STAnalyzer: A Simple Static Analysis Tool for Detecting Cache-Timing Leakages

Alexander Schaub, Olivier Rioul & Sylvain Guilley
June 24, 2019
Layout of the Presentation

Cache-Timing Attacks
- Introduction
- Example Vulnerable Code

Static Code-Analysis
- Problem Statement
- Semantics
- Limitations

Results
- Analysis of First Round NIST PQC Standardization Candidates

Conclusion
Layout of the Presentation

Cache-Timing Attacks
   Introduction
   Example Vulnerable Code

Static Code-Analysis
   Problem Statement
   Semantics
   Limitations

Results
   Analysis of First Round NIST PQC Standardization Candidates

Conclusion
Figure: Per-process memory isolation.
Memory Sharing

Physical Memory

![Diagram of memory sharing]

Figure: Shared memory (dynamically-linked libraries, page duplication,...)
Cache-Line Sharing

Figure: Cache-line sharing between processes.
How to Determine the Presence of Data in the Cache?

Several techniques exist, for instance:

- PRIME + PROBE\(^1\),\(^2\)
- EVICT + TIME\(^3\)
- FLUSH + RELOAD\(^3\)

Example to follow...


### Example: FLUSH+RELOAD

<table>
<thead>
<tr>
<th>Attacker</th>
<th>Victim</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>clflush addr</code></td>
<td></td>
<td><code>addr</code> absent from cache</td>
</tr>
<tr>
<td></td>
<td><code>executes code</code></td>
<td><code>addr</code> might be present</td>
</tr>
<tr>
<td><code>a = rdtsc()</code></td>
<td></td>
<td>if the load was fast, the attacker now knows that <code>addr</code> was accessed</td>
</tr>
<tr>
<td><code>load addr</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>store rdtsc() - a</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>clflush addr</code></td>
<td></td>
<td><code>addr</code> absent from cache</td>
</tr>
<tr>
<td></td>
<td><code>executes code</code></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Recognizing Vulnerable Code

<table>
<thead>
<tr>
<th>How</th>
<th>Data</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exploit</td>
<td>Sensitive indirections</td>
<td>Conditional jump/call</td>
</tr>
<tr>
<td>Reason</td>
<td>Memory load</td>
<td>Code execution</td>
</tr>
<tr>
<td>Code vulnerability</td>
<td>Dereferencing a pointer to a secret-dependent address</td>
<td>Branching on a secret-dependent condition</td>
</tr>
</tbody>
</table>

Note: FLUSH + RELOAD only applicable to **shared** data or code (static arrays, code in shared dynamic libraries, etc.)
Vulnerable Code

\textbf{static} \texttt{gf_t}[] \texttt{gf\_antilog} = {...};
\textbf{static} \texttt{gf_t}[] \texttt{gf\_log} = {...};

\texttt{gf_t \_gf\_mult(gf\_t \_a, gf\_t \_b) \{ \}
  \textbf{if} (a == 0 \mathbf{||} b == 0) \textbf{return} 0;
  \textbf{return} \texttt{gf\_antilog[}
  \texttt{gf\_add(gf\_log[a], gf\_log[b])];
\}

\texttt{gf\_t \_gf\_exp(gf\_t \_b, unsigned \_d) \{ \}
  \texttt{gf\_t r = gf\_one();}
  \texttt{mask = 1<<floor(log2(d));}
  \textbf{while}(mask > 0) \{ \}
    \texttt{r = \_gf\_mult(r, r);}
    \textbf{if} (mask \& d > 0) \{ \}
      \texttt{r = \_gf\_mult(a, r);}
    \}
  \texttt{mask /= 2;}
\}

\begin{tikzpicture}
  \node (gf_exp) at (2.5,3) {gf\_exp};
  \node (gf_mult) at (1.5,2) {gf\_mult};
  \node (gf_example_c_26_9) at (2,0.5) {gf\_example.c:26:9};
  \draw[->] (gf_exp) -- (gf_mult);
  \draw[->] (gf_mult) -- (gf_example_c_26_9);

  \node (25) at (2,2) {25 \textbf{while}(mask > 0) \{ \}
  \node (27) at (2,1.5) {27 \textbf{if} (mask \& d > 0) \{ \}
  \node (17) at (2,1) {17 \textbf{if} (a == 0 \mathbf{||} b == 0) \textbf{return} (gf\_t) 0; \}
  \node (18) at (2,0.5) {18 \textbf{return} \texttt{gf\_antilog[gf\_add(gf\_log[a], gf\_log[b])]; \}

  \draw[->] (25) -- (27);
  \draw[->] (27) -- (17);
  \draw[->] (27) -- (18);
\end{tikzpicture}
Layout of the Presentation

Cache-Timing Attacks
  Introduction
  Example Vulnerable Code

Static Code-Analysis
  Problem Statement
  Semantics
  Limitations

Results
  Analysis of First Round NIST PQC Standardization Candidates

Conclusion
Given a C program, with annotations corresponding to sensitive variables, determine whether the program is potentially vulnerable to cache-timing side channel leaks.

Solution should be easy to use, as accurate as possible, and applicable to most cryptographic implementations written in C.
General Approach

- General idea: perform value dependency propagation, and record table accesses / branching operations depending on sensitive data.

- Values tracked for dependency analysis are sensitive values and initial values of function arguments.

