Memoization Attacks and Copy Protection in Partitioned Applications

Charles W. O’Donnell\textsuperscript{1}, G. Edward Suh\textsuperscript{2}
Marten van Dijk\textsuperscript{1}, Srinivas Devadas\textsuperscript{1}

\textsuperscript{1}Massachusetts Institute of Technology
\textsuperscript{2}Cornell University

IEEE Workshop on Information Assurance
June 22, 2007
Motivation

- **Central concern:** Intellectual Property (IP) Protection of applications
  - Prevent piracy, hide sensitive algorithms, etc

- Stop attacker from reproducing functionality of "protected" software code
  - Only some small regions of application may need protection

- **Operational functionality:** ultimate test of security
  - Unimportant: contents of protected code
  - Important: How protected code is used,
    How attacker can bypass code and still get "useful" results

- One solution: Fully encrypt application
  - Requires: Secure CPU/Co-Processor, remote servers
  - Prevents piracy by requiring a key to execute
  - Speed/power/etc **overheads**
Partitioned Applications

Partitioned Application: only encrypt portions of application
- May provide same security
- Tradeoff security vs. speed

Architecture guarantees secret execution of encrypted code
- Only memory accesses in and out of encrypted code region are visible
- More details later

Central Question: Deciding which regions of an application to encrypt

Key Point: Naïve separation insecure
- Designers must make a balanced decision based on how encrypted region will be used in the application at large

addi r3, r4, 16
lw r5, 0(r15)
sub r6, r5, r3
sti 4(r15), r6
addi r11, r6, r5
Presentation Outline

- Model
  - Define partitioned application and a very limited adversary

- Memoization Attacks
  - Describe problem and method of attack

- Implementing a Memoization Attack
  - Practical issues when performing attack
  - Attack results on real applications

- Indicators of Insecurity
  - Simple omens for when a Memoization Attack will succeed
  - Indicator accuracy results on real applications

- Related Work
  - Long standing research problem

Partitioned Applications Details

- Application code
  - encrypted *private* regions
  - unencrypted *public* regions

- Private regions
  - Executes *secretly*
  - Access special private memory *secretly*
  - Can access regular public memory

Simplifying assumptions:
- *Procedures* are fundamental region units
- *No private state between calls*  (Common case)
- For experiments: in-order memory, no cache

Adversary observes memory bus to attack

---

Observing a Partitioned Application

Execution Trace

- read(A)
- write(B)
- call-priv(A)
- args(A)
- read(B)
- write(C)
- read(C)
- write(D)
- exit()
- read(D)
- write(E)
- ...

Memory

Public Memory
- A
- B
- D
- E

Private Memory
- C

Private Call

Public Memory
What an Adversary Knows

- Adversary can observe memory accesses
  - But what does he “know” about secret region?

- Unlimited possible models…
  - We analyze *weakest* form of adversary, *no priors*
  - This still enough to perform a successful attack

- Our adversary:
  - Can only observe application execution for *reasonable* (polynomial) amount of time
  - Has only limited (polynomial) storage space
  - Has only limited (polynomial) computational power

- Our experiments used one standard x86 server (no farm jobs, etc)
Memoization Attacks

- **Procedures only a set of input-output mappings**

- **Observe** application, remembering inputs and outputs in table
  - Then replace private code and **emulate**

- However, such a simple table is not enough...
Implementing a Memoization Attack

Two main problems

- Input self-determination
- Keeping the “Interaction Table” small

**Input self-determination**

Private procedure

\[
F(a) :
\begin{align*}
\text{if (a):} & \quad b \leftarrow [Z] \\
\text{else:} & \quad b \leftarrow [Y] \\
\text{return (2*b)}
\end{align*}
\]

