MOPS*User's Manual

Hao Chen David Wagner David Schultz
Computer Science Division, UC Berkeley
{hchen,daw,dschultz}@cs.berkeley.edu

October 8, 2002

1 Introduction

MOPS is a tool for finding security bugs or verifying their absence in C programs. These bugs violate *temporal safety properties*, which require that the program must perform certain operations in specific orders. For example, a setuid-*root* process on Unix systems should not execute an untrusted program (e.g., by making the system call *execv()*) before dropping the *root* privilege in its effective user ID (e.g., by calling *seteuid(getuid())*); otherwise, a malicious user may ask the process to execute a program of her choice and then gain the *root* privilege through the program. In another example of prudent coding practice, the call *strncpy(dst, src, n)* in a C program should be followed immediately by the statement *dst[n]='\0'*; otherwise, the array `dst` may not be null-terminated, which makes it vulnerable to buffer overrun attacks. As in these examples, many security properties may be described by temporal safety properties. If a program violates such properties, it is often vulnerable to attacks. Manually checking these properties on every path in a program, however, is often tedious or impossible. MOPS automates this process using model checking, a static analysis technique. MOPS is sound, i.e., it is able to verify that no path in a program may violate a security property, under some mild assumptions (Section 6).

The usage of MOPS is straight forward. First, the user describes a security property by a Finite State Machine (FSA) (Section 5). Then, he runs MOPS to check this property on a C source program. If MOPS determines that the program may violate the property, it prints out an offending path in the program.

This manual is organized as follows. Section 2 describes how to install and build MOPS. Section 3 provides a quick example on how to use MOPS. Section 4 describes the MOPS process in details. The current release of MOPS contains just the software for checking properties. In the future, we hope to collect a database of properties of common interest, but at present, such properties must be identified and codified manually by the user of MOPS. Section 3.3 gives a tutorial on how to express security properties as FSAs and section 5 provides the specification of FSAs. Finally, Section 6 discusses the soundness of MOPS and provides further readings on MOPS.

2 Installation

2.1 System Requirements

MOPS consists of a C program parser and a model checker. The parser is derived from *gcc* and is written in C. It has been tested on sparc/solaris (Solaris 2.6 and Solaris 7) and x86/Linux machines (Redhat 6.2 and Redhat 7). The model checker is written in Java and has been tested on Java 1.3.1 and Java 1.4.

*MOPS: MOdel checking Programs for Security properties

2.2 Installation

1. untar/unzip

```
tar zxvf mops.tar.gz
```

2. compile

```
cd mops
make
```

3. test

```
cd test
make
```

3 A Quick Example

In this section, we will show how to use MOPS through a quick example. First, we will describe a security property. Then, we will use MOPS to check this property in a program. Since MOPS requires the security property to be described by an FSA, we will show how to construct such an FSA finally.

3.1 Security Property

In this example, we consider a security requirement at the time when a privileged process executes an untrusted program in Unix systems. Each Unix process has three user IDs: the real user ID (real *uid*), the effective user ID (effective *uid*), and the saved user ID (saved *uid*). The real user ID identifies the owner of the process and the effective user ID carries access permission[1]. When a process has a zero effective uid, it is privileged (e.g., it has full access to the file system). When a user executes an ordinary (non-setuid) program, both the real uid and the effective uid of the process assume the user's ID. However, when a user executes a *setuid-root* program, the real uid of the process is still the user, but the effective uid of the process becomes zero. Since a setuid-root process is privileged, it should be scrupulous when doing potentially dangerous operations, such as calling *execv()* to execute an untrusted program.

A security property requires that a setuid-root process with effective uid zero should not call *execv()* to run an untrusted program. This is because the effective uid is often inherited during *execv()*. If a process violates this property, it may allow the untrusted program to run with privilege and to eventually compromise the system. This property is described by the FSAs in Figure 1. We will show how to create such FSAs in Section 5.

