Pushdown model checking for security

David Wagner
U.C. Berkeley

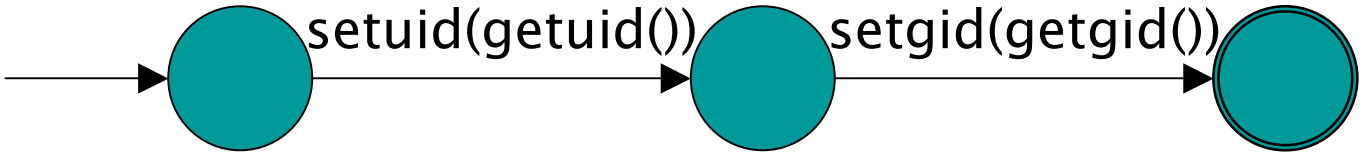
Work by Hao Chen, Ben Schwarz,
and Drew Dean, Jeremy Lin, Geoff Morrison,
David Schultz, Wei Tu, Jacob West

Outline

- Introduction
- Pushdown model checking: theory, background, intuition
- Security properties
- MOPS, a tool for pushdown model checking
- Experience with MOPS

Introduction to MOPS

- Pushdown model checking of C source code
- Security properties expressed as finite state automata (temporal safety properties)

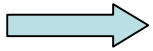


Example: A simplistic FSA to detect dropping privilege in the wrong order.

Pushdown models of C programs

- Abstraction of C program as a pushdown automaton
  - PDA stack = program call stack
    = (pc, retaddr_1, retaddr_2, ...)
  - Models control flow behavior of program (only)
  - Branch statements modelled non-deterministically

```
void f(int n) {
    setuid(getuid());
    g("foo");
    if (n>0) {
        f(n-1);
    }
}
```

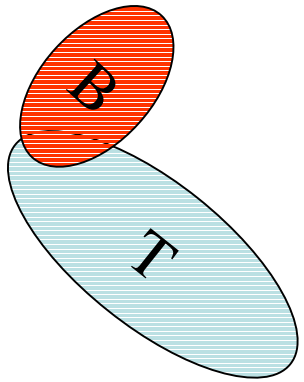


F ::= setuid(...) W
W::= G X
X ::= Y | Z
Y ::= F Z
Z ::= ε

Pushdown model checking

- Σ = set of security operations
  - e.g., Σ = {setuid(getuid()), setgid(getgid())}
- $B \subseteq \Sigma^*$: sequences of ops that violate the property
  - B is specified by a FSA, and thus regular
- $T \subseteq \Sigma^*$: set of feasible traces (sequences of ops) from the program
  - T is a pushdown model, and thus is context-free

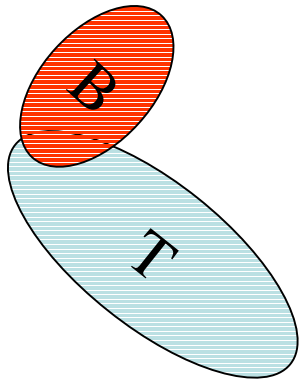
- Question: Is $B \cap T = \emptyset$?



Pushdown model checking

- Σ = set of security operations
  - e.g., Σ = {setuid(getuid()), setgid(getgid())}
- $B \subseteq \Sigma^*$: sequences of ops that violate the property
  - B is specified by a FSA, and thus regular
- $T \subseteq \Sigma^*$: set of feasible traces (sequences of ops) from the program
  - T is a pushdown model, and thus is context-free

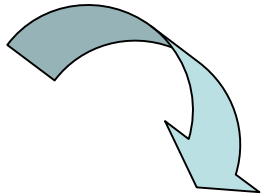
- Question: Is $B \cap T = \emptyset$?
- Algorithm:
  1) Compute the intersection (a CFL)
  2) Test for emptiness



Intersecting a CFL and a FSA

F ::= setuid W
W::= G X
X ::= Y | Z
Y ::= F Z
Z ::= ε

T



B ∩ T

Algorithm:
A ::= B C -> ${}_iA_k ::= {}_iB_j\ {}_jC_k$
for all FSA states i,j,k

${}_0F_1$ ::= setuid ${}_1W_1$	${}_0F_2$ ::= setuid ${}_1W_2$
${}_1W_1$ ::= ${}_1G_1\ {}_1X_1$	${}_1W_2$ ::= ${}_1G_1\ {}_1X_2$ | ${}_1G_2\ {}_2X_2$
${}_1X_1$ ::= ${}_1Y_1$ | ${}_1Z_1$	${}_1X_2$ ::= ${}_1Y_2$ | ${}_1Z_2$
	${}_2X_2$ ::= ${}_2Y_2$ | ${}_2Z_2$
${}_1Y_1$ ::= ${}_1F_1\ {}_1Z_1$	${}_1Y_2$ ::= ${}_1F_1\ {}_1Z_2$ | ${}_1F_2\ {}_2Z_2$
	${}_2Y_2$ ::= ${}_2F_2\ {}_2Z_2$
${}_1Z_1$ ::= ε	${}_2Z_2$ ::= ε