Two possible input sets

\{a = ?, [Z] = ?\}
\{a = ?, [Y] = ?\}

Naïve solution too costly

\{a = ?, [Y] = ?, [Z] = ?\}

Emulating procedure requires *order* information

- Temporal Memoization

Temporal Memoization

Call 1

r1 = fff4
r2 = 7
...
read[A]=5
read[B]=12
read[C]=54
write[Z]=0
set r11 = 1

Call 2

r1 = fff4
r2 = 7
...
read[A]=5
read[B]=12
read[C]=64
write[Z]=8
set r11 = 1

Call 3

r1 = fff4
r2 = 3
...
read[D]=1
read[E]=24
read[F]=20
set r11 = 8

Call 4

r1 = fff4
r2 = 7
...
read[A]=6
read[B]=30
read[G]=50
write[X]=0
set r11 = 4

Emulation:

<table>
<thead>
<tr>
<th>step</th>
<th>①</th>
<th>②</th>
<th>③</th>
<th>④</th>
</tr>
</thead>
<tbody>
<tr>
<td>reads</td>
<td>r1 = fff4</td>
<td>A = 5</td>
<td>B = 12</td>
<td>C = 64</td>
</tr>
<tr>
<td></td>
<td>r2 = 7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>writes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Z = 8 , r11 = 1</td>
</tr>
</tbody>
</table>
Interaction Table Compression

*Keeping the Interaction Table small*

- Table can become huge
- Contains many redundancies

- Instead of table columns, think of execution trace tree
  - Branches in tree occur on *reads*
  - since they solely determine control flow
Interaction Tree Construction

Observed Calls

1. $r_1 = \text{fff4}$
   - read( $A$, 5 )
   - read( $B$, 30 )
   - read( $C$, 54 )
   - write( $Z$, 8 )
   ...

2. $r_1 = \text{fff4}$
   - read( $A$, 10 )
   - read( $C$, 54 )
   - read( $B$, 30 )
   - write( $Z$, 4 )
   ...

3. $r_1 = \text{fff4}$
   - read( $A$, 5 )
   - read( $B$, 77 )
   - write( $Z$, 0 )
   - read( $C$, 54 )
   ...

Compressing the Interaction Tree

- Tree still redundant
Compressing the Interaction Tree

- Tree still redundant
- Introduce path numbers
  (more in paper)
Results of Memoization Attacks

Memoization Attacks can work on some, but not all applications.

Two “types” effected most (defined by context):

- **Partially repeated input sets** (external workloads)
  - Repeats functionality or input workload

- **Compositing input sets** (external workloads)
  - If a few input sets to application cover the input space of single procedure, bounded set of possible inputs
  - If application inputs filtered before reaching private call
  - More dangerous since non-intuitive
Effectiveness on Repeated Workloads

SPEC CPU2000 Parser:

- `special_command()` - Memoization Attack always succeeds
  - Repeats same functionality, changes internal settings

- `is_equal()` – Memoization Attack always succeeds
  - Only run over dictionary data (checks for special tokens)

Size of structures manageable:

<table>
<thead>
<tr>
<th>Size Metric</th>
<th>Parser: <code>special_command()</code></th>
<th>Parser: <code>is_equal()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of tree nodes (compressed)</td>
<td>283</td>
<td>5</td>
</tr>
<tr>
<td>Size on disk</td>
<td>26,972 Bytes</td>
<td>2,042,968 Bytes</td>
</tr>
<tr>
<td>Maximum depth of expanded tree</td>
<td>743</td>
<td>5</td>
</tr>
</tbody>
</table>
## Effectiveness on Composite Workloads

### SPEC CPU2000 Gzip `bi_reverse()`
- Called when working on entire dataset (bit manipulation)
- Memoization Attack successful on 97% of calls

### SPEC CPU2000 Parser `contains_one()`
- Called for every new input
- Memoization Attack successful on 33% of calls

<table>
<thead>
<tr>
<th>Gzip: <code>bi_reverse()</code></th>
<th>Emulating: <code>ref.log</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Observed Inputs</td>
<td>Emulatable Calls</td>
</tr>
<tr>
<td>random</td>
<td>681 / 1797 38%</td>
</tr>
<tr>
<td>random, graphic</td>
<td>1362 / 1797 76%</td>
</tr>
<tr>
<td>random, graphic, program</td>
<td>1518 / 1797 84%</td>
</tr>
<tr>
<td>random, graphic, program, source</td>
<td>1741 / 1797 97%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parser: <code>contains_one()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Workload: <code>lgred.in</code></td>
</tr>
<tr>
<td>Emulating: <code>smred.in</code></td>
</tr>
<tr>
<td>Workload: <code>lgred.in</code></td>
</tr>
<tr>
<td>Emulating: <code>mdred.in</code></td>
</tr>
</tbody>
</table>
Indicators of Insecurity

Memoization Attack feasible
  - But can’t prove exactly when it will work...

