CacheD: Identifying Cache-Based Timing Channels in Production Software

Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, Dinghao Wu

The Pennsylvania State University

USENIX Security 2017
Outline

• Background
• Overview
• Design
• Evaluation
Cache-based Timing Channel Attack

By (indirectly) observing cache status variants, attackers can infer which secret input is feed to the program.

Cache-based Timing Channel Attack

Leakage due to secret-dependent control flow

\[
\begin{align*}
& r = 1; \\
& \text{for } i \text{ from } n-1 \text{ to } 0 \{ \\
& \quad r = r \cdot r \mod m; \\
& \quad \text{if } (e[i] == 1) \{ \\
& \quad \quad r = r \cdot b \mod m; \\
& \quad \}
\}
\end{align*}
\]

- square-and-multiply implementation of modular exponentiation.
- Real vulnerabilities in RSA and ElGamal implementations.

Leakage due to secret-dependent memory access

\[
\begin{align*}
& \text{id}x = 0; \\
& \text{for } i \text{ from } 0 \text{ to } n-1 \{ \\
& \quad \text{id}x = \text{window} \_ \text{ith}(\text{key}, i); \\
& \quad \text{t} = \text{table}[\text{id}x]; \\
& \}
\end{align*}
\]

- sliding-window-based modular exponentiation.
- Real vulnerabilities in AES, RSA and ElGamal implementations.
Threat Model

Attackers share the same hardware platform with the victim
- A common scenario in the era of cloud computing.
- (Indirectly) learn the trace of cache lines being accessed.

Our threat model is stronger than those based on cache hit/miss
- A trace of cache lines being accessed uniquely determines cache hit/miss at any program point.

Cache line-based abstraction makes the vulnerability analysis more general
- This abstraction is independent of cache implementation details.
The Goal of Our Project

Existing work

• Prove the absence of side channels in software.
• Identify the “upper-bound” of information leakage.
• Unable to provide counter examples to trigger vulnerabilities.
• May suffer from scalability issue.
• …

Find cache-based timing channels in production software

• Identify variants in cache line access traces.
• Provide counter examples (a pair of program secrets) to trigger such cache behavior variants.
• Scalable enough to analyze real world software systems.
Set-AssOCIATIVE Cache

The minimal storage unit of a cache is called a line and the cache is divided into sets consisting of the same number of lines.

- **Set index**: locate the set in which the data may be stored.
- **Tag**: confirm the data is present in one of its lines (cache hit vs. cache miss).
- **Line offset**: locate the data inside a cache line.

The upper \(N-L\) bits of a memory address maps a memory access to a cache line access.
Outline

- Background
- **Overview**
- Design
- Evaluation
A Motivating Example

```c
void foo(int secret)
{
    int table[128] = {0};
    int i, t;
    int index = 0;
    for (i=0; i<200; i++)
    {
        index = (index+secret) % 128;
        t = table[index];
        load t, 10 + 4 * index
    }
}
```

Program secret flow in the first loop iteration:

- `secret`

```
mov index, (0+secret)%128
load t, 10 + 4 * index
load t, 10 + 4 * index % 4
```

Secret-dependent memory access (where `secret` is active):

- Capture program secret dependence
- Construct symbolic formula

1. $F_1 = 10 + 4 \cdot secret \mod 128$
2. $F_2 = 10 + 4 \cdot (secret \mod 128) \mod 4$
Checking Cache Line Access Variants

\[ F(\vec{k}) \gg L \neq F(\vec{k}') \gg L \land C \]

\( F(\vec{k}) \): symbolic formula in which the only free variable is program secret \( \vec{k} \).

\( F(\vec{k}') \): substituting all occurrences of \( \vec{k} \) in \( F \) with a new variable \( \vec{k}' \).

\( L \): recall only the upper \( N-L \) bits are relevant to the cache behavior.

\( C \): conjunction of branch conditions; yielding a valid path to the memory access.

**Constraint:** can different secrets (here \( \vec{k} \) must not be equal to \( \vec{k}' \)) lead to different cache line access?