Testing a CFL for emptiness

Worklist algorithm:
1) Mark all terminals
2) For each rule $A ::= B\ C$,
   if B and C are marked, mark A
Continue until fixpoint reached

$_0F_1 ::= \text{setuid}\ {}_1W_1$

$_1W_1 ::= {}_1G_1\ {}_1X_1$

$_1X_1 ::= {}_1Y_1 \mid {}_1Z_1$

$_1Y_1 ::= {}_1F_1\ {}_1Z_1$

$_1Z_1 ::= \varepsilon$

$_0F_2 ::= \text{setuid}\ {}_1W_2$

$_1W_2 ::= {}_1G_1\ {}_1X_2 \mid {}_1G_2\ {}_2X_2$

$_1X_2 ::= {}_1Y_2 \mid {}_1Z_2$

$_2X_2 ::= {}_2Y_2 \mid {}_2Z_2$

$_1Y_2 ::= {}_1F_1\ {}_1Z_2 \mid {}_1F_2\ {}_2Z_2$

$_2Y_2 ::= {}_2F_2\ {}_2Z_2$

$_2Z_2 ::= \varepsilon$

Interpretation and intuition

- ${}_iF_j$ marked $\Leftrightarrow$ if we call f() with FSA starting in state i, f() might return with FSA in state j
- {(i,j) : ${}_iF_j$ marked} = transfer function for f()
- Standard interprocedural analyses algorithm, with function summaries $\Leftrightarrow$ a CFL emptiness test

Some ideas inspired by [Olender,Osterweil86]

A tangent: Generalized PDAs

- Normal PDA:
  Rule: α -> βγ Semantics: αδ…ω => βγδ…ω
- Generalized PDA:
  Rule: [$\Re$] α -> βγ
  Semantics: If αδ…ω matches regexp $\Re$, αδ…ω => βγδ…ω
- Fact: Generalized PDAs can only recognize CFLs.
  Every generalized PDA is equivalent to some normal PDA.
- Relevance: setjmp(), Java stack inspection

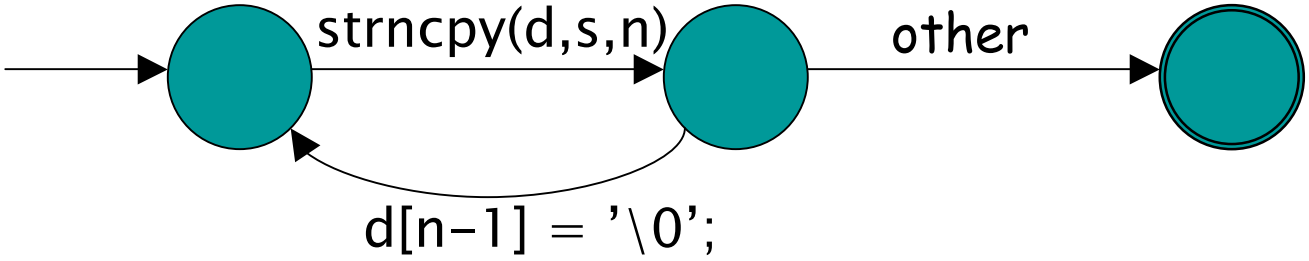
The MOPS implementation

- Fact: The set of reachable configurations of a PDA is a regular language [Büchi, Knuth]
- Let post*(c) = {c′ : c′ is reachable from c}
  - Fact: post*(c) is a regular language, and can be efficiently computed [FWW97, WB98]

- MOPS uses post*() to check whether there exists any reachable configuration where the FSA is in error state
  - We extend post*() to compute a "back-trace" for each reachable error configuration
  - The PDA model is compacted before model checking, for better performance
  - Pattern variables allow FSA to be more expressive

Misuse of strncpy()

- strncpy() does not null-terminate in boundary cases; programmer must force a null terminator explicitly
- Easy bug to miss, since it only triggers at boundary case



Example: A simple FSA to detect misuse of strncpy( ).
Error state indicates possible failure to null-terminate d.

(Real property is much more complex: many ways to terminate; pre-termination vs. post-termination; delayed termination.)