Which procedures will it work for?
  - Running attack to determine is computationally intensive
  - Instead, use indicators that give suggestion of success
    - We give two, but many more possible

Tests show negative results
  - Cannot show positive security (especially given heuristics)

Tests should be
  - computationally simple
  - numerous and self-supporting
Input Saturation

- **Count** unique input values seen by procedure
  - Indicates cost/size of Interaction Tree
- Many ways to estimate input values
  - Our experiment simply counted on few executions

- **Plot** or **"Saturation Weight"** describes count

\[
SW = \frac{1}{N \omega(N)} \int_0^N \omega(c) dc
\]

- **Saturating** when \( SW = 1.0 \)
Results of Input Saturation on Gzip

- Some clearly saturate, others clearly do not
- Some ambiguous → needs more testing

<table>
<thead>
<tr>
<th>Procedure</th>
<th>SW</th>
</tr>
</thead>
<tbody>
<tr>
<td>bi_reverse</td>
<td>0.99</td>
</tr>
<tr>
<td>ct_tally</td>
<td>0.87</td>
</tr>
<tr>
<td>huft_build</td>
<td>0.72</td>
</tr>
<tr>
<td>build_tree</td>
<td>0.51</td>
</tr>
<tr>
<td>longest_match</td>
<td>0.51</td>
</tr>
</tbody>
</table>

![Graph showing normalized number of unique AV pairs read vs. normalized number of procedure calls]
Data Egress

- Output possibly more indicative of complexity than input
- Count unique data created by procedure and data’s importance to rest of program (use for both control & final value)

Egress Weight: \[ \Phi(\eta) = \sum_{(t_i, \kappa_i) \in \eta} \frac{\kappa_i}{t_i} \]

- higher = harder to attack (compared against other procedures in single app)
Results of Data Egress on Gzip

- Both high and low Egress Weights
- Inconsistencies and similarities when compared with Saturation Weight
  - **Lesson:** Must use multiple metrics
- Real attack: *bi_reverse* almost 100%, *ct_tally* tiny success

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Total Unique Writes</th>
<th>Public Readers</th>
<th>$\Phi$ weight</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>bi_reverse</em></td>
<td>259</td>
<td>2</td>
<td>93</td>
</tr>
<tr>
<td><em>ct_tally</em></td>
<td>4,214,758</td>
<td>4</td>
<td>1,343,144</td>
</tr>
<tr>
<td><em>huft_build</em></td>
<td>59,224</td>
<td>4</td>
<td>96</td>
</tr>
<tr>
<td><em>build_tree</em></td>
<td>21,000</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td><em>longest_match</em></td>
<td>515</td>
<td>1</td>
<td>13,010</td>
</tr>
</tbody>
</table>

**Input Saturation**

Related Work – Secrecy & Piracy

Four major areas – By far, incomplete list, showing most related

Software Secrecy

- Gosler – Defined problem, deconstructing [1986]
- Kent – Encrypted processor [1981]

Software Piracy

- Microsoft, others – Online verification [recent]
- Lie, TCG, NGSCB – Tie code to physical CPU [2000-present]
**Related Work – Partitioning & Complexity**

**Program Partitioning**
- Yee – Partitioning for secure coprocessors [1994]
- White, et al – ABYSS, separations for security [1990]
- Zhang, et al – Program slicing for piracy [2003]
- Brumley, et al – Privtrans, monitor-slave separation [2004]
- Zdancewic, et al – For end-to-end information flow [2002]
- Ori Dvir, et al – Remote memory allocation [2005]

