Model-Carrying Code

A Practical Approach for Safe Execution of Untrusted Applications

R. Sekar, V.N. Venkatakrishnan et al.

Presentation by
Gregor König

17.06.2004

Motivation

- Safe execution of untrusted Code
- Unstrusted Code:
  - Document handlers and viewer
  - Instant Messaging
  - Filesharing
  - Mobile Code
    - Applets
    - ActiveX

→ Risk of faulty/malicious code is high

State of the Art

- Code Signing
  - Trusted producer – unsafe execution
- Content inspection
  - Program analysis
  - Difficult for binary code
- Behaviour confinement
  Execution Monitoring
  - Disallow operations
  - Runtime aborts

State of the Art

- Java Security
  - Fine grained
  - Security completely on the code-consumer side
- Proof-Carrying Code
  - The producer has to give prove that code is secure
  - Does not know about consumer security policies

State of the Art

- Problem:
  - Code producer knows nothing about consumers policies
  - Code consumer does not know access needs of the program
- Needed:
  - Cooperation of the two parties
  - Producer can express their security needs
  - Consumer can check if needs are consistent with his security policies

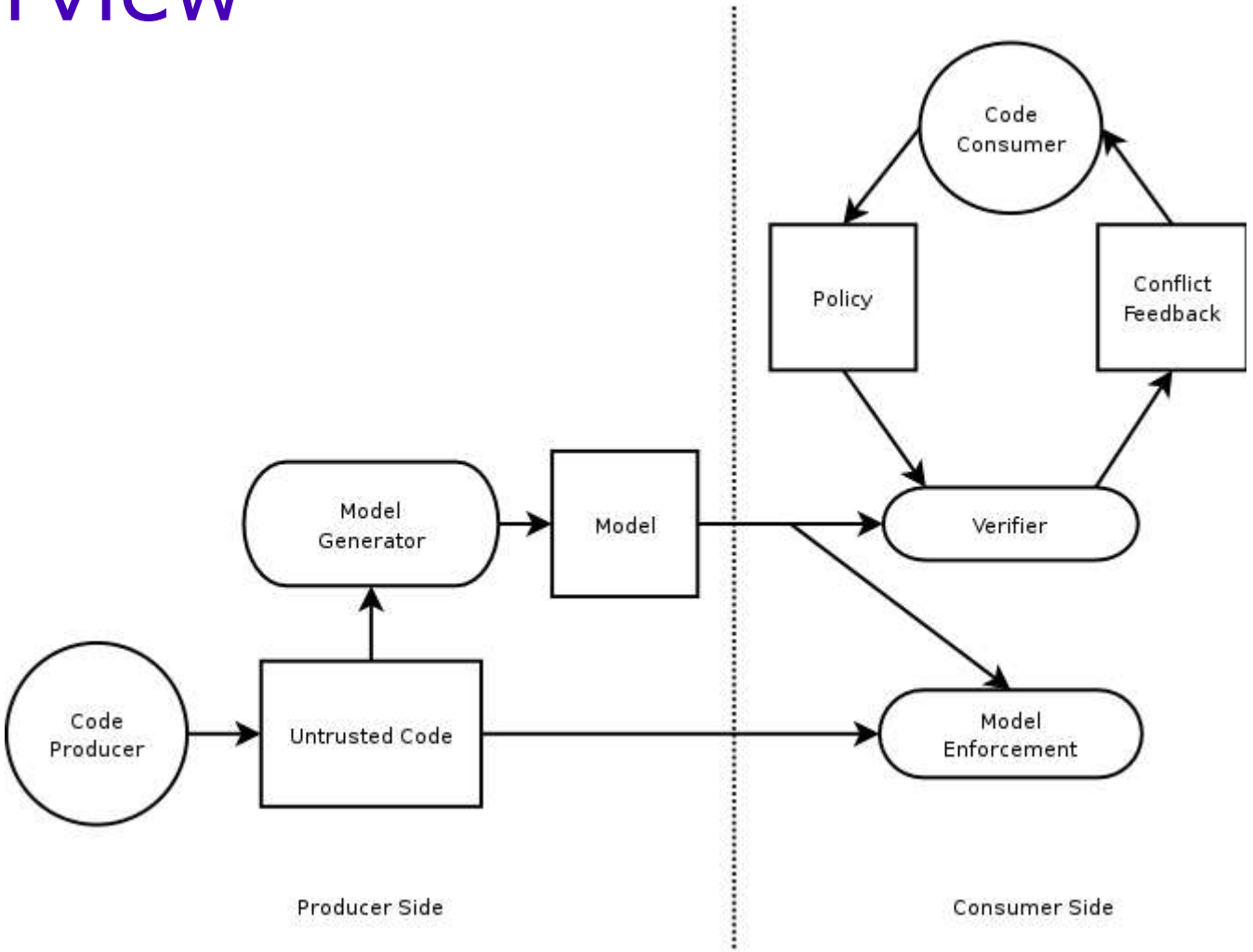
Overview

- Approach
  - Goals
  - Security Policies
  - Model Extraction
  - Verification
  - Enforcement
- Implementation
  - Performance
  - Integration into existing Systems
- Conclusion

Goals

- Benefit from untrusted code.
- Prevent malicious/faulty behaviour.
