Model-Checking In-lined Reference Monitors

Meera Sridhar and Kevin W. Hamlen
CS6301-002: Language-based Security
Publications


In-lined Reference Monitors (IRM)

[Schneider, TISSEC, ‘00]

- enforce safety policies by injecting security guards directly into untrusted binaries
- maintain *history* of security-relevant events
- Advantages:
  - deployment flexibility (OS/VM remains unmodified)
  - enforce richer policies, sequence-sensitive policies
  - code recipient can specify security policy
  - application-specific policies
In-lined Reference Monitors

```java
int count = 0;
...
if (count ≤ 10)
{
    email.Send;
    count ++;
}
else
halt;
...
```

**Policy:** ≤ 10 email sends

**Rewriter:** instruments the untrusted code with IRMs

**Reified security state variable:** keeps track of security state
Aspect-Oriented IRMs

Aspect-Oriented IRMs

Aspect-Oriented Programming [Kiczales et al, ECOOP, 1997] has become a standard approach for implementing IRMs

EXAMPLE:
Policy: at most 10 calls to Mail.mail(Mail.Send,...)

AspectJ implementation:

```java
aspect Monitor {
  private static int counter = 0;

  pointcut sendevent(x): call(Mail.mail(int,..)) &&
    if(thisJoinPoint.getArgs()[0]==x);

  before(): sendevent(Mail.Send) {
    if (counter >= 10)
      throw new Exception("security violation");
    ++counter;
  }
}
```

- **aspect** Monitor
- **private static int** counter = 0;
- **pointcut** sendevent(x): call(Mail.mail(int,..)) &&
  if(thisJoinPoint.getArgs()[0]==x);
- **before()**: sendevent(Mail.Send) {
  if (counter >= 10)
    throw new Exception("security violation");
  ++counter;
}
In-lined Reference Monitors

• Long history of IRM Implementations
  – SASI/PoET [Erlingsson & Schneider, NSPW 99]
  – MOBILE [Hamlen, Morrisett, & Schneider, PLAS 06]
  – Polymer [Ligatti, Bauer, & Walker, TISSEC 09]
  – Java-MOP [Chen & Roşu, TACAS 05]
  – ConSpec [Aktug & Naliuka, SCP 08]
  – FIRM [Li & Wang, ACSAC 10]
  – many others
IRM Example: Web Ad Security
[Louw, Ganesh, Venkatakrishnan, USENIX Security, 2010]

Third Party Ad content given full page access by default! – Confidentiality and Integrity issues
1. Banner ad
2. Skyscraper ad – needs to read page for *contextual targeting* – risk of exposing private content such as email ids
3. Inline text ad – *contextual targeting* – same risk
4. Floating ad – needs control of page real estate – may interfere with trusted components

Certifying In-lined Reference Monitors

1. rewriters contain disassemblers, binary analysis tools, compilers, optimizers, code-generators
2. rewriters may be outsourced to third parties with different security interests
3. policy specifications can change rapidly as new attacks appear and new vulnerabilities are discovered

Without certification, TCB large & complex!
Certifying In-lined Reference Monitors

- certifying IRMs easier than verifying safety of arbitrary code!
- lighter weight
  - SPIN vs. our early work
- different from Proof-Carrying Code (PCC)
  - PCC rewriters (certifying compilers) leverage source level info typically unavailable to binary rewriters

Related work:
- ConSpec (certification via contracts)
- MoBILe (certification via type-checking)
Bottom Line: Runtime monitoring is very powerful, but we want the high assurance of static analysis.

Solution: Static verification of IRMs yields best of both worlds! Combine the power & flexibility of runtime monitoring with strong formal guarantees of static analysis.
Certifying In-lined Reference Monitors

What do we want from the certifier?

• automatic, machine-certification of IRMs on-demand
• formal guarantees of
  ✓ soundness
  ✓ transparency (behavior-preservation)
• light-weight certifier (embedded systems)
Aspect-Oriented IRM In-lining and Certification

- Binary code
- Aspect weaver/in-liner
- Self-monitoring code
- TRUSTED
- Aspect
- Verifier
  - Execute
  - Reject
**SPoX Policy Example** [Hamlen, Jones, PLAS, 2008]

**Policy:** at most 10 calls to `Mail.mail(Mail.Send,...)`

**Security Automaton:**

- `¬sendevent`  
- `sendevent`  
- `s=0`  
- `s=1`  
- `...`  
- `sendevent`  
- `s=10`