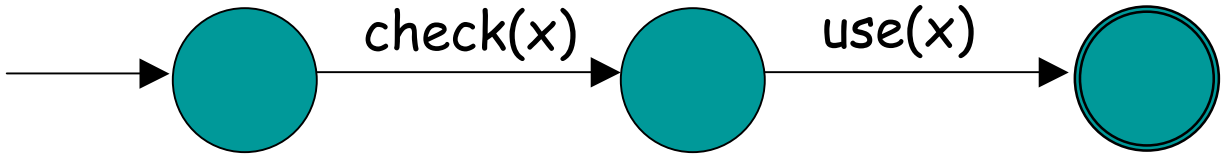
TOCTTOU (time-of-check to time-of-use)

- Canonical example of a TOCTTOU vulnerability:

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

- Notice: not an atomic operation!

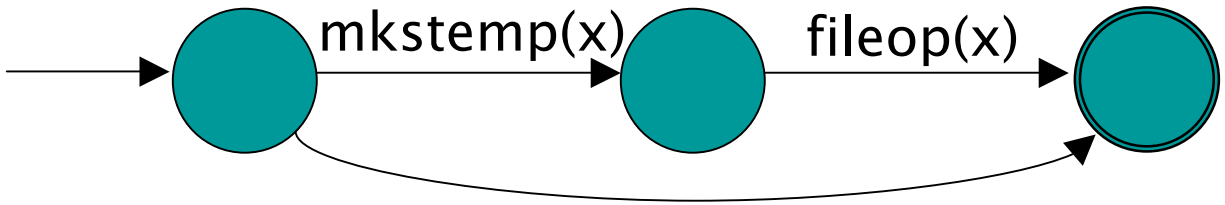
- Bug: Permissions may change between access() & open()
  - Attacker can arrange for this to happen in an attack



check = { access, lstat, stat, readlink, statfs }
use = { chmod, open, remove, unlink, mount, link, mkdir, rmdir … }

Insecure temporary file creation/use

- Temporary file creation requires special care:
  1) unguessable filename; 2) safe permissions;
  3) file ops should use fd, not filename (TOCTTOU)



{ tmpnam(), tempnam(), mktemp(), tmpfile() }

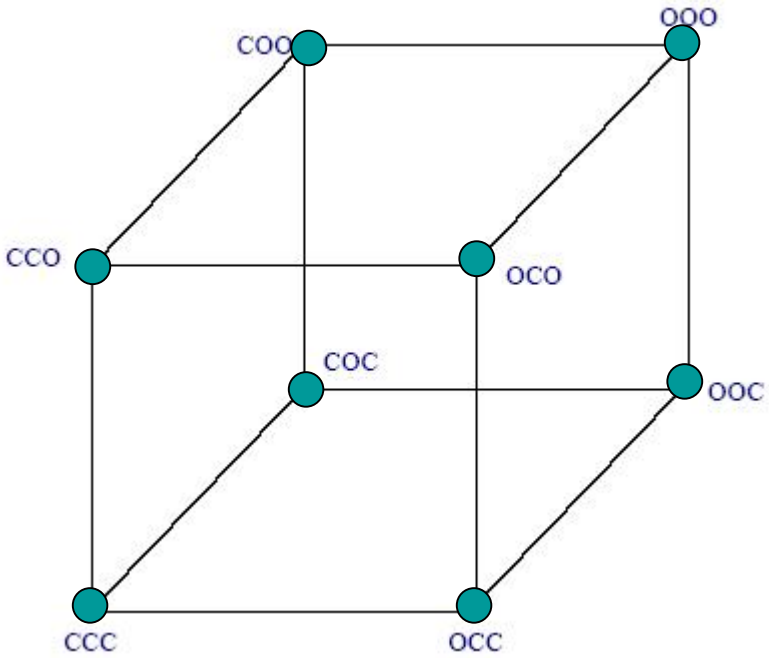
fileop(x) = { open(x), chmod(x), remove(x), unlink(x) … }

Stderr vulnerabilities

- Example of a setuid program with a stderr vulnerability:

```
fd = open("/etc/password", O_RDRW);
if (doit(argv[0], fd) < 0)
    perror(argv[0]);
```

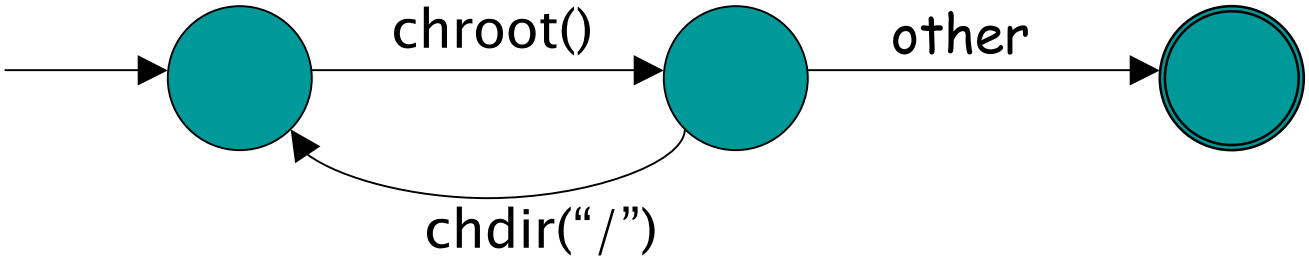
- Threat: Attacker might call us with fd 2 closed; then perror() will over-write password file



Transitions along edges of cube.
Program might start at any state.
Calling open() from any state except OOO is an error.