- Decouple code producer and consumer.
- Combine
  - convenience of static analysis,
  - Practicality of execution monitoring.

Overview



The Model-Carrying Code Framework
Figure inspired by [1]

Overview

- Producer Side
  - Generates Model M
  - Produces Application A
- Consumer Side
  - Policy Satisfaction
    $B[M] \subseteq B[P]$
  - Model Safety
    $B[A] \subseteq B[M]$

  $\rightarrow B[A] \subseteq B[P]$

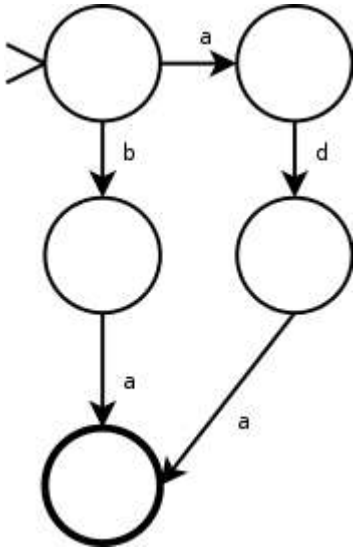
Security Policies

- Access control
- Resource usage policies
- History sensitive policies
  - e.g.: files created by program
- Language:
  - Extended finite state automata (EFSA)
    - FSA + ability to store values in a finite number of variables

Finite State Automaton

- 5-tupel A=(V,E,λ,I,T)

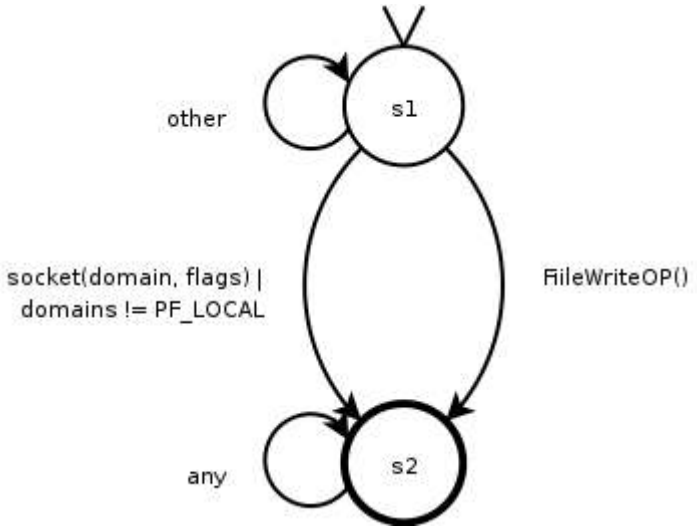
  V Vertices (states)
  E Edges
  λ function VxE → P(V)
  I I ⊂ V (initial states)
  T T ⊂ V (terminal states)



Security Policy

- BMSL Language
  - Behaviour Monitoring Specification Language

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



Model Extraction

- Static analysis
  - Program behaviour never violates the model.
  - Compile-time checking of values of variables is hard.
    → too conservative models

- Execution Monitoring
  - Monitoring program behaviour using test cases
    →non-conservative models

FSA Learning Algorithm

- Learning FSA from strings (execution traces) computationally hard problem
  - e.g.: abcda