**SPoX formalization:**

```plaintext
(state name="s")
(pointcut name="sendevent"
 (and (call Mail.mail) (argval 1 (inseq Mail.Send))))
(forall "i" from 0 to 9
 (edge name="increment" 
 (pc name="sendevent")
 (nodes "s" i, i+1)))
(edge name="violation" 
 (pc name="sendevent")
 (nodes "s" 10, #))
```

abstract security state  
pointcuts: automaton edge labels (events)  
edges: security state transitions
Aspect-Oriented IRM In-lining and Certification

- Binary code
- Aspect weaver/in-liner
- Self-monitoring code
- Verifier
- Execute
- Reject

TRUSTED
Approach: Model-checking

- policy model + new binary code are the two inputs to model-checker
- model-checking process
  - abstract-interpret new binary code
  - interpreter bi-simulates code and automaton
  - model-checker proves that there are no automaton-rejected states in any reachable flows
- **Main Challenge:** How to curb state-space explosion?

---


In-lining Example

**Policy:** at most 10 calls to Mail.mail(Mail.Send, ...)

```java
if (x == Mail.Send) {
    if (counter >= 0 && counter <= 9)
        temp_counter = counter + 1;
    else
        throw new Exception(“security violation”);
    counter = temp_counter;
}
Mail.mail(x, ...);
```
Abstract Interpretation Example

**Policy:** at most 10 calls to Mail.mail(Mail.Send,...)

```java
if (x == Mail.Send) {
    if (counter >= 0 && counter <= 9)
        temp_counter = counter + 1;
    else
        throw new Exception("security violation");
    counter = temp_counter;
}
Mail.mail(x,...);
```

**Legend:**
- \( s \) = abstract security state (from SPoX policy)
- \( c \) = counter (reified state)
- \( t \) = temp_counter (reified state)
**Policy:** at most 10 calls to Mail.mail(Mail.Send,...)

```java
if (x == Mail.Send) {
    if (counter >= 0 && counter <= 9)
        temp_counter = counter + 1;
    else
        throw new Exception("security violation");
    counter = temp_counter;
}
Mail.mail(x,...);
```

**Legend:**
- $s =$ abstract security state (from SPoX policy)
- $c =$ counter (reified state)
- $t =$ temp_counter (reified state)
Abstract Interpretation Example

**Policy:** at most 10 calls to Mail.mail(Mail.Send,...)

```java
if (x == Mail.Send) {
    if (counter >= 0 && counter <= 9)
        temp_counter = counter + 1;
    else
        throw new Exception("security violation");
    counter = temp_counter;
} Mail.mail(x,...); // s=s+1
```

**Legend:**
- s = abstract security state (from SPoX policy)
- c = counter (reified state)
- t = temp_counter (reified state)

**Diagram:**
- s≤10 \& s=c
- s≤10 \& s=c \& x=Mail.Send
- ... \& c≥0 \& c≤9
- ... \& t=c+1
- ... \& s=c_0 \& ... \& t=c_0+1 \& c=t
- s_0≤10 \& ... \& s=s_0+1
- s≤10 \& s=c

**SYNC!**
Synchronization States

• Definition
  – A state is synchronized when the abstract and reified security states “match”
  – different definition of “match” for each aspect implementation
  – each binary rewriter declares its definition of “match”
  – definition remains untrusted by verifier!

• Certification
  – verifies that initial symbolic state is synchronized
  – abstracts state to just “sync” whenever possible
  – uses “sync” as a loop invariant whenever possible
  – conservatively rejects if “sync” is insufficient to verify safety

• Controlling state-space explosion
  – vast majority of state-exploration reduces to linear-time sync-preservation checks
  – remaining exploration verifies that small blocks of in-lined code are sync-preserving, and that sync-preservation implies safety
  – “wrong” definition of sync just causes conservative rejection or slow convergence

Model-checking Certifier Implementation for SPoX IRM System

- IRM system for Java bytecode
- Prolog (about 5200 lines)
  - implements abstract interpreter
  - implements model-checker
    - decides boolean sentences over symbolic states
    - implemented with Constraint Logic Programming (CLP)
- Java code (about 9100 lines)
  - parses Java bytecode binaries using BCEL
  - outputs Prolog structures for certification
  - answers Prolog’s questions(e.g., class inheritance)
- Capabilities and limitations
  - certifier fully inter-procedural and inter-modular
  - almost all loops verify easily using sync as loop invariant
    - monitor-introduced loops in non-sync regions (rare) are the only hard ones
  - supports most forms of reflection
    - certifier just verifies adequacy of guards of reflective operations
  - synchronization invariant must be expressible as linear constraints
  - multithreading not supported
# Model-checking Certifier Implementation for SPoX IRM System