3.2 Running MOPS

The program in Figure 2 runs as a setuid-root process. It tries to drop the root privilege before executing an untrusted program. We will now use MOPS to check if the program satisfies the security property in Figure 1. To run the following commands, you should be in the directory `mops/test` and set the environment variable `CLASSPATH` to `:../src/class:../lib/java-getopt-1.0.9.jar`.

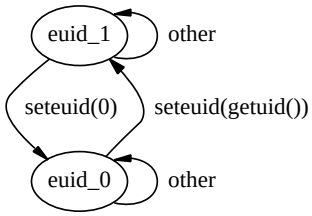
1. Parse the C program `hello.c` into a Control Flow Graph (CFG) `hello.cfg`:

```
gcc -B ../rc/ -S -o hello.cfg hello.c
```

[1] We won't discuss the saved user ID here and we refer the interested users elsewhere [1].

```
t euid_0 euid_1 { function_call { identifier seteuid } { function_call {
identifier getuid } } }
t euid_0 euid_0 { other }
t euid_1 euid_0 { function_call { identifier seteuid } { lexical_cst 0 } }
t euid_1 euid_1 { other }
```

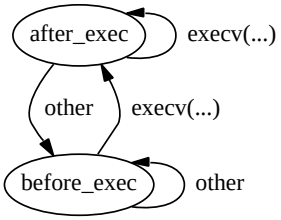
(a) An FSA describing the transition of the effective user ID in a setuid-root process



(b) Graphical representation of the FSA in Figure 1(a)

```
t before_exec after_exec { function_call { identifier execv } { ellipsis } }
t before_exec before_exec { other }
t after_exec after_exec { function_call { identifier execv } { ellipsis } }
t after_exec before_exec { other }
```

(c) A FSA describing a process calling *execv*



(d) Graphical representation of the FSA in Figure 1(c)

```
setuid.fsa exec.fsa
q euid_0 before_exec
f euid_0 after_exec
```

(e) A meta-FSA describing the composition of the FSAs in figure 1(a) and Figure 1(c)

Figure 1: Formal representation of the security property that a setuid-root process should not call *execv* with the effective uid zero

```
void drop_priv()
{
  struct passwd *passwd;

  if ((passwd = getpwuid(getuid())) == NULL)
  {
    printf("getpwuid() failed");
    return;
  }
  printf("Drop user %s's privilege\n", passwd->pw_name);
  seteuid(getuid());
}

int main(int argc, char *argv[])
{
  drop_priv();
  printf("About to exec\n");
  execv(argv[1], argv + 1);
}
```

Figure 2: A program violating the security property in Figure 1

This command instructs *gcc* to: (1)preprocess `hello.c` using *cpp*; and (2) parse `hello.c` using MOPS's parser *cc1* in the directory `../rc/` and write the CFG into `hello.cfg`. Don't forget the trailing slash after the directory name in `../rc/`; it is needed. The `-B` option specifies the directory where MOPS's parser *cc1* is located. The `-S` option tells *gcc* to stop after parsing.

2. Compact the CFG `hello.cfg` into a smaller CFG `hello.s.cfg`:

   ```
   java CfgCompact setuidexec.mfsa hello.cfg main hello.s.cfg
   ```

   Often we can drastically reduce the size of a CFG by removing most statements from its C program that are irrelevant to a security property. This is called compacting a CFG. The above command does just that, where `setuidexec.mfsa` is the security property illustrated in Figure 1 and `main` is the entry function in the CFG.

3. Model check the security property `setuidexec.mfsa` on the compacted CFG `hello.s.cfg`:

   ```
   java Check setuidexec.mfsa hello.s.cfg main hello.s.tra
   ```

   In the above command, `main` specifies the entry function in the CFG. If the model checker finds a program path that violates the security property (often called an *error trace*), it outputs the message "`The property is violated in the program`" and writes the error trace into `hello.s.tra`. The best way to understand the violation is to inspect the trace; however, since the trace is from the compacted CFG (because the model checker runs on the compacted CFG), we have to transform it into the corresponding trace in the C program.