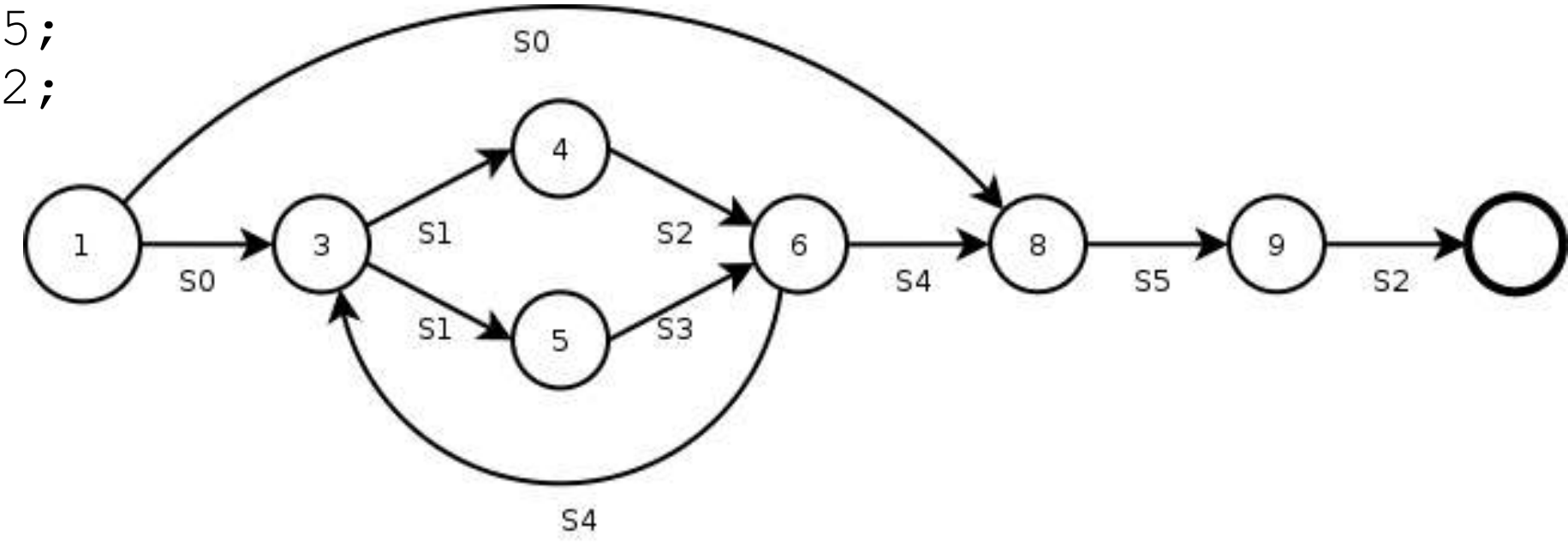
    → remember call location

    systemcall S at position PC:
    - new state PC
    - new transition S from previous PC' to PC

FSA Learning Algorithm

```
1   S0;
2   while(..){
3      S1;
4      if (..) S2;
5      else S3;
6      S4;
7   }
8   S5;
9   S2;
```

```
S0/1 S5/8 S2/9
S0/1 S1/3 S2/4 S4/6
S1/3 S3/5 S4/6 S5/8 S2/9
```



Learning Argument Values

- Model Extractor:
  - Online component
    - Intercepts system calls
    - Logs system calls
    - Logs useful system call arguments
  - Offline component
    - Extractor remembers system call arguments for each edge in FSA
    - Multiple arguments for one edge:
      - Threshold
      - Accumulate (e.g. /tmp/*)

Learning Argument Relationships

- Relationship between arguments:
  - open / write
  - accept / connect
- Which pairs?
  - Same argument type
  - Equality of argument

Web Log Analyzer

```
int main(int argc, char *argv[]) {
    int sd, rc, i, log_fd,out_fd,flag = 1;
    struct sockaddr_in remoteServAddr;
    char recvline[SIG_SIZE+1], sendline[SIG_SIZE+1];
    char buf[READ_SIZE];

    init_remote_server_addr(&remoteServAddr,...);
    init_sendmsg(sendline,...);
    sd = socket(PF_INET,SOCK_STREAM,0); ◀
    connect(sd, (struct sockaddr*)&remoteServAddr,sizeof(...)); ◀
    send(sd, sendline, strlen(sendline)+1,0); ◀
    recv(sd, recvline, SIG_SIZE,0); ◀
    recvline[SIG_SIZE] ='\0';
    log_fd = open("/var/log/httpd/access_log",O_RDONLY); ◀
    out_fd = open("/tmp/logfile",O_CREAT|O_WRONLY); ◀
    close(sd); ◀
    ...
}
```