<table>
<thead>
<tr>
<th>Program</th>
<th>Policy</th>
<th>File Sizes (KB)</th>
<th># Classes</th>
<th>Rewrite Time (s)</th>
<th># Events</th>
<th>Total Verif. Time (s)</th>
<th>Model-check Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>old</td>
<td>new</td>
<td>libs</td>
<td>old</td>
<td>libs</td>
<td></td>
</tr>
<tr>
<td>EJE</td>
<td>NoExecSaves</td>
<td>439</td>
<td>439</td>
<td>0</td>
<td>147</td>
<td>0</td>
<td>6.1</td>
</tr>
<tr>
<td>RText</td>
<td></td>
<td>1264</td>
<td>1266</td>
<td>835</td>
<td>448</td>
<td>680</td>
<td>52.1</td>
</tr>
<tr>
<td>JSesh</td>
<td></td>
<td>1923</td>
<td>1924</td>
<td>20878</td>
<td>863</td>
<td>1849</td>
<td>57.8</td>
</tr>
<tr>
<td>vrenamer</td>
<td>NoExecRename</td>
<td>924</td>
<td>927</td>
<td>835</td>
<td>583</td>
<td>0</td>
<td>50.1</td>
</tr>
<tr>
<td>jconsole</td>
<td>NoUnsafeDel</td>
<td>35</td>
<td>36</td>
<td>0</td>
<td>33</td>
<td>0</td>
<td>0.6</td>
</tr>
<tr>
<td>jWeather</td>
<td>NoSendsAfterReads</td>
<td>288</td>
<td>294</td>
<td>0</td>
<td>186</td>
<td>0</td>
<td>12.3</td>
</tr>
<tr>
<td>YTDownload</td>
<td></td>
<td>279</td>
<td>281</td>
<td>0</td>
<td>148</td>
<td>0</td>
<td>17.8</td>
</tr>
<tr>
<td>jfilecrypt</td>
<td>NoGui</td>
<td>303</td>
<td>303</td>
<td>0</td>
<td>164</td>
<td>0</td>
<td>9.7</td>
</tr>
<tr>
<td>jknight</td>
<td>OnlySSH</td>
<td>166</td>
<td>166</td>
<td>4753</td>
<td>146</td>
<td>2675</td>
<td>4.5</td>
</tr>
<tr>
<td>Multivalent</td>
<td>EncryptPDF</td>
<td>1115</td>
<td>1116</td>
<td>0</td>
<td>559</td>
<td>0</td>
<td>129.9</td>
</tr>
<tr>
<td>tn5250j</td>
<td>PortRestrict</td>
<td>646</td>
<td>646</td>
<td>0</td>
<td>416</td>
<td>0</td>
<td>85.4</td>
</tr>
<tr>
<td>jrdesktop</td>
<td>SafePort</td>
<td>343</td>
<td>343</td>
<td>0</td>
<td>163</td>
<td>0</td>
<td>8.3</td>
</tr>
<tr>
<td>JVMail</td>
<td>TenMails</td>
<td>24</td>
<td>25</td>
<td>0</td>
<td>21</td>
<td>0</td>
<td>1.6</td>
</tr>
<tr>
<td>JackMail</td>
<td></td>
<td>165</td>
<td>166</td>
<td>369</td>
<td>30</td>
<td>269</td>
<td>2.5</td>
</tr>
<tr>
<td>Jeti</td>
<td>CapLoginAttmpts</td>
<td>484</td>
<td>484</td>
<td>0</td>
<td>422</td>
<td>0</td>
<td>15.3</td>
</tr>
<tr>
<td>ChangeDB</td>
<td>CapMembers</td>
<td>82</td>
<td>83</td>
<td>404</td>
<td>63</td>
<td>286</td>
<td>4.3</td>
</tr>
<tr>
<td>projtimer</td>
<td>CapFileCreates</td>
<td>34</td>
<td>34</td>
<td>0</td>
<td>25</td>
<td>0</td>
<td>15.3</td>
</tr>
<tr>
<td>xnap</td>
<td>NoFreeRiding</td>
<td>1250</td>
<td>1251</td>
<td>0</td>
<td>878</td>
<td>0</td>
<td>24.8</td>
</tr>
<tr>
<td>Phex</td>
<td></td>
<td>4586</td>
<td>4586</td>
<td>3799</td>
<td>1353</td>
<td>830</td>
<td>69.4</td>
</tr>
<tr>
<td>Webgoat</td>
<td>NoSqlXss</td>
<td>429</td>
<td>431</td>
<td>6338</td>
<td>159</td>
<td>3579</td>
<td>16.7</td>
</tr>
<tr>
<td>OpenMRS</td>
<td>NoSqlInject</td>
<td>1781</td>
<td>1783</td>
<td>24279</td>
<td>932</td>
<td>17185</td>
<td>78.7</td>
</tr>
<tr>
<td><strong>Averages</strong></td>
<td>****</td>
<td><strong>747</strong></td>
<td><strong>748</strong></td>
<td><strong>2522</strong></td>
<td><strong>369</strong></td>
<td><strong>1120</strong></td>
<td><strong>32.4</strong></td>
</tr>
</tbody>
</table>
IRMs must be fairly light-weight because they run on the code-consumer side

