces.

3.1. 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.

3.1.1. 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[4]. 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`, `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. For further treatment on this topic, we refer the interested reader elsewhere [7].

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

[4]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.

calls are safe. So we check the following property, which is a good approximation of the desired behavior (see Figure 2):

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 2(a)) and the second one describes whether the process has called the `execl`, `popen`, and `system` (Figure 2(b)[5]). 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 2(a)). MOPS automatically computes the the parallel composition of the two FSAs, which represents the integrated property.

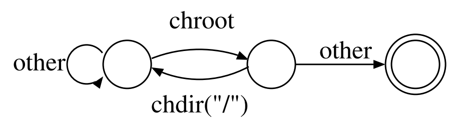
3.1.2. Create Chroot Jails Securely

`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 3(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 3(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.

[5] 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 Property 2

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

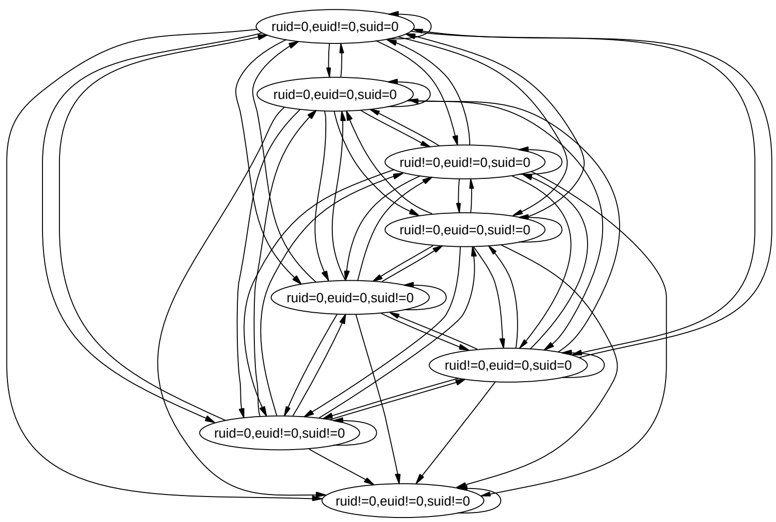
(b) A program segment violating Property 2. Note that the program fails to call `chdir("/")` after `chroot()`, so if `filename` is *"../../etc/passwd"*, a security violation ensues.

Figure 3. An FSA illustrating Property 2 ("*`chroot()` must always be immediately followed by `chdir("/")`*") and a program violating it.

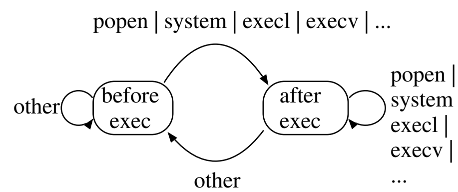
3.1.3. Avoid Race Conditions When Accessing the File System

When a process accesses a file by its name, the process should avoid race conditions, which are potential security vulnerabilities [5]. As an example, consider a privileged process that runs on behalf of a normal user and that wants to constrain itself to access only files owned by the normal user. A naive implementation 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 4(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 conservative approach for detecting race conditions is to check if the program passes the same file name to two



(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 2. 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*.

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*[6].

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`.

3.1.4. 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* 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(a) is installed setuid-*root*. The adversary runs the attack program in Figure 5(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*.

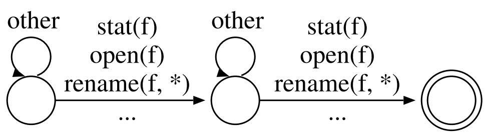
The avoid this attack, a prudent program should observe the following property:

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*[7].

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*.

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

[7] 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.



