Model-Carrying Code

Sekar, Venkatakrishnan, Basu, Bhatkar, DuVarney
Stony Brook University

presented by Michael Roitzsch

Problem

- people run software from untrusted sources
- all software runs with full user privileges

Solution Space

execution monitoring — MCC — static analysis

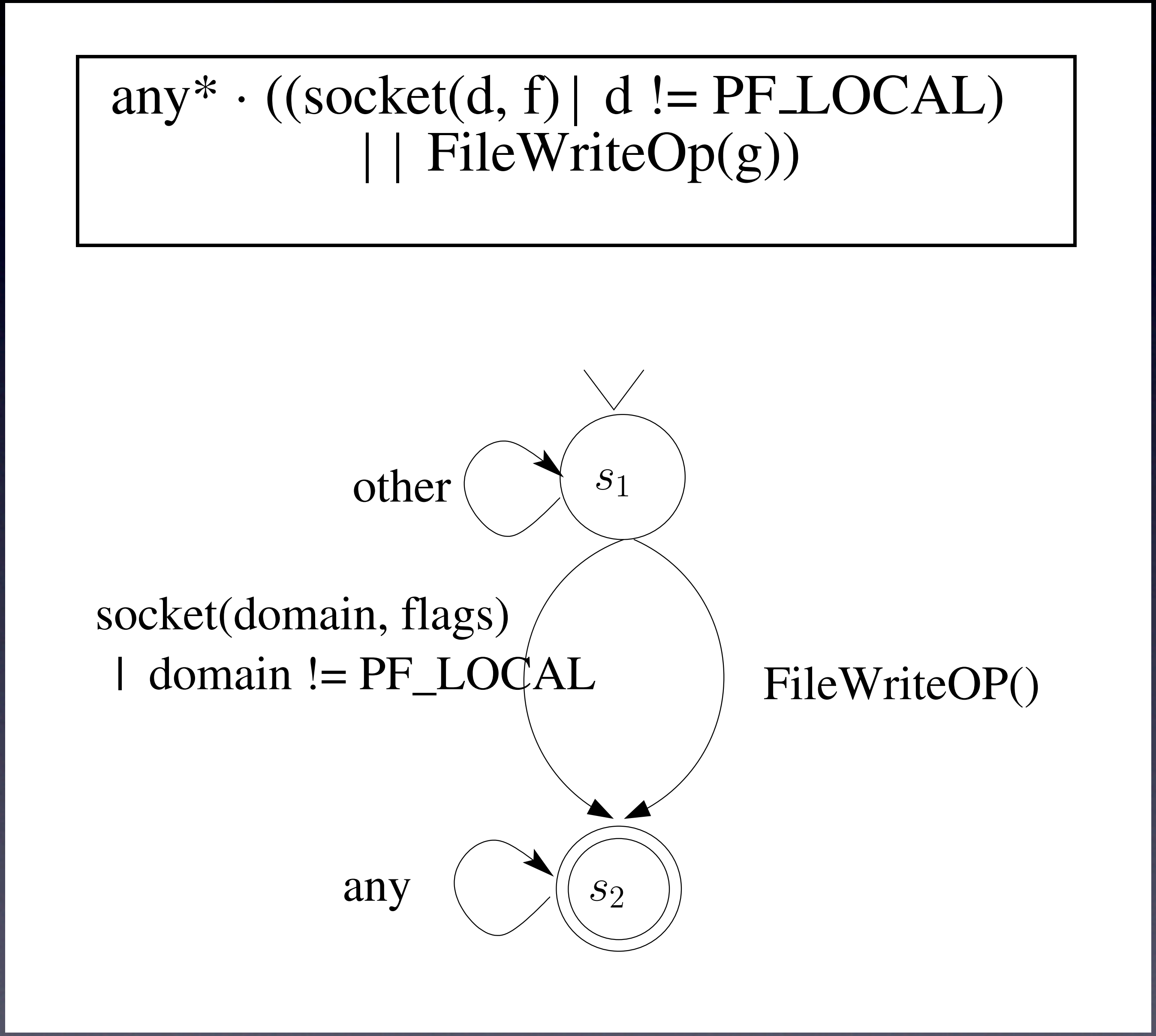
- violation detected at runtime
- consumer specifies policy
- practical implementations

- violation detected prior to running
- producer-generated proof limits policies
- practical difficulties

Policy

- behavior modelled by externally observable events (system calls)
- access-control and resource-usage policies
- describe bad sequences of events
- extended finite state automata (EFSA)
- policy-violating traces are accepted

Policy



any* · ((socket(d, f)| d != PF_LOCAL)
| | FileWriteOp(g))

Model

- single model must be usable for different policies
- model should closely capture syscall behavior
- EFSA to represent syscalls plus arguments