4. Transform the trace from the compacted CFG (`hello.s.tra`) into a trace in the C program (`hello.tra`):

   ```
   java Transform hello.cfg hello.s.tra hello.tra
   ```

   The transformed trace file (`hello.tra`) look like the following:

```
...
hello.c:19: <euid_0 , before_exec>  1
hello.c:21: <euid_0 , before_exec>  1
hello.c:6: <euid_0 , before_exec>  2
hello.c:8: <euid_0 , before_exec>  2
...
```

Each line consists of three fields, which describe the execution of a statement in the program. The first field shows the location of the statement by its file name and line number. For example, `hello.c:19` refers to the statement on Line 19 in the file `hello.c`. The second field shows the state of the program just before the statement is executed. For example, the state `<euid_0 , before_exec>` refers to the state with the same name in the FSA `setuidexec.mfsa` (Figure 1(e)). The third field shows the depth of the call stack just before the statement is executed – the depth is 1 inside the entry function of the program.

5. View the error trace (`hello.tra`):

   ```
   emacs hello.tra
   ```

   For convenience, the error trace is written in the same format as a compilation log. The user can load the trace into *emacs* and use the command *next-error* to step through each line of it while *emacs* displays the corresponding statement in the source file.

   If you're not familiar with *emacs*, this means that typing `C-x ‘` repeatedly steps you through the lines of the error trace. (Here the notation `C-x ‘` represents typing Control-x followed by the back-apostrophe key.) The top half of the window will show the trace file (with the current line at top), and the bottom half of the window will show the C source file (with the cursor positioned on the current line of source code).

   If you follow the execution trace in this example, you will see that this trace represents an execution where `drop_priv()` is called, the else branch of the `if` statement is followed, the code returns back to `main()`, and then the `exec()` statement is executed. This is a violation of the security property, because we have never dropped privilege, as is indicated by the fact that the last line of the trace file `hello.tra` indicates we have entered state `<euid_0 , after_exec>`. In short, we have found a security vulnerability in the example program: if the program fails to get a `passwd` entry, then it fails to drop privilege before calling *exec*, which is probably not what was intended.

   Note that the model checker is imprecise: it might sometimes yield false alarms. This typically happens where `Check` outputs a path that appears to violate the security property but in fact is not a feasible path in the program. Thus, when viewing the path, keep an eye out to watch whether the resulting path is feasible, or whether it represents a false alarm. In this example, the path represents a real bug.

3.3 Constructing an FSA

We will now show how to use tell MOPS about a security property of interest to us. If we want to check a program for some class of bugs, the general strategy will be to identify a security property that correct programs should satisfy. Then, we negate the property and write down a regular expression expressing the set of all traces that violate the security property, and we convert this into an FSA. Finally, we encode the FSA using the MOPS syntax, as described below.

We will motivate this process by way of an example: we will construct an FSA for the security property that a setuid-root process should not call *execv()* while its effective uid is zero. Each FSA has a set of states

and a set of transitions. We create an FSA to describe the transition of the effective uid between zero and non-zero in a setuid-root process, shown in Figure 1(b). The FSA has two states, one representing the value zero and the other a non-zero value in the effective uid. The FSA makes transitions between these two states by the system calls *seteuid(0)* and *seteuid(getuid())*. Note the transitions with the special label *other*. They are a form of wildcard: if the FSA is at the state s and the C program is about to execute a statement e, but no label on any outgoing transition from the state s matches the statement e, then the FSA follows the transition labeled *other* from the state s.

Similarly, we create another FSA to describe whether a process has just called *execv*, as shown in Figure 1(d). The FSA has two states, one representing that the process has just called *execv* and the other representing otherwise.

To describe the security property, we combine the two FSAs above to create a *meta-FSA* shown in Figure 1(e). A meta-FSA describes an FSA that is the product of multiple FSAs. The state space of the meta-FSA is the Cartesian product of the state spaces of all the FSAs. A meta-FSA file has three sections. The first section specifies the names of FSAs whose product is this meta-FSA. The second section specifies the initial states of the meta-FSA, and the third section specifies the final states of the meta-FSA. In this example, the meta-FSA file `setuidexec.mfsa` consists of the following lines:

- `setuid.fsa exec.fsa`

  This line specifies the FSAs, `setuid.fsa` and `exec.fsa`, whose product is this meta-FSA.

- `q euid_0 before_exec`

  This specifies that an initial state of the meta-FSA is a tuple consisting of the state *euid_0* from the first FSA (`setuid.fsa`) and the state *before_exec* from the second FSA (`exec.fsa`).

- `f euid_0 after_exec`

  This specifies that a final state of the meta-FSA is a tuple consisting of the state *euid_0* from the first FSA (`setuid.fsa`) and the state *after_exec* from the second FSA (`exec.fsa`). The final state of the FSA corresponds to the accept state, i.e., the resulting meta-FSA will accept all traces that end at the final state, and the idea is that these traces should be exactly the bad ones that violate the desired security property.

4 Using MOPS

The MOPS process consists of the following steps: parse each source file into a CFG, merge multiple CFGs into a single one, compact the merged CFG, model check the compacted CFG, and transform the error traces, if any, into a more user-friendly format.

4.1 Parse Source Program

When *gcc* compiles a file, it calls the following programs sequentially:

- *cpp*: The C preprocessor transforms a C source file `foo.c` into a preprocessed file `foo.i`.
- *cc1*: The parser transforms a preprocessed file `foo.i` into an assembly file `foo.s`.
- *as*: The assembler transforms an assembly file `foo.s` into an object file `foo.o`.

MOPS's parser *cc1* is derived from *RC*'s parser [2], which is derived from *gcc*'s *cc1*. As such, MOPS's *cc1* accepts most command line options to GCC's *cc1* and is able to parse most programs that *gcc* parses. MOPS's *cc1* accepts a C source file and outputs the CFG of the file. If the user supplies a `-o filename` option, the CFG file is named `filename`; otherwise, the CFG file is named after the C source file with the suffix `.s` replacing the suffix `.c` (e.g. the CFG file for the source file `foo.c` will be named `foo.s`).

The user can parse a C source file `foo.c` into a CFG file `foo.cfg` using MOPS in two alternative ways:

- First, preprocess `foo.c` into `foo.i`:

  ```
  cpp foo.c
  ```

  Then, parse `foo.i` using MOPS's cc1:

  ```
  rc/cc1 -o foo.cfg foo.c
  ```

- Alternatively, the user may let *gcc* call *cpp* and MOPS's *cc1* automatically:

  ```
  gcc -B rc/ -S -o foo.cfg foo.i
  ```

  The `-B` option supplies *gcc* with additional paths to look for the executables *cpp*, *cc1*, and *as*. Since only *cc1* is in the directory `rc/`, *gcc* will use the default *cpp* to preprocess the C source file and then use MOPS's *cc1* in the directory `rc/` to parse it. Note that the directory argument to the `-B` option must ends with the character `/`. The `-S` option tells *gcc* to stop after parsing.

4.2 Merge Multiple CFGs

MOPS's parser generates one CFG file for each C source file. If a program consists of multiple C source files, their CFG files have to be merged into a single CFG file. MOPS offers a program, *CfgMerge*, for this task. The following command merges the CFG files *foo1.cfg*, *foo2.cfg*, and *foo3.cfg* into *foo.cfg*:

```
java CfgMerge foo.cfg foo1.cfg foo2.cfg foo3.cfg
```

4.3 Compact CFG

As C source files becomes larger, the sizes of their CFGs increase rapidly, which can easily overwhelm the model checker in time and space requirements. Fortunately, most CFGs can be compacted into much smaller ones in such a way that the model checker will always generate the same result on the CFGs before and after compaction. The compaction is based on the observation that, for most CFGs and most security properties, most statements in the CFG are irrelevant to the security property and therefore can be safely removed. For example, if the security property checks only for the *seteuid* and *execv* system calls, all the other statements in the CFG are irrelevant. The program `CfgCompact` compacts a CFG. The following command compacts the CFG `foo.cfg` with regard to the security property `model.mfsa` and writes the compacted CFG to `foo.s.cfg` (the argument `main` specifies the entry function of the CFG):

```
java CfgCompact model.mfsa foo.cfg main foo.s.cfg
```

Note: Compacting CFGs is mandatory, even if the CFG is small, because it does other transformations that the subsequent steps depend on.