Code example taken from [1]

EFSA Model

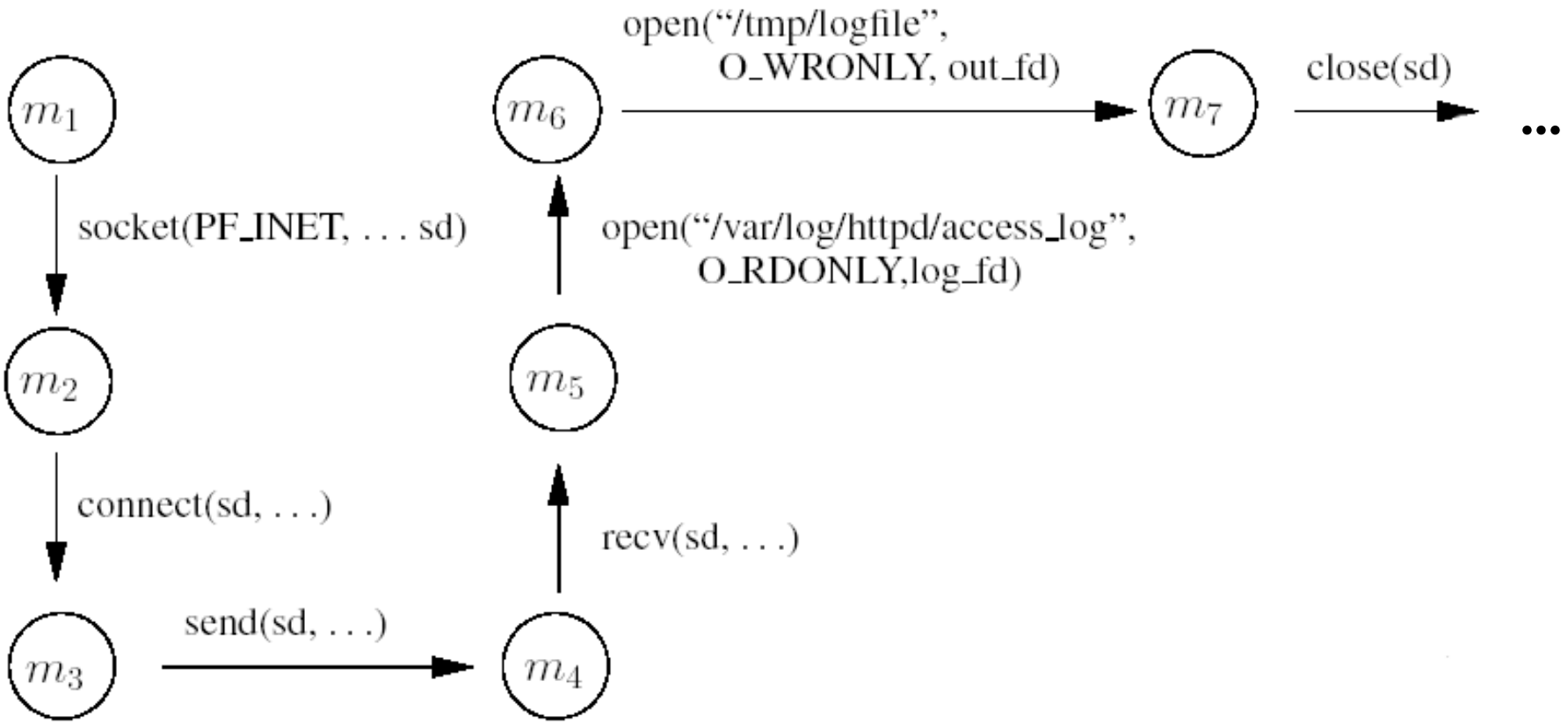


Figure taken from [1]