2. binary code parsing, code generation: tedious and error-prone
   – DCG’s facilitate binary parser implementation
   – Reversible predicates combine parser and code-generator into one piece of code!

3. IRM must elegantly implement many AST analyses and optimizations during rewriting
   – needed to preserve policy-compliant programs, generate efficient code
   – ASTs very elegantly represented and manipulated as Prolog structures

4. Instrumented code should be amenable to formal verification
   – Prolog implementation of binary rewriting isomorphic to a search for a correctness proof
   – excellent for integration with a certifying IRM system or a PCC system


A Simple LTL Model Checker written in Prolog for ActionScript Bytecode

```
1 % verify/2 takes a state and an existentially quantified LTL formula and checks
2 % whether the formula holds for that state.
3 %
4 % Atomic Propositions are labeled by 'ap'.
5 %
6 % holds/2 is true when the atomic proposition holds
7 % in the current state
8 %
9 % ftype/2 is a mapping from top-level temporal operators to their interpretation semantics
10 %
11 % The clause for 'a and b' should ensure that 'a' and 'b' hold on the same execution path. For simplicity
12 % of presentation, we omit this check here.
13 %
14 verify(State, F) :- ftype(F, inductive),
15 verify_inductive(State, F).
16 verify(State, F) :- ftype(F, coinductive),
17 verify_coinductive(State, F).
18 :- tabled verify_inductive/2.
19 verify_inductive(S, ap(AP)) :- holds(S, AP). % p
20 % Logical operators
21 verify_inductive(S, not(ap(AP))) :- % not(p)
22 \+ holds(S, AP).
23 verify_inductive(S, or(A,B)) :- % a or b
24 verify(S, A); verify(S, B).
25 verify_inductive(S, and(A,B)) :- % a and b
26 verify(S, A); verify(S, B).
27 % Inductive temporal operators
28 verify_inductive(S, x(A)) :- % X(a)
29 trans(S, S1), verify(S1, A).
30 verify_inductive(S, f(A)) :- % F(a)
31 verify(S, A); verify(S, x(f(A))).
32 verify_inductive(S, u(A,B)) :- % a U b
33 verify(S, B);
34 verify(S, and(A, x(u(A,B)))).
35 :- coinductive verify_coinductive/2.
36 % Coinductive temporal operators
37 verify_coinductive(S, g(A)) :- % G(a)
38 verify(S, and(A, x(g(A)))).
39 verify_coinductive(S, r(A,B)) :- % a R b
40 verify(S, and(A,B)).
41 % {a and b both occur, releasing b}
42 verify_coinductive(S, r(A,B)) :-
43 verify(S, and(B, x(r(A,B)))).
44 % {a does not hold, so b is not released)
```
FlashJaX: IRM technology for Web Ads
Proof of Certifier Correctness

- bismulation of concrete and abstract machines
  - concrete operational semantics of Java bytecode based on ClassicJava [Flatt, Krishnamurthi, & Felleisen, POPL 98]
  - abstract operational semantics of our interpreter
  - soundness relation between abstract and concrete states
- denotational semantics of SPoX [Hamlen & Jones, PLAS 08]
- preservation: The abstract machine soundly abstracts the concrete machine step-wise (uses soundness relation).
- progress: If the abstract machine doesn’t reject, the concrete machine doesn’t violate the policy. Abstract machine covers all real executions.


Concrete Machine

**Language Syntax (Simplified ActionScript)**

\[
i ::= \text{ifle } L \mid \text{getlocal } n \mid \text{setlocal } n \mid \text{jmp } L \mid \\
\quad \text{event } e \mid \text{setstate } n \mid \text{ifstate } n \ L
\]

**Programs and Labels**

\[
P ::= (L, p, s) \quad \text{(programs)}
\]

\[
p : L \to i \quad \text{(instruction labels)}
\]

\[
s : L \to L \quad \text{(label successors)}
\]

**Concrete States**

\[
\kappa ::= \langle L : i, \sigma, v, m, \tau \rangle \quad \text{(configurations)}
\]

\[
\sigma ::= \cdot \mid v :: \sigma \quad \text{(concrete stacks)}
\]

\[
v \in \mathbb{Z} \quad \text{(concrete values)}
\]

\[
v : \mathbb{Z} \to v \quad \text{(concrete stores)}
\]

\[
m \in \mathbb{Z} \quad \text{(concrete reified state)}
\]

\[
e \in \Sigma \quad \text{(events)}
\]

\[
\tau \in \Sigma^* \quad \text{(concrete traces)}
\]

\[
\chi_0 = \langle L_0 : p(L_0), \cdot, v_0, 0, \epsilon \rangle \quad \text{(initial configurations)}
\]

\[
v_0 = \mathbb{Z} \times \{0\} \quad \text{(initial stores)}
\]

Concrete Small-step Operational Semantics

\[ \frac{n_1 \leq n_2}{\langle L_1 : \text{ifle} \ L_2, n_1::n_2::\sigma, \nu, m, \tau \rangle \rightarrow \langle L_2 : p(L_2), \sigma, \nu, m, \tau \rangle} \quad (\text{CIFLEPos}) \]

\[ \frac{n_1 > n_2}{\langle L_1 : \text{ifle} \ L_2, n_1::n_2::\sigma, \nu, m, \tau \rangle \rightarrow \langle s(L_1) : p(s(L_1)), \sigma, \nu, m, \tau \rangle} \quad (\text{CIFLENeg}) \]

\[ \langle L : \text{getlocal} n, \sigma, \nu, m, \tau \rangle \rightarrow \langle s(L) : p(s(L)), \nu(n)::\sigma, \nu, m, \tau \rangle \quad (\text{CGETLOCAL}) \]

\[ \langle L : \text{setlocal} n, n_1::\sigma, \nu, m, \tau \rangle \rightarrow \langle s(L) : p(s(L)), \sigma, \nu[n := n_1], m, \tau \rangle \quad (\text{CSETLOCAL}) \]

\[ \langle L_1 : \text{jmp} L_2, \sigma, \nu, m, \tau \rangle \rightarrow \langle L_2 : p(L_2), \sigma, \nu, m, \tau \rangle \quad (\text{CJMP}) \]

\[ \tau e \in \mathcal{P} \]

\[ \langle L : \text{event} e, \sigma, \nu, m, \tau \rangle \rightarrow \langle s(L) : p(s(L)), \sigma, \nu, m, \tau e \rangle \quad (\text{CEVENT}) \]

\[ \langle L : \text{setstate} n, \sigma, \nu, m, \tau \rangle \rightarrow \langle s(L) : p(s(L)), \sigma, \nu, n, \tau \rangle \quad (\text{CSETSTATE}) \]

\[ \frac{m \neq n}{\langle L_1 : \text{ifstate} n \ L_2, \sigma, \nu, m, \tau \rangle \rightarrow \langle s(L_1) : p(s(L_1)), \sigma, \nu, m, \tau \rangle} \quad (\text{CIFSTATENeg}) \]

\[ \langle L_1 : \text{ifstate} n \ L_2, \sigma, \nu, m, \tau \rangle \rightarrow \langle L_2 : p(L_2), \sigma, \nu, n, \tau \rangle \quad (\text{CIFSTATEPos}) \]
Abstract States

\[ \hat{x} ::= \bot \mid \langle L : i, \hat{\sigma}, \hat{\nu}, m, (Res(q_m), \hat{\tau}) \rangle \mid \langle L : i, \hat{\sigma}, \hat{\nu}, \top_{VS}, \hat{\tau} \rangle \]

(abstract configs)

(abstract values)

(evaluation stacks)

(abstract stores)

(abstract reified state)

(bounded traces)

(abstract traces)

\[ \hat{\sigma} ::= \cdot \mid \hat{\nu} :: \hat{\sigma} \]

\[ \hat{\nu} \in VS \]

\[ \hat{\nu} : \mathbb{Z} \rightarrow \hat{\nu} \]

\[ \hat{m} \in \mathbb{Z} \cup \top_{VS} \]

\[ \hat{\tau} \in \bigcup_{n \leq k} : \Sigma^n \]

\[ \hat{\tau} \in SS \]
Abstract Small-step Operational Semantics

\[
\begin{align*}
  n_1 & \leq n_2 \\
  \langle L_1 : {\text{ifle}} L_2, n_1::n_2::\sigma, \nu, \hat{m}, \hat{\tau} \rangle & \rightsquigarrow \langle L_2 : p(L_2), \sigma, \nu, \hat{m}, \hat{\tau} \rangle & \text{(AIFLEPOS)} \\
  n_1 & > n_2 \\
  \langle L_1 : {\text{ifle}} L_2, n_1::n_2::\sigma, \nu, \hat{m}, \hat{\tau} \rangle & \rightsquigarrow \langle s(L_1) : p(s(L_1)), \sigma, \nu, \hat{m}, \hat{\tau} \rangle & \text{(AIFLENEG)} \\
  \top_{VS} & \in \{va_1, va_2\} \\
  L' & \in \{L_2, s(L_1)\} \\
  \langle L_1 : {\text{ifle}} L_2, va_1::va_2::\sigma, \nu, \hat{m}, \hat{\tau} \rangle & \rightsquigarrow \langle L' : p(L'), \sigma, \nu, \hat{m}, \hat{\tau} \rangle & \text{(AIFLETOP)} \\
  \langle L : {\text{getlocal}} n, \sigma, \nu, \hat{m}, \hat{\tau} \rangle & \rightsquigarrow \langle s(L) : p(s(L)), \nu(n)::\sigma, \nu, \hat{m}, \hat{\tau} \rangle & \text{(AGETLOCAL)} \\
  \langle L : {\text{setlocal}} n, va_1::\sigma, \nu, \hat{m}, \hat{\tau} \rangle & \rightsquigarrow \langle s(L) : p(s(L)), \sigma, \nu[n := va_1], \hat{m}, \hat{\tau} \rangle & \text{(ASETLOCAL)} \\
  \langle L_1 : {\text{jmp}} L_2, \sigma, \nu, \hat{m}, \hat{\tau} \rangle & \rightsquigarrow \langle L_2 : p(L_2), \sigma, \nu, \hat{m}, \hat{\tau} \rangle & \text{(AJMP)} \\
  \hat{\tau} & \subseteq \hat{\tau}' \subseteq \mathcal{P} \\
  \langle L : {\text{event}} e, \sigma, \nu, \hat{m}, \hat{\tau} \rangle & \rightsquigarrow \langle s(L) : p(s(L)), \sigma, \nu, \hat{m}, \hat{\tau}' \rangle & \text{(AEVENT)} \\
  \hat{\tau} & \subseteq \text{Res(qn)} \\
  \langle L : {\text{setstate}} n, \sigma, \nu, \hat{m}, \hat{\tau} \rangle & \rightsquigarrow \langle s(L) : p(s(L)), \sigma, \nu, n, (\text{Res(qn)}, e) \rangle & \text{(ASETSTATE)} \\
  \hat{m} & \in \{n, \top\} \\
  \langle L_1 : {\text{ifstate}} n L_2, \sigma, \nu, \hat{m}, (S, \tau) \rangle & \rightsquigarrow \langle L_2 : p(L_2), \sigma, \nu, n, (\text{Res(qn)}, \tau) \rangle & \text{(AIFSTATEPOS)} \\
  \hat{m} & \neq n \\
  (S - \text{Res(qn)}) & \tau \subseteq \hat{\tau} \\
  \langle L_1 : {\text{ifstate}} n L_2, \sigma, \nu, \hat{m}, (S, \tau) \rangle & \rightsquigarrow \langle s(L_1) : p(s(L_1)), \sigma, \nu, \hat{m}, \hat{\tau} \rangle & \text{(AIFSTATENEG)}
\end{align*}
\]
Other Proofs of Correctness

• Proof of Convergence
  – proof bounds height of abstraction lattice
  – abstract machine reaches fixed point in $O(n^2)$, $n =$ security automaton size


• Proof of Correctness of IRM *Transparency* Certifier
  – My SCP paper presents the first automated transparency-verifier for IRMs
  – untrusted, external invariant-generator
    • safely leverages rewriter-specific instrumentation information during verification
  – correctness of IRM transparency certifier extends previous proof with trace equivalence

References


References


Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. **Certified In-lined Reference Monitoring on .NET.** In *Proc. of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS): 7-16, 2006.*


References