- Algorithm consist in tracking the state of three objects during the exploration of the AST:
  - Dependencies between variables and values, as a bipartite graph $G$
  - List of leaking variables, with corresponding code instruction, call graph and dependency chain, $L$
  - "Additional" dependencies, to take branching behavior into account, as a set of values $I$
### Semantics for Simple Operations

<table>
<thead>
<tr>
<th>inst</th>
<th>( G' = \phi_G(G, I; \text{inst}) )</th>
<th>( L' = \phi_L(L, G; \text{inst}) )</th>
<th>( I' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{var} = \text{expr} )</td>
<td>( G \cup {\text{var} \rightarrow G(\langle \text{expr} \rangle) \cup I} )</td>
<td>( L )</td>
<td>( I )</td>
</tr>
<tr>
<td>( \text{var op}_2 \text{=} \text{expr} )</td>
<td>( G \cup {\text{var} \rightarrow G(\langle \text{expr} \rangle) \cup I} )</td>
<td>( L )</td>
<td>( I )</td>
</tr>
<tr>
<td>( \text{var[expr}_1\text{]} = \text{expr}_2 )</td>
<td>( G \cup {\ast \text{var} \rightarrow G(\langle \text{expr}_2 \rangle) \cup I} )</td>
<td>( L \cup G(\langle \text{expr}_1 \rangle) )</td>
<td>( I )</td>
</tr>
<tr>
<td>( \text{if}(\text{expr}){\text{inst}} )</td>
<td>( \phi_G(G, I'; \text{inst}) )</td>
<td>( G(\langle \text{expr} \rangle) \cup \phi_L(L, G; \text{inst}) )</td>
<td>( I \cup G(\langle \text{expr} \rangle) )</td>
</tr>
<tr>
<td>( \text{return} \ \text{expr} )</td>
<td>( G \cup {\text{RET} \rightarrow G(\langle \text{expr} \rangle) \cup I} )</td>
<td>( L )</td>
<td>( I )</td>
</tr>
</tbody>
</table>

Note: analyzing loops consists in computing a fixed point, and a function call in applying a previously determined dependency graph, after translating variable names.
C pointers make the value analysis more complicated - values can be aliased, for instance.

Solution: for each pointer, build a set of memory locations it might point-to.

On every pointer assignment, update this set according to the set of the assignee.

Formalized by Andersen⁴, known as "points-to" analysis.

Might overestimate the set of possible memory locations, but this is necessary in order to avoid false positives.

---

void foo(int a) {
    int *p = malloc(8); // &p: {p}
    int *q = malloc(8); // &p: {p}, &q: {q}

    if (a > 0) {
        q = p; // &p: {p}, &q: {p}
    }
    else {
        p = q; // &p: {q}, &p: {p}
    }
    // &p: {p, q}, &q: {p, q}

    ...
}
Limitations

- Recursive functions not supported
- Complex goto operations not supported (but fixable)
- Casts between different structures, or between different pointer indirections are not correctly handled, e.g. *(int **)p when chasing pointers
- Incorrect or "risky" code could in theory lead to missed leakages, because of buffer overflows, array out-of-bound accesses, or obfuscated pointer arithmetic.
False positives can arise in some situations, for instance when:

- the result of an operation involving sensitive values, is not sensitive itself (the value of $s-s$ does not depend on $s$, or the hash of a sensitive value might not be sensitive)

- dead code is into account, e.g.
  
  ```
  if (condition_that_never_happens) {
    leak_sensitive_value(s);
  } will still count as a leakage
  ```

- conditional code is turned into constant-time code by the compiler
Cache-Timing Attacks
  Introduction
  Example Vulnerable Code

Static Code-Analysis
  Problem Statement
  Semantics
  Limitations

Results
  Analysis of First Round NIST PQC Standardization Candidates

Conclusion
NIST Post-Quantum Cryptography Contest - Overview

- Quantum computers will break asymmetric cryptography
- Alternatives to RSA and ECC need to be developed and vetted for security, evaluated for performance
- 69 algorithms submitted to NIST, mostly lattice-based, code-based and multivariate cryptography
- Selection for the second round announced in January 2019
Results

Vulnerable Implementations

Figure: Total number of potential vulnerabilities found for each analyzed candidate

Note: 52 out of the 69 submissions were analyzed.
Results

Vulnerable Implementations

Out of 52 analyzed candidates:

- Potential vulnerabilities in **42** submissions (80.8%)
  - More than 100 reported vulnerabilities in **17** submissions
  - More than 1000 reported vulnerabilities in **3** submissions

- 4 submissions with easily fixable / probably not exploitable vulnerabilities (EMBLEM, Lima, Giophantus, OKCN-AKCN in the MLWE variant)

- 10 Submissions without detected vulnerabilities (Frodo, Rainbow, Hila5, Saber, CRYSTALS-Kyber, LOTUS, NewHope, ntruprime, ThreeBears and Titanium)
Results
Types of Vulnerabilities

We noticed some repeating patterns in the detected vulnerabilities.

- Gaussian sampling leak
- Other sampling leaks
- GMP library use (at least the standalone implementation)
- Operations in finite fields
- Other: AES re-implementation, matrix operations, error-decoding...
Layout of the Presentation

Cache-Timing Attacks
  Introduction
  Example Vulnerable Code

Static Code-Analysis
  Problem Statement
  Semantics
  Limitations

Results
  Analysis of First Round NIST PQC Standardization Candidates

Conclusion
Conclusion

- We presented STAnalyzer, an algorithm and a tool to detect potential side-channel leakages in C implementations.
- Our program is able to analyze even large, unmodified programs, as shown by our analysis of most post-quantum proposals submitted to NIST.
- There are no missed leaks with this approach, at the cost of a few false positives.
- Not all leakages are exploitable, but assessing their exploitability automatically is a hard problem.
- **Perspective**: combining static analysis techniques with a dynamic analysis could allow us to assess the exploitability of the detected vulnerabilities and provide more information of practical importance.