4.4 Model Check CFG

The model checker determines whether the C source program of a CFG satisfies a security property. It achieves this by determining whether any final state in the FSA that describes the security property, called the *security model*, is reachable in the program — if so, the program may violate the security property. The following command checks the CFG `foo.s.cfg` for the security property described in `model.mfsa`:

```
java Check model.mfsa foo.s.cfg entry_function_name foo.s.tra
```

where `entry_function_name` is the entry function of the program (usually `main`). If the program may violate the property, the model checker writes an error trace into `foo.s.tra`. Since often there are many such traces, the model checker tries to find the shortest one.

In addition to an error trace, the model checker can provide another output: it can identity all the points in the program where the program may be executed in a final state of the security model. In other words, it will show all program statements where the security property might be violated, without producing an error trace showing how to get there. To get all such program points into the file `foo.s.lst`, run the following command:

```
java -p Check model.mfsa foo.s.cfg entry_function_name foo.s.lst
```

4.5 Transforming the Trace

Since the error traces or program points generated by the model checker come from the compacted CFG, the user needs to transform them into the ones in the original CFG to understand them.

To transform an error trace, run

```
java Transform foo.cfg foo.s.tra foo.tra
```

where `foo.cfg` is the original CFG, `foo.s.tra` is the error trace generated by the model checker, and `foo.tra` is the corresponding trace in the original CFG. The transformed traces have the same format as compilation logs from compilers.

The user can load the trace into *emacs* and use the command *next-error* to step through each line of it while *emacs* displays the corresponding statement in the source file. If you're not familiar with *emacs*, this means that typing `C-x ‘` repeatedly steps you through the lines of the error trace. (Here the notation `C-x ‘` represents typing Control-x followed by the back-apostrophe key.) The top half of the window will show the trace file (with the current line at top), and the bottom half of the window will show the C source file (with the cursor positioned on the current line of source code).

If you use *vim*, run `vim -q foo.tra"`, and type `:cn` repeatedly to cycle through the lines in the trace file. You may map the function key `F3` to the `next-line` functionality by adding

```
map #3 :cn^V^M
```

to your /.vimrc file (here `^X` represents a control-`X` character, which can be entered in *vim* by typing control-`V` control-`X`). Type `:h quickfix` within a *vim* window for more on the `next-line` functionality.

To transform program points, run

```
java Transform -p foo.cfg foo.s.lst foo.lst
```

where `foo.cfg` is the original CFG, `foo.s.lst` contains the program points generated by the model checker, and `foo.lst` contains the corresponding program points in the original CFG.

4.6 Visualization Tools

MOPS offers tools for visualizing CFGs and FSAs.

4.6.1 Cfg2Dot

Cfg2Dot transforms a CFG into a graphical representation in the *dot* format, which can be transformed into the postscript format by Graphviz [3].

```
java Cfg2Dot [-a] foo.cfg foo.dot
```

The option `-a` instructs *Cfg2Dot* to also include ASTs in the graph.

4.6.2 Fsa2Dot

Fsa2Dot transforms an FSA into a graphical representation in the *dot* format.

```
java Fsa2Dot [-l] fsa_file dot_file
```

The option `-l` specifies that labels (ASTs) on the transitions in the FSA should be included in the graph.

5 FSA Representation of Security Properties

MOPS uses Finite State Automata to describe security properties.

5.1 Finite State Automaton

The user describes an FSA in a text file. When MOPS processes the file, it ignores empty lines, C style comments (`/* ...*/`), and C++ style comments (`//`). A line in the file describes a transition in the FSA using the following format:

```
t state_before state_after ast
```

where `state_before` is the starting state, `state_after` is the ending state, and `ast` is the Abstract Syntax Tree (AST) that represents the statement that triggers the transition (to be described in Section 5.3).