MMC Framework

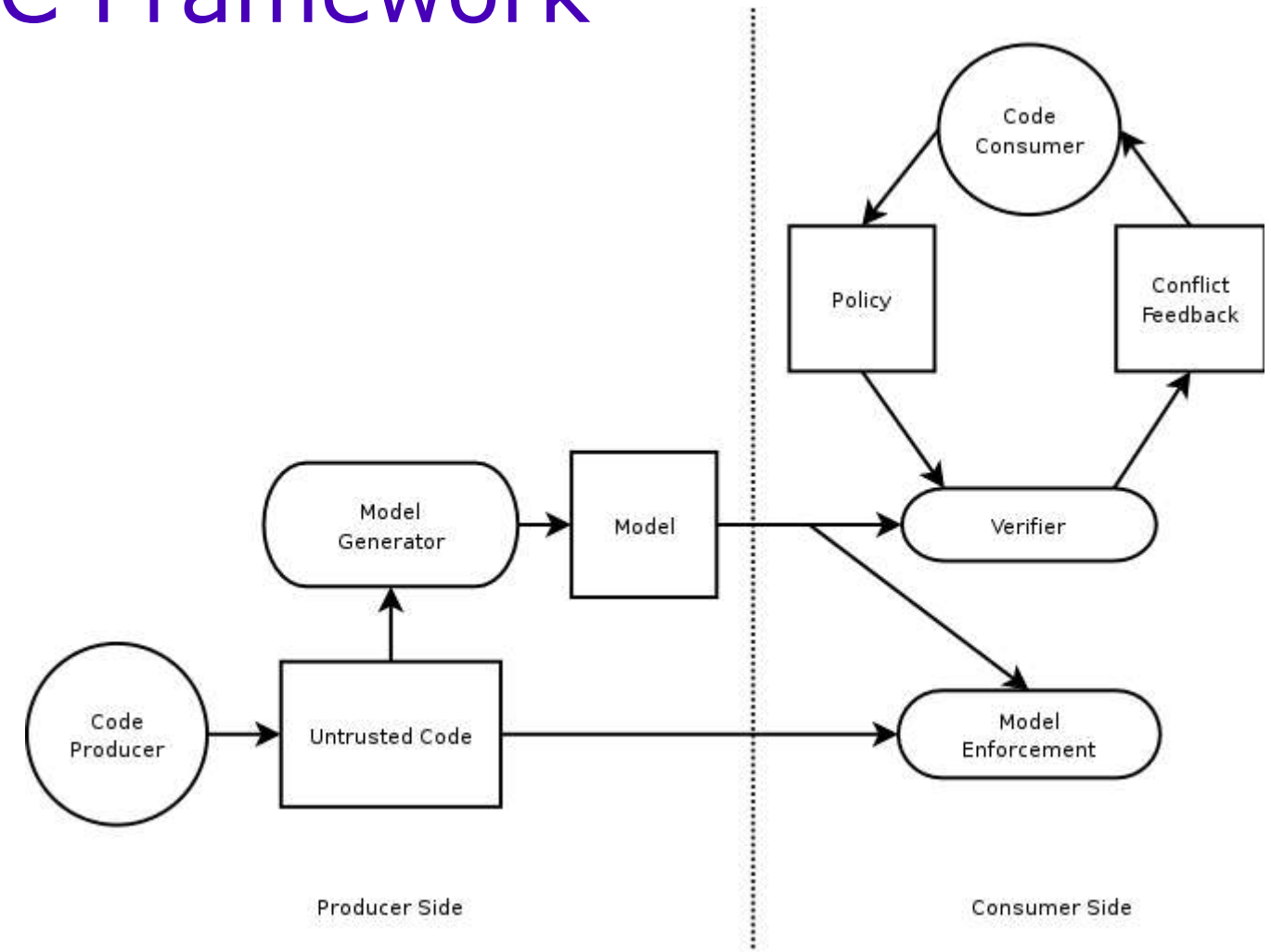


Figure inspired by [1]