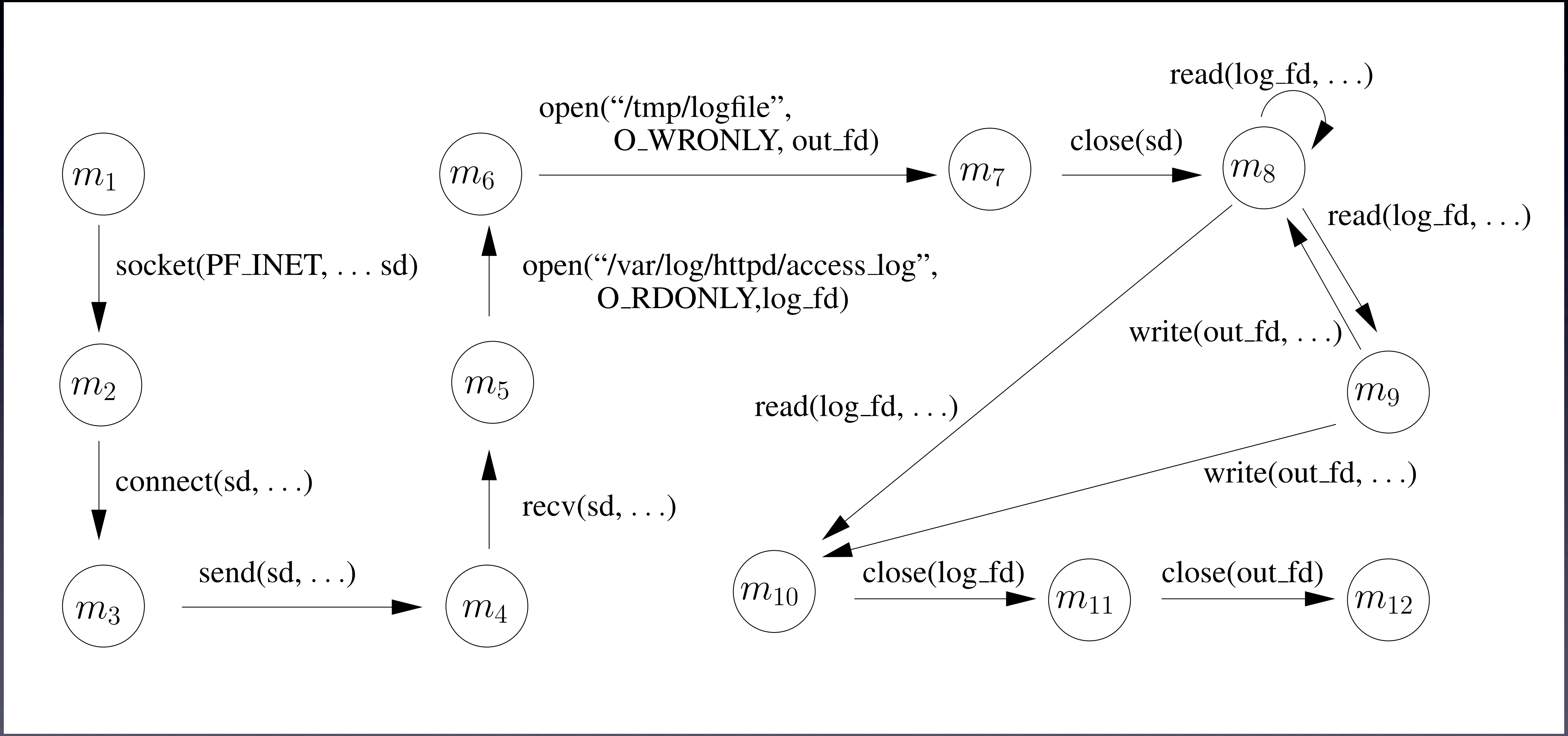
Model Generation

- based on tracing
- learning process should cover program behavior well
- fully automated
- log all system calls with arguments and preprocess

Model Generation

1. learn FSA states and transitions
2. learn argument values
3. learn argument relationships

Model



m_1
socket(PF_INET, . . . sd)
m_2
connect(sd, . . .)
m_3
send(sd, . . .)
m_4
recv(sd, . . .)
m_5
open("/var/log/httpd/access_log",
O_RDONLY,log_fd)
m_6
open("/tmp/logfile",
O_WRONLY, out_fd)
m_7
close(sd)
m_8
read(log_fd, . . .)
read(log_fd, . . .)
write(out_fd, . . .)
m_9
read(log_fd, . . .)
write(out_fd, . . .)
m_{10}
close(log_fd)
m_{11}
close(out_fd)
m_{12}

Verification

- build product automaton of model and policy
- check for satisfyability
- some conditions need to be evaluated optimistically
- present conflict summary to the user and allow policy adaption

Enforcement

- validate actual syscalls against the model at runtime
- on violation, program is malicious or model inaccurate
- abort application

Enforcement

Application	Overhead
xpdf	30%
gaim	21%
http-analyze	24%

Criticism

- model might be too loose due to optimistic aggregation – **false negatives**
- model might be too tight due to insufficient trace coverage – **false positives**
- termination especially on corner cases, where you want your app to exit gracefully
- Return error instead of termination?

Criticism

- Are the policies readable?
- they seem retrofitted
- Are they more suitable to blacklists?
- models do not compose easily, so no individual library models
- would have been cool for browser plugins

Criticism

- Multithreading anyone?
- I am not convinced that stateless filters would not solve the same problems much easier.
  - far less overhead
  - readable policies
  - already deployed

AppArmor

```
/usr/sbin/ntpd flags=(complain) {
   #include <abstractions/base>
   #include <abstractions/nameservice>
   #include <abstractions/xad>
   capability net_bind_service,
   capability setgid,
   capability setuid,
   capability sys_chroot,
   capability sys_time,
   network inet dgram,
   /etc/ntp.conf r,
   /etc/ntp/drift* rwl,
   /etc/ntp/keys r,
   /var/run/ntpd.pid w,
}
```

Seatbelt

```
(deny default)
(allow process-fork)
(allow process-exec (regex "^/usr/sbin/ntpd$"))
(allow sysctl-read)
(allow network*)
(allow file-read-data file-read-metadata
   (regex "^(/private)?/etc/ntp\\.(conf|keys)$"))
(allow file-read-data file-read-metadata file-write-data
   (regex "^(/private)?/var/db/ntp\\.drift(\\.TEMP)?$"))
(allow file-write* file-read-data file-read-metadata
   (regex "^(/private)?/var/run/ntpd\\.pid$"))
(allow time-set)
(import "bsd.sb")
```