5.2 Meta-FSA

Often it is convenient to decompose a complex security property into a set of simpler ones. MOPS allows the user to describe each simpler property by an FSA. Then, the user describes the complex property by way of these FSAs in a *meta-FSA*.

A meta-FSA describes an FSA that is the product of multiple FSAs. The first line in the meta-FSA file enumerates each of the multiple FSAs:

```
file1.fsa file2.fsa file3.fsa ...
```

After the first line, each subsequent line describes an initial state or final state of the meta-FSA. The state space of the meta-FSA is the Cartesian product of the state spaces of the multiple FSAs. The following line describes an initial state:

```
q state1 state2 state3 ...
```

where `state1` is a state in `file1.fsa`, `state2` is a state in `file2.fsa`, and so on. The following line describes a final state:

```
f state1 state2 state3 ...
```

MOPS requires the user to build a meta-FSA in all cases, even if the security property can be simply described in a single FSA, in which case the meta-FSA contains only one FSA.

5.3 Abstract Syntax Tree

An Abstract Syntax Tree (AST) represents a statement in a program. The following defines the grammar of the AST:

```
ast ::= { kind content+ }

kind ::= real_kind | meta_kind

real_kind ::=  function_decl | variable_decl | ...

meta_kind ::=  not | or | any| ellipsis | var | other

content ::= ast | string
```

For example, the AST for the statement `setuid(0)` is:

```
{ function_call { identifier setuid } { lexical_cst 0 } }
```

The file `Ast.java` in the directory `src` contains a list of all the values for `real_kind`.

5.3.1 Meta Kinds

MOPS allows the user to specify *meta kinds*. Five of them, `not, or, any, ellipsis`, and `other`, are discussed here. The last one, `var`, will be described in Section 5.3.2.

- `not`. It represents the logical *not*. For example, the following AST matches `setuid(x)` where `x` is not `0`:

  ```
  { function_call { identifier setuid } { not { lexical_cst 0 } } }
  ```

- `or`. It represents the logical *or*. The following AST matches `setuid(0)` or `seteuid(0)`:

  ```
  { function_call { or { identifier setuid } { identifier seteuid } }
    { lexical_cst 0 } }
  ```

- `any`. It matches any single AST. The following AST matches `execv(''rm'', x)` where `x` is an arbitrary expression:

  ```
  { function_call { identifier execv } ''rm'' { any } }
  ```

- `ellipsis`. It matches zero or more ASTs. The following AST matches `printf(...)`:

  ```
  { function_call { identifier printf } { ellipsis } }
  ```

```
t closed opened { binary "=" { var x } { function_call { identifier
"open" } } }
t opened closed { function_call { identifier "close" } { var x } }
```

(a) An FSA that describes the `open` and `close` calls using a pattern variable

```
int main()
{
    int fd1, fd2;
    fd1 = open("/etc/motd", O_RDONLY);
    fd2 = open("/etc/passwd", O_RDONLY);
    close(fd1);
    close(fd2);
}
```

(b) A C program that makes the `open` and `close` calls

Figure 3: An FSA and a C program that demonstrate the utility of pattern variables

- `other`. This is analogous to the *default* label in a *switch* statement in C. When a state has an *other* transition, a transition whose AST is { `other` }, and no other transition from the same state matches a statement, the FSA takes the *other* transition. For example, when the following FSA is in the state `euid_1`, it will makes a transition to the state `euid_0` if the statement is `setuid(0)`, because the statement matches the first transition; however, the FSA will stay in same state for all other statements because they match the *other* transition.

```
t euid_1 euid_0 { function_call { identifier setuid } { lexical_cst 0 } }
t euid_1 euid_1 { other }
```