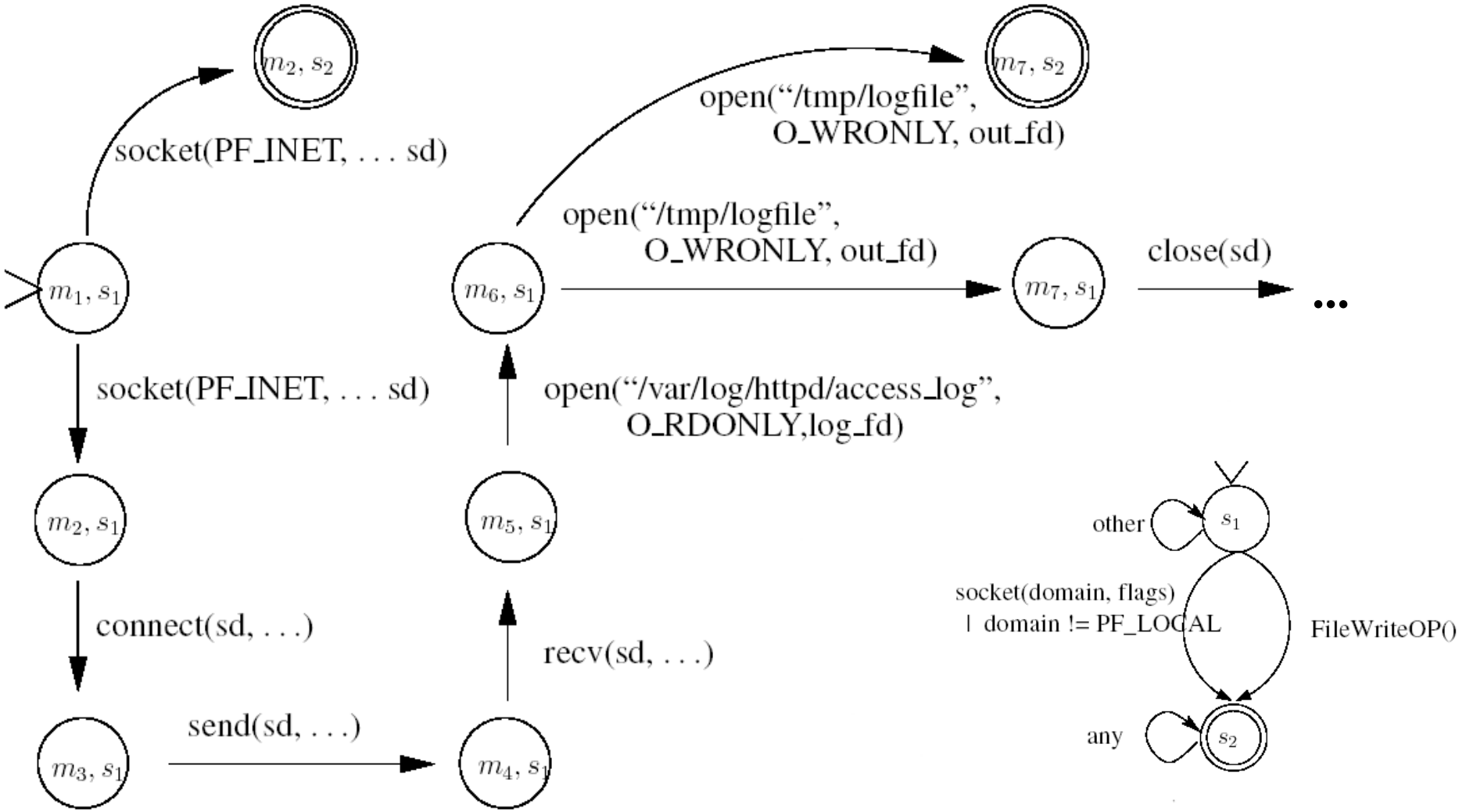
Verification

- Does the model M satisfy security policy P?
  $B[M] \subseteq B[P]$?

  Policy automaton represents negation $\overline{P}$ of P
  $B[M] \cap B[\overline{P}] = \varnothing$?

- M x $\overline{P}$ represents violations
  → all feasible paths lead to a violating state

Product Automaton



Figures taken from [1]

Conflict Representation

- Product is projected onto policy
- Common apsects of multiple vialoating paths are combined
  (→ approach similar to aggregation during model extraction)

  `open` operation on file `/tmp/logfile` in write mode
  `socket` operation involving the domain `PF_INET`