(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 4. 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.

```
// 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(a)

Figure 5. A program vulnerable to the attack on standard file descriptors and an exploiting program.

3.1.5. 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)` [8] 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.

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

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`.*

3.2. Software Programs

We selected six large, network-related, security-sensitive packages and two small setuid-*root* programs from Redhat Linux 9. See Figure 6 for a list of software packages we analyzed, their descriptions, and the number of lines of code analyzed (counted using "`wc -l *.c`").

3.3. Performance

We ran all the experiments on an 1.5GHz Pentium 4 single-CPU PC with 1GB memory that runs Redhat Linux 9. Figure 7 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 prop-

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 6. Software packages that we checked using MOPS

erties 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 (see Section 5.1.). 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).

3.4. Usability

To evaluate the usability of MOPS, we examine the three stages in the MOPS process: (1) the user describes a security property in an FSA; (2) the user runs MOPS on a software package; and (3) the user analyzes the error traces from MOPS. For the first task, once the user formalizes a security property into a control-flow centric temporal safety property, it is fairly easy to describe it using an FSA. Although one needs to become familiar with the syntax of ASTs before one can write new FSAs, this proved to be an easy task in our experiments – users without compiler background learned how to develop new FSAs quickly by looking at examples and by letting MOPS's parser generate sample ASTs. For the second task, we have written tools for integrating MOPS into the build process of software packages that are built by GCC. Currently, the tools are able to analyze packages distributed as `.tar` files or as source RPM packages. To run MOPS on these packages, the user runs a simple MOPS front-end and provides it with the names of the packages and the names of FSAs, and then the tools take care of the rest of model checking. Note that the user need not modify any Makefile in the packages (see Section 5.2. for details). For the third task, MOPS reports all the errors in the program by listing all error traces that violate the property. Section 5.1. will describe how we improved MOPS so that only one error trace is reported for each

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 7. Running time of MOPS and number of error traces reported by MOPS on five properties

unique programming error. This improvement greatly reduced the number of error traces that we had to examine manually in Figure 7.

To evaluate the usability of MOPS for programmers that are not MOPS developers, we asked three undergraduate and one graduate Computer Science students to run these experiments independently. The students were able to finish the experiments within a few weeks in their spare time, and the learning curve was not too high. This provides evidence that these enhancements to MOPS have made it fairly easy to use.

4. Findings

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

4.1. 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 [7]. 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[9]. *ssh* may need to execute another program, 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 [17], 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
```

[9] *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*.

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.

4.2. 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*.

4.3. 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. Experience and Lessons

Checking real, large software packages requires a great deal of engineering effort. In our experiments, we quickly discovered that we needed to improve MOPS's error reporting and to automate the build process of software packages for MOPS before we could practically analyze large software packages. This forced us to extend MOPS in two key areas: error reporting and build integration.

5.1. Improvement on MOPS's Error Reporting

When MOPS finds potential violations of security properties in a program, it reports error traces, which are useful to the programmers for identifying errors in the program. Since MOPS is conservative, it may report false positive traces, i.e. traces that in fact do not violate security properties but that are misconceived as violations due to MOPS's imprecise analysis. It is left up to the user to decide which error traces are real and which are false positives. In its first implementation, MOPS could only report one error trace for each violated property. Because of this restriction, the presence of a false positive trace effectively prevented MOPS from reporting further, possibly real, traces. To overcome this problem, MOPS must be able to report multiple, and ideally *all*, error traces. However, we also discovered that a single programming error can cause many, sometimes infinitely many, traces, so it is undesirable or impractical to report all of them. We concluded that what we really want is for MOPS to show all programming errors by reporting one error trace as a witness to each programming error. This approach satisfies our seemingly contradictory desires to review all the programming errors and to avoid reviewing redundant error traces.

We consider two different error traces as witnesses to the same programming error if both traces make transitions to an identical error state e from an identical non-error state s at an identical program point p. The unique programming error that both these traces witness is represented by the tuple (e, s, p).[10] Using this definition, for each unique programming error (e, s, p), MOPS searches for only one shortest error trace as a witness and reports it if it exists. This improvement provided orders of magnitude reduction in the number of error traces that the user was forced to examine. For example, it gave us the fairly small number of total traces reported in Figure 7.

This approach is more precise than the one used by Ball et al. for localizing errors in model checking [2]. The major difference is that our approach distinguishes each unique programming error by a unique tuple of a program point, the non-error state before the program point, and the error state after the program point, but their approach distinguishes each error by just a unique program point. It is clear that their approach is less precise because many program points are shared by error traces and correct traces. To illustrate the limitation of their approach, let us look at each of the two alternative algorithms used in their approach. Their first algorithm considers all the program points that are in some error traces but that are absent from any correct traces as error causes. This algorithm has the problem that it will overlook all the error causes that are shared by error traces and correct traces. Their second algorithm, intended to solve the above problem, collects all the tuples (s, p), where p is a program point and s is the state of the program at p, that are in some error traces but that are absent from any correct traces. Then, the algorithm considers all the program points in the collected tuples as error causes. The drawback of this algorithm is that it will mistakenly treat many innocent program points as error causes. To illustrate the problem, let $t = (t_1, t_2, \ldots, t_n)$ be an error trace, where each element in the trace is a tuple of a program point and a program state, and let t_i be the first element where the trace enters an error state. Their algorithm will treat all the program points in the tuples after t_i on the trace as error causes, even though many of them may not be error causes at all. In summary, our approach is more precise in identifying unique error causes.

5.2. Automated Build Process

One of our goals was to make it easy to check many software packages. The core of MOPS consists of three programs:

- *mops_cc*: a parser that takes a file containing C source code and generates a Control Flow Graph(CFG).

[10] What constitutes a unique programming error is subjective and debatable. Here we use just one possible interpretation.

- *mops_ld*: a linker that takes a set of CFG files and merges them into a single CFG file, resolving cross references of external functions and variables.

- *mops_check*: the model checker that takes a CFG and an FSA and decides if the program may violate the security property in the FSA.

Our build integration mechanism is built on top of these three core components. Amusingly, it took us three major iterations before we settled upon an acceptable method for integrating MOPS into the applications' build processes.

First try: running MOPS by hand In our first implementation, we provided no build integration support at all: the user was forced to run the above three programs by hand. This quickly proved unwieldy for packages containing multiple source files and also made it tricky to ensure that *mops_cc* received the same options as the real *cc*.

Second try: modifying Makefile The next thing we tried was to integrate these three programs into the build process of the package manually. As a first attempt, we tried to put these programs into the *Makefile* of each package. Although this works on packages with simple Makefiles (e.g., OpenSSH), it quickly becomes unusable on packages with complicated and multiple Makefiles. Furthermore, rerunning *autoconf* will remove any modifications to Makefiles. Therefore, we soon discovered that we needed an automatic approach for integrating MOPS into the build process.

Third try: interposing on GCC Our solution was to hook into GCC. Most packages use GCC to build executables. By setting the `GCC_EXEC_PREFIX` environment variable, we instruct the GCC front-end to use *mops_cc* as the parser instead of *cc1* and *mops_ld* as the linker instead of *collect2*. So after we build the package, each object file contains a CFG from a single source program file and each executable file contains a merged CFG instead of machine code that GCC would normally build. A nice side benefit of this approach is that we can be sure that MOPS runs on exactly the (preprocessed) code that is compiled, since *mops_cc* runs after the preprocessor (*cpp*).

Refinement: building both CFGs and machine code The above process does not build any machine code. This causes problems in some packages, which build and run executables to generate header files needed for future compilation. Since the above MOPS process replaces executables with CFGs, it causes these packages to fail to build. Similarly, it breaks *autoconf*. An obvious solution is to let *mops_cc*, for each program file *foo.c*, build both a CFG file *foo.cfg* and a machine code file *foo.o*. Also *mops_ld* should link not only the CFG files but also the machine code files.

Refinement: combining each CFG and its machine code into one file The above process, however, causes yet some other packages to fail because of the weak linkage between machine code (*.o* files) and CFGs (*.cfg* files). In some packages, the build process moves an object file, renames it, or puts it into an archive. In such cases, the corresponding CFG file is left dangling and the link to the *.o* file is broken. A possible solution is to modify the `PATH` environment variable to trap the commands that cause the above problems, such as *mv*, *ar*, etc. This, however, is laborious and is hard to make complete. Instead, we adopted an approach that takes advantage of the ELF file format, which allows multiple sections. Since GCC generates object files and executables in ELF, we extended *mops_cc* and *mops_ld* to insert CFGs into a comment section in the appropriate object files and executable files. Then, *mops_check* extracts CFGs from executables for model checking. This way, a CFG is always in the same file as its corresponding machine code, no matter how the build process moves, renames, or archives the file, the fidelity of MOPS's analysis is ensured.

Finally, we wrote a front-end that lets the user run MOPS on packages distributed as `.tar` files or source RPM packages *at the push of a button*. The user simply provides the front-end with the packages and the security properties, and the front-end will invoke the model checker on the packages using their appropriate build processes. This push-button capability has made a significant qualitative difference in our use of MOPS: by removing artificial barriers to use of MOPS, it has freed us up to spend the majority of our time on the code auditing task itself. For packages with special build processes, we can modify their Makefiles to use *mops_cc* as the compiler, use *mops_ld* as the linker, and do model checking immediately after linking. In this way, we have integrated MOPS into the build process of EROS (Extremely Reliable Operating System) [19].

5.3. Value of Tool Support

It is striking that we were able to find so many bugs in these mature, widely used software packages that are probably among the best designed, implemented, audited packages around. Given that MOPS can find security bugs in these programs, our experience suggests that we probably cannot have confidence in the rest of our software.

Static analysis tools like MOPS are valuable in finding vulnerabilities in the programs that run today, but they are even more valuable in preventing vulnerabilities from being inadvertently introduced into programs in the future. Take OpenSSH, for example. In version 2.5.2p2, *ssh* drops all privileges permanently by calling `setuid(getuid())`. In the newer version 3.5p1, however, *ssh* introduces a weakness because it adds a call `seteuid(getuid())` before `setuid(getuid())` and therefore fails to drop privileges completely. Incorporating static analysis tools into the build processes of software packages would help the programmers to catch vulnerabilities as soon as they are introduced into the programs. In addition, in the same way that regression tests prevent old bugs from being re-introduced, MOPS could be used to prevent old security holes from being re-introduced into security-critical applications.

5.4. Configurations Make Static Analysis Difficult

Many packages have build-time configurations and run-time configurations. Build-time configurations affect which code gets compiled and linked. Some of them are set by *autoconf*, which detects the features of the platform, and some can be set by the user. Therefore, one CFG reflects only one build-time configuration, but in some cases only certain configurations result in vulnerable executables. This is often true for programs that make `setuid`-like calls. Because these calls differ on different platforms, many programs put them into conditional macros (`#ifdef`) hoping that they work correctly on every platform. However, MOPS can only check the package for the property on the platforms where the package has been built.

Run-time configurations affect the control flow of the program — some control flows are feasible only in some configurations. Since MOPS lacks data flow analysis, run-time configuration causes false positives. Furthermore, there are often constraints on the parameters in a configuration. Therefore, even a data-flow savvy tool cannot avoid some false positives without domain knowledge of each configuration. This demands more human assistance. Therefore, configuration-dependent code remains a weak spot for software model checking, static analysis, and formal verification in general.

6. Related Work

A number of static analysis techniques have been used to detect specific security vulnerabilities in software. Wagner et al. used integer range analysis to find buffer overruns [21]. 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 [14]. CQUAL [11] 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 [18] and to verify authorization hook placement in the Linux Security Model framework [23], which are examples of the development of sound analysis for verification of particular security properties. The application of CQUAL, however, is limited by its flow insensitivity and context insensitivity, although it is being extended to support both.

Metal [9, 1] is a general tool that checks for rule violations in operating systems, using meta-level compilation to write system-specific compiler extensions. Use of Metal basically requires the programmer/security analyst to specify annotations on the program being studied. Metal then propagates these annotations and is very good at finding mismatches: *e.g.*, when a pointer to a user-mode address is dereferenced inside the kernel. Over time, Metal has evolved from a primarily intra-procedural tool to an inter-procedural tool. Recent work [22] has been on improving the annotation process. The most substantial difference between Metal and MOPS are Metal's annotations: with more effort, Metal can find data-driven bugs that MOPS currently cannot. Overall, the approaches of MOPS and Metal are converging. One could easily recommend the use of both tools to analyze a particular piece of software, with each tool used where it has particular strength. Moreover, we expect that the results of this paper would carry over to Metal; all of the security properties we considered can be expressed within Metal's annotation system, and it seems likely that most (or all) of the bugs we found with MOPS could also have been found with Metal.