Proper setup of chroot jails

- chroot() creates a jail.
- Problem: jail breaks are possible, if cwd is outside jail.



Real FSA is much more complex. There are safe idioms (e.g., chdir(d); chroot(d);) not shown here.

MOPS in the large

- Experiment: Analyze an entire Linux distribution
  - Redhat 9, all C packages (732 pkgs, ~ 50 MLOC)
  - Security analysis at an unprecedented scale
- Team of 4 manually examined 900+ warnings
  - 1 grad student; 3 undergrads new to MOPS
  - Exhaustive analysis of TOCTTOU, tmpfile, others; statistical sampling of strncpy
  - Laborious: multiple person-months of effort
- Found **108** new security holes in Linux apps

Security Property	Warnings	Real bugs	Bug ratio
TOCTTOU	790	41	5%
temporary files	108	34	35%
strncpy	1378	(unknown)	~ 5-10%
Total	2333	108+	

Practical issues

Good error reporting makes a huge difference.

- Error causes: Find line of code that is the likely "cause".
- Error clustering: Group errors by "cause".
- Exhaustive error reporting: Find all errors.
  Show shortest/simplest ones first.
- Backtracking: Provide a stack dump. Also, build a
  trace that shows how this error point can be reached.
- UI: Can browse code annotated with FSA states.
  Let user quickly jump to location of any FSA transition.

Practical issues

Build integration: harder than it seems.

- Try #1: Edit makefiles by hand. This sucks!
- Try #2: Interpose with GCC_EXEC_PREFIX.
  Build .cfg instead of .o. Fails: autoconf, ...
- Try #3: Interpose. Build both .cfg and .o.
  Fails: They get out of sync. Build process renames .o's.
- Try #4: Build both, stuff .cfg into .o file with ELF tricks.
  Better. But reveals that gcc interposition is fragile.
- Try #5: Replace /usr/bin/{cc1,ld} with wrapper.
  Currently successful with > 98% of Debian packages.

Bug #1: "zip"

Pathname from cmd line

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

... eventually writes new zipfile to `d` ...

Bug #2: "ar"

```
exists = lstat (to, &s) == 0;
if (! exists ||
     (!S_ISLNK (s.st_mode) && s.st_nlink == 1)){
    ret = rename (from, to);
    if (ret == 0) {
      if (exists) {
        chmod (to, s.st_mode & 0777);
        if (chown (to, s.st_uid, s.st_gid) >= 0)
           chmod (to, s.st_mode & 07777);
      }
    }
}
```

Bug #3

```
static void open_files() {
  int fd;
  create_file_names();
  if (input_file == 0) {
    input_file = fopen(input_file_name, "r");
    if (input_file == 0)
       open_error(input_file_name);
    fd = mkstemp(action_file_name);
    if (fd < 0 || (action_file =
                   fdopen(fd, "w")) == NULL) {
      if (fd >= 0)
        close(fd);
      open_error(action_file_name);
    }
}
void open_error(char *f) {
  perror(f); unlink(action_file_name); exit(1);
}
```

Lessons & surprises from the MOPS effort

- Unexpectedly, most real bugs were local
- False alarm rate high. Doing better requires deeper modeling of OS/filesystem semantics.
  - Path sensitivity only good for ≤ 2x improvement
  - Many non-bugs were still very interesting (represented fragile assumptions about environment)
- Engineering for analysis at scale is highly non-trivial
  - Good UI, explanation of errors is critical
  - Build integration so important — and so hard — that we re-implemented it no less than five times
- But worth it: Large-scale experiments incredibly valuable
- Tech. transfer: techniques being adopted in commercial security code scanning tools

Concluding thoughts

- Software has bugs. Security bugs are particularly costly. Tools can help spot them & fix them before they're exploited.
- Pushdown model checking is simple—and it works.
  - MOPS's analysis core is crude by modern standards, but still pretty effective: > 100 bugs, in 50M LoC.
  - Error reporting, UI, build integration, FSA's potentially more important than analysis itself.
- Make it real. Experiment at scale. How many bugs can you find in a modern Linux distribution?

Questions?