Policy Refinement

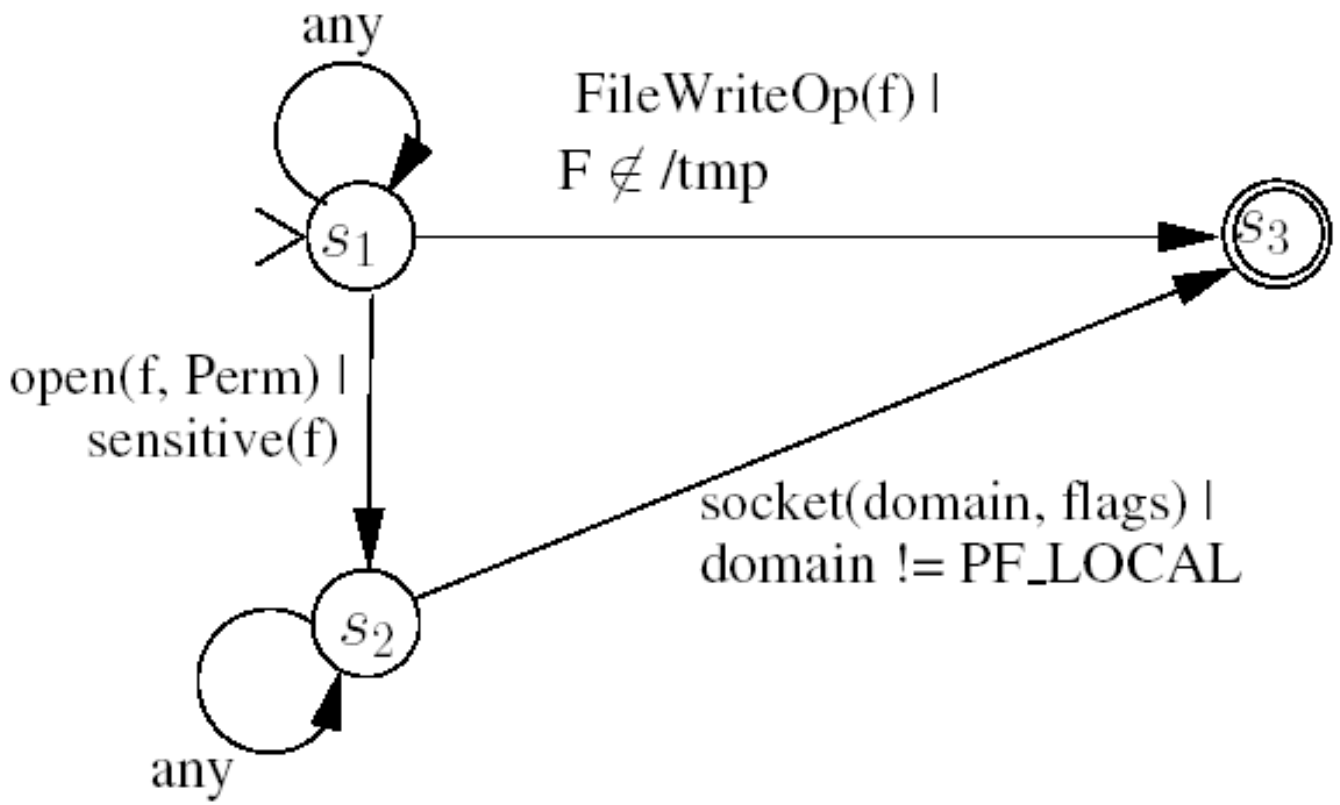


Figure taken from [1]

Model Enforcement

- Why not policy enforcement?
  - Possible in Framework
  - Allows operations identified as illegal by producer
- Model is enforced during execution
  - System call interception
  - Matching arguments against model

Model Enforcement

- Violating behaviour:
  - Runtime abort
    - Most code producers honest
  - Reasons:
    - Intentional misrepresentation
    - Unintentionally: Model constructed through runtime monitoring → test cases not sufficient

Implementation

- Model generation
  - System call interception: Linux ptrace
  - Context switches → 40% - 200% overhead
    - Order of few minutes
    - But offline activity
  - 9KLOC C++

Implementation

- Verifier
  - XSB logic programming system
  - 300 LOC Prolog
- Model Enforcement
  - In-kernel module to perform system call interception
  - 2-30% overhead without tuning attempts