SLAM [3, 4] is a pioneer project that uses software model checking to verify temporal safety properties in programs. It validates a program against a well designed interface using an iterative process. During each iteration, a model checker determines the reachability of certain states in a boolean abstraction of the source program and a theorem prover verifies the path given by the model checker. If the path is infeasible, additional predicates are added and the process enters a new iteration. SLAM is very precise, and as such, represents a promising direction in lightweight formal verification. SLAM, however, does not yet scale to very large programs. Similar to SLAM, BLAST is another software model checker for C programs and uses counterexample-driven automatic abstraction refinement to construct an abstract model which is model checked for safety properties [12]. Compared to SLAM and BLAST, MOPS trades precision for scalability and efficiency by considering only control flow and ignoring most data flow, as we conjecture that many security properties can be verified without data flow analysis. Also since MOPS is not an iterative process, it does not suffer from possible non-termination as SLAM and BLAST do. ESP [8] 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. ESP is between SLAM/BLAST and MOPS in scalability and precision.

A key contribution to the usability of MOPS is its ability to report only one error trace for each error cause in the program, which significantly reduces the number of error traces that the programmer has to review. Among the tools discussed above, only SLAM has documented a similar ability [2]. Compared to MOPS's approach, SLAM's approach is less precise because it may overlook error causes or report spurious error causes (see Section 5.1. for a detailed discussion).

Despite important technical differences, we believe these diverse approaches to software model checking share significant common ground, and we expect that much of our implementation experience would transfer over to other tools (such as Metal, ESP, SLAM, or BLAST). This makes our results all the more meaningful.

7. Conclusion

We have shown how to use software model checking to analyze the security of large legacy applications. We reported on our implementation experience, which shows that model checking is both feasible and useful for real programs. After performing extensive case studies and analyzing over one million lines of source code, we discovered over a dozen security bugs in mature, widely-deployed software. These bugs escaped notice until now, despite many earlier manual security audits, because the security properties were somewhat subtle and non-trivial to check by hand. This demonstrates that MOPS allows us to efficiently and automatically detect non-obvious security violations. Our conclusion is that model checking is an effective means of improving the quality of security-critical software at scale.

Acknowledgment

We are grateful to David Schultz, who built a first prototype of the automated build process and who implemented a first version of the *stderr* property; and to Geoff Morrison, who helped improve the automated build process and who implemented the MOPS user interface for reviewing error traces in HTML formats. Ben Schwarz, Jacob West, Jeremy Lin, and Geoff Morrison helped verify many results in this paper after the initial submission of this paper. We also thank Peter Neumann, Jim Larus, and the anonymous reviewers for their comments on an earlier draft of this paper.

References

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

[2] T. Ball, M. Naik, and S. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *POPL '03: Proceedings of the ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, 2003.

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

[4] T. Ball and S. 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.

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

[6] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, 2002.

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

[8] M. Das, S. Lerner, and M. 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.

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

[10] D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *5th Intl. Conference Verification, Model Checking and Abstract Interpretation (VMCAI '04)*, 2004.

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

[12] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *Proceedings of the 10th SPIN Workshop on Model Checking Software*, 2003.

[13] D. Jackson and J. Wing. Lightweight formal methods. *IEEE Computer*, pages 21–22, April 1996.

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

[15] RATS. `http://www.securesoftware.com/rats.php`.

[16] H. Rogers. *Theory of Recursive Functions and Effective Computability*. MIT Press, 1987.

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

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

[19] J. Shapiro, J. Smith, and D. Farber. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, 1999.

[20] J. Viega, J. T. Bloch, T. Kohno, and G. McGraw. Its4: A static vulnerability scanner for c and c++ code. In *Proceedings of the 16th Annual Computer Security Applications Conference*, December 2000.

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

[22] J. Yun, T. Kremenek, Y. Xie, and D. Engler. Meca: an extensible, expressive system and language for statically checking security properties. In V. Atluri and P. Liu, editors, *Proceedings of the 10th ACM Conference on Computer and Communication Security*, pages 321–334, Washington, DC, October 2003. ACM.

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