Specification-Based Monitoring

Jonathon Giffin, Somesh Jha, Barton Miller
University of Wisconsin
{giffin,jha,bart}@cs.wisc.edu

WiSA – Wisconsin Safety Analyzer
http://www.cs.wisc.edu/wisa
Problem

• Intrusion detection: How do you know when a process has been subverted?
  - Host-based intrusion detection
  - Remote intrusion detection

• How do malicious users subvert processes?
  - Host-based
  - Via remote execution
Our Solution

Specification-based monitoring

• Specify constraints upon program behavior
  - Static analysis of binary code

• At run-time, ensure execution does not violate specification
  - Limits execution to correct process behavior
Milestones

• **Dyck model**
  - Efficient & accurate program specification
  - Unites previous work in null calls with specification structure
  - Strong theoretical foundation

• **Data-flow-sensitive analysis**
  - Branch predicates based upon system call return values

• **Second conference paper (submitted)**
Overview

Static binary analysis for intrusion detection
- Shortcomings with prior program models
- Dyck model addresses shortcomings
- Null call squelching
- Measurements of accuracy & efficiency
Specification-Based Monitoring

• **Specify constraints upon program behavior**
  - Static analysis of binary code
  - *Construct automaton modeling all system call sequences the program can generate*

• **Ensure execution does not violate specification**
  - Operate the automaton
  - If no valid states, then intrusion attempt occurred
Architecture

User Program

EEL
- Parse Binary
- Build CFGs
- Memory Analysis
- Generate Code
- Rewrite
- Generate Rewritten Binary

Clients
- Build DDG
- Build SDG
- Build Program Specification
- Program Spec

1 August 2003
WiSA - Jonathon Giffin
Simplified Architecture

- User Program
- Analyzer
  - Build Program Specification
  - Rewritten Binary
  - Program Spec
Specification-Based Monitoring

- Our **runtime monitor** monitors program execution at some event interface layer
  - Operating system kernel traps
  - Remote system calls
- Ensures program events match specification
Model Construction

- User Program
- Analyzer
  - Build Program Specification
- Rewritten Binary
- Program Spec

Binary Program
Control Flow Graphs
Local Automata
Global Automaton
Code Example

```c
int log_fd;
void log(const char *m)
{
    int s=strlen(m);
    s=write(log_fd,m,s);
}
```

```asm
log:
   save %sp, -96, %sp
   call strlen
   mov %i0, %o0
   mov %o0, %o2
   mov %i0, %o1
   call write
   ld [log_fd], %o0
   ret
   restore
```
int log_fd;
void log(const char *m)
{
    int s=strlen(m);
    s=write(log_fd,m,s);
}
```c
void secure_exec()
{
    struct stat s;
    stat("/sbin/mailconf", &s);
    if (s.st_size == 1013288)
    {
        log("execing conf");
        exec("/sbin/mailconf");
    }
}
```
void safe_link(char *f, *t)
{
    char msg[500];
    unlink(t);
    link(f, t);
    snprintf(msg, 50, "Linked %s to %s", f, t);
    log(msg);
}
NFA Model

safe_link

unlink(?)

link(?,?)

log

write(?,?,?)

log

log

secure_exec

stat("/sbin/mailconf")

log

exec("/sbin/mailconf")
Impossible Paths

unlink("/sbin/mailconf");
link("/bin/sh", "/sbin/mailconf");
write(-1, 0, 0);
exec("/sbin/mailconf");

Bypasses stat check in secure_exec.
PDA Model

safe_link

unlink(?)

link(?,?)

push X

push Y

write(?,?,?)

log

ε

ε

ε

ε

pop X

pop Y

secure_exec

stat(“/sbin/mailconf”)

exec(“/sbin/mailconf”)
PDA State Explosion

• \(\varepsilon\)-edge identifiers maintained on a stack
  - Stack may grow to be unbounded

• First solution
  - Use stack abstractions: bound stack height
  - Practically limited to size 0 [NFA]

• Null call insertion reduced non-determinism
  - Increased stack bound to 4
Dyck Model

- Efficiently tracks calling context
- As powerful as full PDA
- Efficiency approaches NFA model
Dyck Model

• Bracketed context-free language
  [Ginsberg & Harrison 67]

\[
[y \text{ write }]_y \text{ exec }
\]
\[
\text{unlink symlink } [x \text{ write }]_x
\]

• Matching brackets are alphabet symbols
  - Null calls can serve this purpose
  - Exposes stack operations to runtime monitor
Dyck Model

safe_link

unlink(?)

link(?,?)

log

write(?,?,?)

secure_exec

stat("/sbin/mailconf")

exec("/sbin/mailconf")

Dyck Model

exec("/sbin/mailconf")

stat("/sbin/mailconf")

secure_exec

write(?,?,?)

log

link(?,?)

unlink(?)

safe_link
Dyck Model

• Subsequences of bracket symbols form a **Dyck language**
  [Chomsky & Scheutzenberger 63]
  [von Dyck 1882]

• Dyck languages maintain PDA expressiveness
  [Chomsky 62]

\[ L = h(D \cap R) \]
Binary Rewriting

Binary Program

Analyzer
Build Program Specification

Rewritten Binary

Program Spec

Binary Program

Rewritten Binary
void secure_exec()
{
    struct stat s;
    stat("/sbin/mailconf", &s);
    if (s.st_size == 1013288)
    {
        leftX();
        log("execing conf");
        rightX();
        exec("/sbin/mailconf");
    }
}
Null Call Squelching

- Naïve Dyck call insertion leads to high cost
- Null call squelching eliminates irrelevant null calls

Relevant: \[ ... [x \text{ write }]_x ... \]

Irrelevant: \[ ... [x ]_x ... \]
Null Call Squelching

Upper bound:

- The squelched Dyck model generates at most $2hn$ null calls
  - $h$ is the diameter of the call graph
  - $n$ is the number of true system calls executed
## Test Programs

<table>
<thead>
<tr>
<th>Program</th>
<th>Number of Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>procmail</td>
<td>107,246</td>
</tr>
<tr>
<td>gzip</td>
<td>56,710</td>
</tr>
<tr>
<td>eject</td>
<td>70,177</td>
</tr>
<tr>
<td>fdformat</td>
<td>67,874</td>
</tr>
<tr>
<td>ps</td>
<td>59,814</td>
</tr>
<tr>
<td>cat</td>
<td>54,028</td>
</tr>
</tbody>
</table>
Number of Monitor Calls Generated

Number of Calls

- NFA
- Dyck
- Squelched

- procmail
- gzip
- eject
- fdformat
- ps
- cat

7,346,551
# Runtime Overheads

<table>
<thead>
<tr>
<th>Program</th>
<th>NFA</th>
<th>Original</th>
<th>Increase</th>
<th>Squelched</th>
<th>Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>procmail</td>
<td>0.27</td>
<td>0.70</td>
<td>160 %</td>
<td>0.21</td>
<td>---</td>
</tr>
<tr>
<td>gzip</td>
<td>7.36</td>
<td>1381.33</td>
<td>18700 %</td>
<td>8.80</td>
<td>19.6%</td>
</tr>
<tr>
<td>eject</td>
<td>5.67</td>
<td>5.44</td>
<td>---</td>
<td>5.35</td>
<td>---</td>
</tr>
<tr>
<td>fdformat</td>
<td>112.01</td>
<td>112.38</td>
<td>0.3%</td>
<td>112.42</td>
<td>0.4%</td>
</tr>
<tr>
<td>ps</td>
<td>0.36</td>
<td>0.94</td>
<td>160 %</td>
<td>0.31</td>
<td>---</td>
</tr>
<tr>
<td>cat</td>
<td>0.00</td>
<td>0.00</td>
<td>---</td>
<td>0.00</td>
<td>---</td>
</tr>
</tbody>
</table>

Execution times in seconds
Accuracy Metric

- Average branching factor
NFA and Dyck Model Accuracy

Average Branching Factor

NFA
Squelched
Dyck

procmail  gzip  eject  fdformat  ps  cat
Important Ideas

• Statically constructed program models historically compromise accuracy for efficiency.

• The Dyck Model is the first efficient context-sensitive specification.

• Null call squelching bounds the number of Dyck null calls generated.
Specification-Based Monitoring

Jonathon Giffin, Somesh Jha, Barton Miller
University of Wisconsin
{giffin, jha, bart}@cs.wisc.edu

WiSA - Wisconsin Safety Analyzer