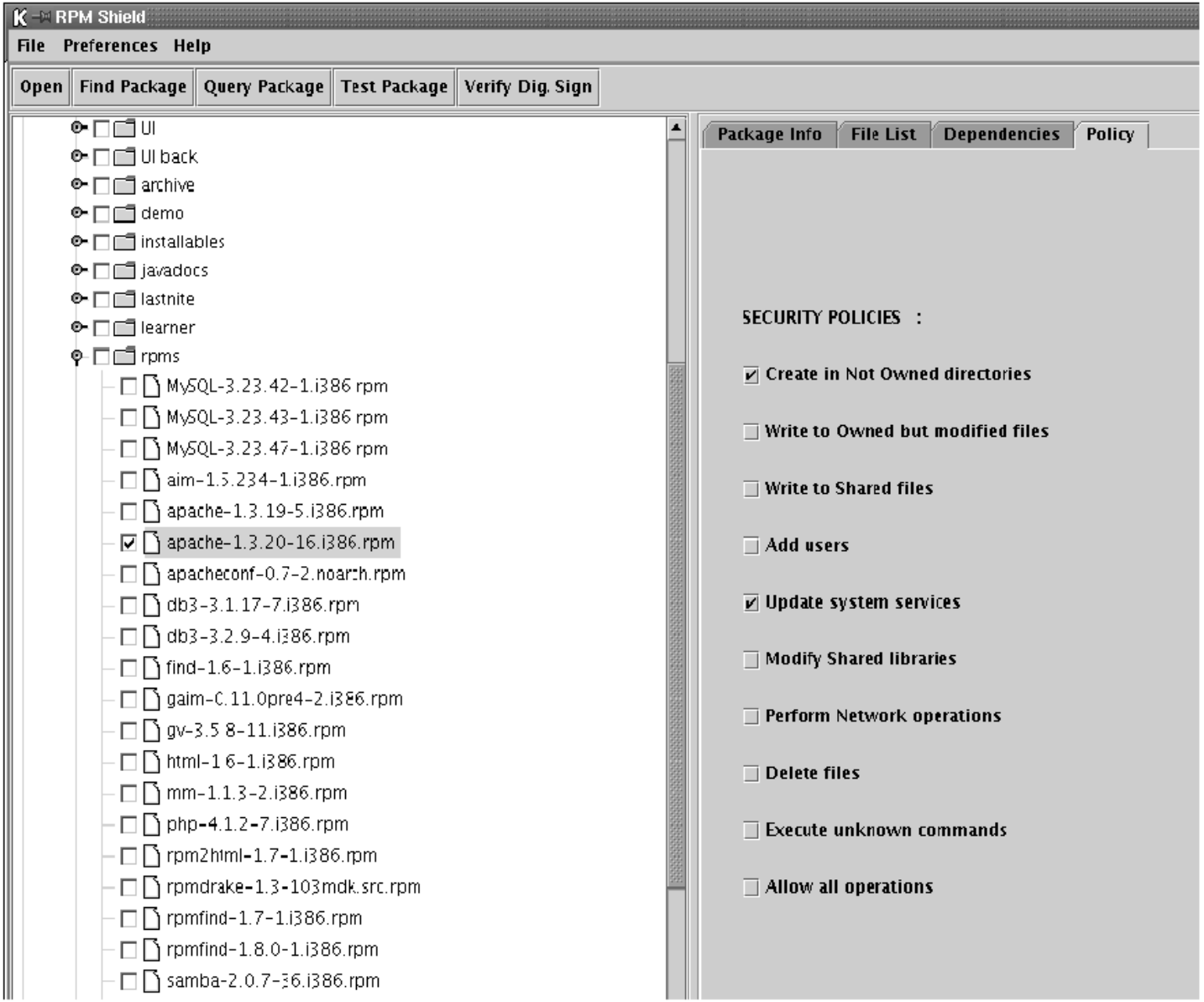
Performance

App	Xpdf 1.0	Gaim 0.53	http-analyze
Size KB	906	3173	333
States	125	283	158
Transitions	455	937	391
Enforcement Overhead	30%	21%	2.4%
Verification msec	1.0	1.8	0.7

Results taken from [1]

Integration

- Synergy with cryptographic signing
  - Signed models
- RMPShield
  - RedHat Package Manager
  - Assure installation of new package does not manipulate existing files (overwrite, ..)
  - Control over pre- and post installation scripts



Screenshot taken from [2]

Conclusion

- Practical and useable
- Acceptable Overhead
- ToDo:
  - Catalog of policies for applications
    Verifier chooses policy automatically
  - Improve understandability of nontrivial security violations

Thank you
for your Attention.

References

[1] R. Sekar, V.N. Vankatakrishnan; *Model-Carrying Code: A Practical Approach for Safe Execution of Untrusted Applications;* SOSP'03

[2] V.N. Vankatakrishnan, R. Sekar; *An Approach for Secure Software Installation;* RPMShield Documentation http://www.seclab.cs.sunysb.edu/rpmshield/docs/rpmshield.ps

[3] XSB - A Logic Programming and Deductive Database system for Unix and Windows; http://xsb.sf.net