5.3.2 Pattern Variables

The last meta-kind, `var`, is used for *pattern variables*. An AST {`var x`} is a pattern variable, which matches any expression. For example, suppose we want to build an FSA that describes the opening and closing of files. This FSA has two states: a state `opened` and a state `closed`. The FSA makes a transition from the state `closed` to the state `opened` when the program executes a statement `x = open(...)` where `x` is a file descriptor. When the program closes the same file descriptor by calling `close(x)`, the FSA makes a transition from `opened` to `closed`[2]. Note that the special variable `x` in the two statements `x = open(...)` and `close(x)` serves as a wildcard — it doesn't refer to a variable named `x` in the program. Rather, it is ready to match any expression in that position in the two statements in the program. Such special variables are called pattern variables: their ASTs look like those of ordinary variables except that their kind is `var` (in contrast, the kind of ordinary variables is `identifier`). Figure 3(a) shows the FSA that we have just discussed.

[2]We do not consider variable aliasing, i.e. assign the file descriptor to another value, the alias, and then close the alias.

If an FSA contains pattern variables, MOPS instantiates the FSA by replacing the pattern variables with the actual variables in the program that the pattern variables match, possibly resulting in multiple instantiations of the FSA. For example, when MOPS instantiates the FSA in Figure 3(a) on the program in Figure 3(b), it finds that the pattern variable `x` may match the variables `fd1` and `fd2` in the program. Therefore, MOPS instantiates the FSA into two FSAs. Then, MOPS checks the program against each FSA instance separately.

The scope of a pattern variable is the FSA file. In other words, when a meta-FSA contains multiple FSAs, pattern variables from different FSAs with the same name do not collide.

5.4 FSA Transitions

The user may specify multiple transitions from a state in an FSA. When MOPS determines which transition the FSA should take after a statement is executed in the program, it examines all the transitions from the state except the *other* transition in the order that they are specified in the FSA file. During the process, if the statement matches the AST on an transition, the FSA takes that transition. If the statement does not match the AST on any transition, but if an *other* transition is present, it is taken; if no *other* transition is present, the FSA stays in the same state. Therefore, although the user may specify multiple, arbitrary transitions from a state, the FSA is deterministic.

The CFG compaction algorithm requires that no FSA should use the AST { `any` }, because this AST is superseded by the *other* transition, whose AST is { `other` }. Furthermore, whenever the FSA follows multiple *other* transitions consecutively, it should end in the same state as if it follows just one *other* transition. In other words, every state that has one or more incoming *other* transitions should not have any outgoing *other* transition.

5.5 Automatic Generation

Manual construction of FSAs may be tedious and error-prone for complex properties. Sometimes the user may leverage the operating system to build FSAs automatically. The details are described elsewhere [1].

6 Discussion

6.1 Soundness

The *soundness* of a program analysis tool means that it will not miss any bugs in a program that are of the types specified by the user. MOPS is sound if the following requirements are satisfied:

- The program is a portable, single-threaded C program that has no implementation-defined behavior: for example, no buffer overruns and no runtime code generation.
- The program has no function pointers, signal handlers, and non-local jumps via setjmp/longjmp, because MOPS ignores them when building CFGs. Although this approximation introduces unsoundness, it is not a fundamental limitation of the approach but rather a limitation of the current implementation. We can overcome this problem by manually transforming the control flow that MOPS ignores to the equivalent ones that MOPS considers. We are working on automating this process and we hope to add it to a future version of MOPS.

6.2 Further Readings

The internal mechanism of MOPS is described in two papers [4, 5].

7 Availability

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

References

[1] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *Proc. of the Eleventh Usenix Security Symposium*, San Francisco, CA, 2002.

[2] David Gay. Region based compiler. `http://www.cs.berkeley.edu/~dgay/rc/`.

[3] AT&T Labs Research. Graphviz. `http://www.research.att.com/sw/tools/graphviz/`.

[4] Hao Chen and David Wagner. MOPS: an infrastructure for examining security properties ofsoftware. Technical Report UCB//CSD-02-1197, UC Berkeley, 2002.

[5] Hao Chen and David Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communication Security*, Washington, DC, November 2002.