Efficient Context-Sensitive Intrusion Detection

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

WiSA - Wisconsin Safety Analyzer
http://www.cs.wisc.edu/wisa
Model-Based Intrusion Detection

- Constructing program models using static binary analysis
- Accuracy/performance tradeoff in prior models
- Our new Dyck model solves tradeoff
- Data-flow analysis to recover arguments
Worldview

- Running processes make operating system requests
- Changes to trusted computing base done via these requests
- Attacker subverts process to generate malicious requests
Our Objective

• Detect malicious activity before harm caused to local machine

• ... before operating system executes malicious system call
Model-Based Intrusion Detection

- Build model of correct program behavior
- Runtime monitor ensures execution does not violate model
- Runtime monitor must be part of trusted computing base
Automated Model Construction

- **Dynamic analysis**
  - Under-approximates correct behavior
  - False alarms
  - Forrest, Sekar, Lee

- **Static analysis**
  - Over-approximates correct behavior
  - False negatives
  - Wagner & Dean, our work
  - Previous attempts at precise models problematic
Automated Model Construction

- **Static analysis challenge**
  - Design an efficient, context-sensitive model

- **Techniques**
  - Dyck model
  - Argument recovery
Our Approach

• **Build model of correct program behavior**
  - Static analysis of binary code
  - Construct an automaton modeling all system call sequences the program can generate

• **Ensure execution does not violate model**
  - Use automaton to monitor system calls.
  - If automaton reaches an invalid state, then an intrusion attempt occurred.
Program Analysis

User Program

Analyzer

Rewritten Program

Program Model
Model Construction

User Program

Analyzer

Rewritten Program

Program Model

Binary Program

Control Flow Graphs

Local Automata

Global Automaton
Impossible Path

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

unlink(?)

link(?,?)

done

log

write(?,?,?,?)

done

exec \_wrap

stat("/sbin/mailconf")

done

e

e

e

link \_wrap exec \_wrap

push X

push Y

pop X

pop Y

e

exec("/sbin/mailconf")
PDA State Explosion

• e-edge identifiers maintained on a stack
  - Stack non-determinism is expensive
  - Unbounded stacks add complexity
  - Best-known algorithm: cubic in automaton size

• Unusable as program model
  - Orders of magnitude slowing of application
    • [Wagner et al. 01, Giffin et al. 02]
  - Conclusion: only weaker NFA models have reasonable performance
Dyck Model

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

Implication: accuracy & performance can coexist
- Invalidates previous conclusion
Dyck Model

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

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

• Matching brackets are alphabet symbols
  - Exposes stack operations to runtime monitor
  - Rewrite binary to generate bracket symbols
null
Binary Rewriting

Binary Program \rightarrow User Program \rightarrow Analyzer \rightarrow Rewritten Program \rightarrow Program Model

Binary Program \rightarrow Rewritten Binary
Dyck Null Call Insertion

- Insert code to generate bracket symbols around function call sites
- Notify monitor of stack activity
- Null call squelching prevents high cost

```c
void
link_wrap(char *f, char *t)
{
    char msg[BUFFSIZE];
    unlink(t);
    link(f, t);
    snprintf(msg, BUFFSIZE,
        "Linked %s to %s, f, t);
    leftX();
    log(msg);
    rightX();
}
```
## Test Programs

<table>
<thead>
<tr>
<th>Program</th>
<th>Number of Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>procmail</td>
<td>112,951</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>cat</td>
<td>52,028</td>
</tr>
</tbody>
</table>
# Runtime Overheads

Execution times in seconds

<table>
<thead>
<tr>
<th>Program</th>
<th>Base</th>
<th>NFA</th>
<th>Increase</th>
<th>Dyck</th>
<th>Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>procmail</td>
<td>0.42</td>
<td>0.37</td>
<td>0%</td>
<td>0.40</td>
<td>0%</td>
</tr>
<tr>
<td>gzip</td>
<td>7.02</td>
<td>6.61</td>
<td>0%</td>
<td>7.16</td>
<td>2%</td>
</tr>
<tr>
<td>eject</td>
<td>5.14</td>
<td>5.17</td>
<td>1%</td>
<td>5.22</td>
<td>2%</td>
</tr>
<tr>
<td>fdformat</td>
<td>112.41</td>
<td>112.36</td>
<td>0%</td>
<td>112.38</td>
<td>0%</td>
</tr>
<tr>
<td>cat</td>
<td>54.65</td>
<td>56.32</td>
<td>3%</td>
<td>80.78</td>
<td>48%</td>
</tr>
</tbody>
</table>
Accuracy Metric

- Average branching factor
NFA and Dyck Model Accuracy

Average Branching Factor

- procmail
- gzip
- eject
- fdformat
- cat

Legend:
- NFA
- Squelched Dyck
Data-Flow Analysis

• Can use knowledge of argument values to make model more precise.

• Use data-flow analysis of arguments:
  - Argument recovery
    • Sets of constant values
    • Sets of regular expression strings
  - System call return values that control branching
  - Argument dependencies upon system call return values
Data-Flow Analysis

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

```c
int log_fd;
void log(const char *m)
{
    int s=strlen(m);
    s=write(log_fd,m,s);
    if (s<=0)
        exit(1);
}
```
Argument Dependencies

\[
\begin{align*}
\text{fd1} &= \text{open}("/home/foo", \text{O_RDWR}); \\
\text{fd2} &= \text{open}("/etc/passwd", \text{O_RDWR}); \\
\text{read}(\text{fd2}, \text{buf}, \text{BUFSIZE}); \\
\text{write}(\text{fd1}, \text{buf}, \text{BUFSIZE}); \\
\text{open}_1() &= 3; \\
\text{open}_2() &= 4;
\end{align*}
\]
## Data Flow Sensitivity

<table>
<thead>
<tr>
<th>Program</th>
<th>Number of System Call Sites</th>
<th>Number Affecting Branches</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>procmail</td>
<td>203</td>
<td>97</td>
<td>48%</td>
</tr>
<tr>
<td>gzip</td>
<td>96</td>
<td>54</td>
<td>56%</td>
</tr>
<tr>
<td>eject</td>
<td>159</td>
<td>101</td>
<td>64%</td>
</tr>
<tr>
<td>fdformat</td>
<td>197</td>
<td>103</td>
<td>52%</td>
</tr>
<tr>
<td>cat</td>
<td>108</td>
<td>45</td>
<td>42%</td>
</tr>
</tbody>
</table>
Effects of Argument Capture (Squelched Dyck Model)

Average Branching Factor

- procmail
- gzip
- eject
- fdformat
- cat

- No Capture
- Intraprocedural Analysis, Constant Values
- Interprocedural Analysis, General Values

5 February 2004
WiSA - Jonathon T. Giffin
Important Ideas

• Model-based intrusion detection forces execution behavior to match model.

• Statically constructed program models historically compromise accuracy for efficiency.

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

• Data-flow analysis restricts undetected attacks by improving model precision.
Efficient Context-Sensitive Intrusion Detection

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

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