**Application Complexity**
- McCabe
- Kent
- Henry, et al
- Munson, et al
Conclusions

Partitioned Applications are not automatically “secure”
  - Secret code can be reconstructed

Memoization Attacks are feasible and non-trivial
  - Even when using a weak adversary with no heuristics
    - Although they cannot always succeed
  - Can be implemented and performed on a regular computer
  - Repeated Workloads very easily emulated
  - Composite Workloads also can be emulated

Simple tests indicate when Memoization Attacks might succeed
  - Easier to perform than full attack
  - But, not a guarantee (use many tests)
  - Can aid software designer
Tree from Hidden Control Flow Graph

Private Procedure

Interaction Tree

Observed Sequences

\{A, B, C, E, B, C, F\}
\{A, B, D, E, F\}
\{A, B, D, E, B, C, F\}
\{A, B, C, E, F\}

Interaction Tree Construction Steps

Observed Calls

1. \( r_1 = \text{fff4} \)
   - \text{read( A, 5 )}
   - \text{read( B, 30 )}
   - \text{read( C, 54 )}
   - \text{write( Z, 8 )}
   - ...

2. \( r_1 = \text{fff4} \)
   - \text{read( A, 10 )}
   - \text{read( C, 54 )}
   - \text{read( B, 30 )}
   - \text{write( Z, 4 )}
   - ...

3. \( r_1 = \text{fff4} \)
   - \text{read( A, 5 )}
   - \text{read( B, 77 )}
   - \text{write( Z, 0 )}
   - \text{read( C, 54 )}
   - ...

Diagram:

- **Node A**
  - \text{read( A, 5 )}
  - \text{read( B, 10 )}
  - \text{read( C, 54 )}
  - \text{write( Z, 8 )}
  - \text{write( Z, 4 )}

- **Node B**
  - \text{read( A, 30 )}
  - \text{read( B, 77 )}
  - \text{write( Z, 0 )}

- **Node C**
  - \text{read( A, 54 )}
  - \text{write( Z, 8 )}

Emulating with Interaction Tree

**Emulation:**
- \( r_1 = \text{fff4} \)
- \( A = 5 \)
- \( B = 77 \)
- \( \text{write}(Z, 0) \)
- \( C = 54 \)
- ...
Interaction Table Path Numbers

- Path numbers enable joins and loops in Interaction Tree
- Each path number refers to unique branch of un-compressed tree
- Nodes in Interaction Table can contain multiple path numbers

<table>
<thead>
<tr>
<th>Address</th>
<th>Read Value</th>
<th>Write AV Pairs</th>
<th>Path Numbers</th>
<th>Next Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>0xffff4</td>
<td>-</td>
<td>{0 \rightarrow 1}</td>
<td>r3</td>
</tr>
<tr>
<td></td>
<td>0xffc0</td>
<td>-</td>
<td>{0 \rightarrow 2}</td>
<td>r3</td>
</tr>
<tr>
<td>r3</td>
<td>0x7</td>
<td>(0x4410, 0x1e)</td>
<td>{1}</td>
<td>0x4072</td>
</tr>
<tr>
<td></td>
<td>0x7</td>
<td>(0x4420, 0x60)</td>
<td>{2}</td>
<td>0x4104</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0x4424, 0x0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x3</td>
<td>-</td>
<td>{1 \rightarrow 4}</td>
<td>0x4100</td>
</tr>
<tr>
<td></td>
<td>0x3</td>
<td>(0x4420, 0x5c)</td>
<td>{2 \rightarrow 5}</td>
<td>0x4100</td>
</tr>
<tr>
<td>0x4072</td>
<td>0x1</td>
<td>-</td>
<td>{1,...}</td>
<td>0x4100</td>
</tr>
<tr>
<td></td>
<td>0x2</td>
<td>-</td>
<td>{1 \rightarrow 3,...}</td>
<td>0x4100</td>
</tr>
<tr>
<td>0x4100</td>
<td>0x20</td>
<td>-</td>
<td>{5,...}</td>
<td>0x4088</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Repeated/Composite Workloads

Repeated Functionality:

Multiple Workloads: