Efficient Context-Sensitive Intrusion Detection

Jonathon T. Giffin  
Somesh Jha  
Barton P. Miller

University of Wisconsin

{giffin,jha,bart}@cs.wisc.edu
Intrusion Detection Problems

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

• Detection efficiency: Can subversion be detected in real-time?
Our Solution

Model-based anomaly detection

• 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
  - Strong theoretical foundation
  - Demonstrated that program state exposure improves performance
  - Two new papers published
    • [NDSS 2004, Oakland 2004]

• **Static analysis infrastructure**
  - Analyze dynamically-linked executables
  - Build Dyck models without binary rewriting
Overview

Attacks
• Server attack (conventional host-based IDS)
• Remote execution attack (remote IDS)

Model-based intrusion detection
• Constructing program models using static binary analysis
• Accuracy/performance tradeoff in prior models
• Deterministic PDA models solve tradeoff
• Combining static & dynamic analysis
Worldview

• Running processes make operating system requests

• Changes to trusted computing base done via these requests

• Attacker subverts process to generate malicious requests
Example: Server Attack

- **Goal:** Execute malicious code in the server
Example: Remote Execution Attack

![Diagram showing the relationship between Shadow Process and User Process in a remote execution attack. The Shadow Process is labeled with 'giffin', the User Process with 'nobody', and the system calls are shown with an arrow labeled 'system calls'.]
Example: Remote Execution Attack

- **A**: Submitting Host
  - Shadow Process: `giffin`

- **C**: Remote Execution Host
  - Malicious User Process: `nobody`
  - Lurker Process: `nobody`

- **System Calls** from `giffin` to `nobody`

- **Fork** operation from `nobody`
Example: Remote Execution Attack

Shadow Process

* bart

rm -rf *

Submitting Host

system calls

Innocent User Process

* nobody

Remote Execution Host

Control remote system calls

Lurker Process

* nobody

attach
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

- Model: automaton specifying all valid system call sequences

- Runtime monitor ensures execution does not violate model
Model-Based Intrusion Detection

- Model must be fast to operate

- Model must accurately represent program
  - Context-sensitive models restrict impossible paths
Automated Model Construction

- Learn via training runs
  - Under-approximates correct behavior
  - False alarms
  - Forrest, Sekar, Lee

- Static code 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, accurate model

- **Answers**
  - Dyck model
  - Data flow analysis to recover arguments
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.
Model-Based Intrusion Detection

User Program

Analyzer

Rewritten Program

Program Model
Model Construction

Analyzer

User Program

Rewritten Program

Program Model

Binary Program

Control Flow Graphs

Local Automata

Global Automaton
char *filename;

pid_t[2] pid;

int prepare (int index) {
    char buf[20];
    pid[index] = getpid();
    strcpy(buf, filename);
    return open(buf, O_RDWR);
}
void action (void) {
    uid_t uid = getuid();
    int handle;

    if (uid != 0) {
        handle = prepare(1);
        read(handle, ...);
    } else {
        handle = prepare(0);
        write(handle, ...);
    }

    close(handle);
}
NFA Model

getuid

ɛ

getpid

ɛ

ɛ

open

read

write

close
Impossible Path Exploit

```c
void action (void) {
    uid_t uid = getuid();
    int handle;

    if (uid != 0) {
        handle = prepare(1);
        read(handle, ...);
    } else {
        handle = prepare(0);
        write(handle, ...);
    }
    close(handle);
}
```
PDA Model

- getuid → push Y
- push Y → push X
- push X → getpid
- getpid → open
- open → read
- read → write
- write → close
- close
**PDA Problems**

- Impossible paths still exist
  - Non-determinism indicates missing execution information

- PDA run-time state explosion
  - $\varepsilon$-edge identifiers maintained on a stack
  - Stack non-determinism is expensive
  - post* algorithm: cubic in automaton size
PDA Problems

• **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
## Some Recent History…

<table>
<thead>
<tr>
<th>Year</th>
<th>Model</th>
<th>Speed</th>
<th>Imp Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001</td>
<td>Wagner</td>
<td>NFA</td>
<td>Moderate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PDA</td>
<td>Slow</td>
</tr>
<tr>
<td></td>
<td>Sekar</td>
<td>DFA</td>
<td>Fast</td>
</tr>
<tr>
<td>2002</td>
<td>Giffin</td>
<td>NFA</td>
<td>Fast</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BPDA</td>
<td>Moderate</td>
</tr>
<tr>
<td>2003</td>
<td>Feng</td>
<td>VtPaths</td>
<td>Fast</td>
</tr>
<tr>
<td>2004</td>
<td>Giffin</td>
<td>Dyck</td>
<td>Fast</td>
</tr>
<tr>
<td></td>
<td>Feng</td>
<td>VPStatic</td>
<td>Fast</td>
</tr>
</tbody>
</table>
State non-determinism is cheap.

NFA

State non-determinism
Non-Deterministic PDA

State non-determinism

Stack non-determinism

Stack non-determinism is expensive.
Deterministic PDA (DPDA)

- Model exposes stack operations & target states
- Possible exponential increase in model size
Deterministic PDA (DPDA)

- Replace $\varepsilon$-edges with symbol describing stack operation & target state
Deterministic PDA (DPDA)

- Replace $\varepsilon$-edges with symbol describing stack operation & target state.
- Input symbol describes:
  - $[X]$: How to update stack
  - $/1$ : How to traverse automaton transitions
Stack-Deterministic PDA (sDPDA)

Dyck Model

- Model exposes stack operations
- No increase in model size
Stack-Deterministic PDA (sDPDA)
Input Symbol Processing Complexity

<table>
<thead>
<tr>
<th>Model</th>
<th>Time Complexity</th>
<th>Space Complexity</th>
<th>Input Alphabet Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDA</td>
<td>$O(nm^2)$</td>
<td>$O(nm^2)$</td>
<td>$k$</td>
</tr>
<tr>
<td>DPDA</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$\Theta(knr)$</td>
</tr>
<tr>
<td>sDPDA</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$\Theta(kr)$</td>
</tr>
</tbody>
</table>

- $n$ is state count
- $m$ is transition count
- $k$ is PDA input alphabet size
- $r$ is PDA stack alphabet size
Dyck Model

\[ \text{getuid } [X \text{ getpid open } ]_X \text{ read close} \]
\[ \text{getuid } [Y \text{ getpid open } ]_Y \text{ write close} \]

- Matching brackets are alphabet symbols
  - Expose stack operations to runtime monitor
  - Language of bracket symbols is a Dyck language
  - Rewrite binary to generate bracket symbols
Binary Rewriting

User Program

Analyzer

Rewritten Program

Program Model

Binary Program → Rewritten Binary
Determinizing via Binary Rewriting

- Insert code to generate bracket symbols around function call sites
- Notify monitor of stack activity
- Determinizes stack operations

```c
void action (void) {
    uid_t uid = getuid();
    int handle;

    if (uid != 0) {
        precall(X);
        handle = prepare(1);
        postcall(X);
        read(handle, ...);
    } else {
        precall(Y);
        handle = prepare(0);
        postcall(Y);
        write(handle, ...);
    }

    close(handle);
}
```
void action (void) {
    uid_t uid = getuid();
    int handle;

    if (uid != 0) {
        precall(X);
        handle = prepare(1);
        postcall(X);
        read(handle, ...);
    } else {
        precall(Y);
        handle = prepare(0);
        postcall(Y);
        write(handle, ...);
    }

    close(handle);
}

Determinizing via Binary Rewriting

• Dyck null calls meaningful only when prepare generates system calls

Relevant:
... [x write ]x ...
Determinizing via Binary Rewriting

- Maintain **history stack** in rewritten binary
- Records null calls encountered since last system call

```c
void action (void) {
    uid_t uid = getuid();
    int handle;

    if (uid != 0) {
        precall(X);
        handle = prepare(1);
        postcall(X);
        read(handle, ...);
    } else {
        precall(Y);
        handle = prepare(0);
        postcall(Y);
        write(handle, ...);
    }
}
```

**Relevant:**

```
... [x write ]_x ... 
```

**Irrelevant:**

```
... [x ]_x ... 
```
Null Call Squelching

Relevant:

\[
\ldots \quad [x \text{ write }]_x \quad \ldots
\]

History Stack:

\[
[ \ldots ]_x \quad ]_B \quad [C \quad [x
\]

16 August 2004  Jonathon Giffin
Null Call Squelching

Irrelevant:

... [x] [x] ...

History Stack:

[ ]_A [ ]_B [ ]_C [ ]_x [ ]_x

Squelching bounds number of null calls produced
Determinizing via Stackwalks

- Recover stored return values by walking the call stack of the running process

Current: $0x1003c, 0x318f0, 0x22cd8$
Determinizing via Stackwalks

• Recover stored return values by walking the call stack of the running process

Current:

0x1003c, 0x318f0, 0x22cd8

Previous:

0x1003c, 0x29af4

• Compare to previous stack to generate pop and push input symbols

• Requires no binary rewriting
Determinizing via Stackwalks

• Recover stored return values by walking the call stack of the running process

<table>
<thead>
<tr>
<th>Current:</th>
<th>Previous:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1003c, 0x318f0, 0x22cd8</td>
<td>0x1003c, 0x29af4</td>
</tr>
</tbody>
</table>

• Compare to previous stack to generate pop and push input symbols

• Requires no binary rewriting
Dyck Model

Dyck model stack-determinizes PDA

Stack determinism

Only one valid stack configuration

Stack non-determinism
Study 1: Dyck Model

<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>

- Squelching removed
- 7.3 million symbols

- High null call count
- Workload specific
Accuracy Metric

- Average branching factor

getpid

chown

open
Study 2: DPDA vs. sDPDA

<table>
<thead>
<tr>
<th>Program</th>
<th>Number of Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>htzipd</td>
<td>110,096</td>
</tr>
<tr>
<td>gzip</td>
<td>57,271</td>
</tr>
<tr>
<td>cat</td>
<td>52,601</td>
</tr>
</tbody>
</table>
Program Execution Time Overheads (% Increase)

- **htzipd**: VPStatic increase
- **gzip**: VPStatic increase
- **cat**: VPStatic increase

Increase in Execution Time (%)
Program Memory Use Overheads (% Increase)

Increase in Memory Use (%)

- VPStatic
- Dyck

htzipd
gzip
cat
Analysis Combinations

- **Static analysis**
  - Conservative, nearly sound
  - Incorporating configuration information requires expensive analyses

- **Dynamic analysis**
  - Under-approximation produces false alarms
  - Reveals how configuration settings affect execution
Analysis Combinations

- Combined model
  - Dynamic: identify system call arguments
  - Dynamic: identify program branch behavior
  - Static: build Dyck model with added restrictions from dynamic analyses

- Joint work with Wenke Lee
Important Ideas

• Formalizing program models facilitates understanding & comparison.

• Exposing additional program state improves monitoring speed & model accuracy.