**Satisfiable solution:** cache line access variants would depend on program secrets.
Return to the Motivating Example

```c
void foo(int secret)
{
    int table[128] = {0};
    int i, t;
    int index = 0;
    for (i=0; i<200; i++)
    {
        index = (index+secret) % 128;
        t = table[index];
        t = table[index%4];
    }
}
```

**t = table[index]:**
- memory access formula
- cache line access formula
- satisfiability of \( F(k) \gg 6 \neq F(k') \gg 6 \]
- interpretation of the results: different \( k \) can access different cache lines

\[
F(k) \equiv 10+4 \times k \mod 128 \\
F(k) \gg 6 \equiv 10+4 \times k \mod 128 \gg 6 \\
[k=1, k'=30]
\]
Return to the Motivating Example

```c
void foo(int secret)
{
    int table[128] = {0};
    int i, t;
    int index = 0;
    for (i=0; i<200; i++)
    {
        index = (index+secret) % 128;
        t = table[index];
        t = table[index%4];
    }
}
```

On the current execution trace, this memory access is free from the threats we are interested.

- **t = table[index%4]:**
  - Memory access formula: \( F(k) \equiv 10 + 4 \cdot (k \mod 128) \mod 4 \)
  - Cache line access formula: \( F(k) \gg 6 \equiv 10 + 4 \cdot (k \mod 128) \mod 4 \gg 6 \)
  - Satisfiability of \( F(k) \gg 6 \neq F(k') \gg 6 \) \( \Rightarrow \) NO satisfiable solution
  - Interpretation of the results: \( \Rightarrow \) Cache line access is independent with k
Outline

• Background
• Overview
• Design
• Evaluation
Design Overview

- Online: execution trace-loging to get a single trace.
- Offline: execution trace analysis:
  - **Taint analysis** with the program secrets as taint seed.
  - Lift program secrets as free symbols, and perform **symbolic execution** on the **tainted instructions**.
  - For each collected pair of **memory addressing formula** and **path condition**, perform constraint solving.
  - The “**satisfiable**” solution for a memory access indicates a potential vulnerable program point.
Taint Analysis

Leverage taint analysis to capture instructions that are dependent on the program secrets

- **Taint propagation rules:**
  - **Registers:** straightforward tainting rules.
  - **CPU flags:** taint all CPU flags that can be affected in case any operand in an instruction is tainted.
  - **Memory:**
    - Taint memory if the stored content is tainted (*explicit information flow*).
    - Whenever the base registers or the offsets are tainted, the visited memory content is tainted (*implicit information flow*).
Symbolic Execution

Symbolic execution is used to construct memory addressing formulas and path conditions

• Symbolization of program secrets:
  • One free “key” symbol for register or memory that store the secrets.

• Symbolic engine:
  • Computation along the trace through symbolic values.
  • Implicit information flow through memory access: symbolize the visited memory content if the addressing formula contains key symbols.
Optimization: Identify Independent Vulnerabilities

In general, we create new “key” symbol for the load output.

- **No solution for** $F(\vec{k}) \neq F(\vec{k'}) \land C$
  - Memory access is independent of the secret.
  - Opt: no need to create new key symbol.

- **Has solution for** $F(\vec{k}) \gg L \neq F(\vec{k'}) \gg L \land C$
  - Independent point: further vulnerable points ”depend” on this one.
  - Timing channel could probably leak the same piece of information.
  - Most-likely attack surface but attack on further points are feasible.

- **Has solution for** $F(\vec{k}) \neq F(\vec{k'}) \land C$ but no solution for $F(\vec{k}) \gg L \neq F(\vec{k'}) \gg L \land C$
  - Different secrets would still access different memory cells.
  - Can construct information flow so we create new “key” symbol.
Outline

- Background
- Overview
- Design
- Evaluation
## Evaluation Cases

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Implementation</th>
<th>Versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA</td>
<td>Libgcrypt</td>
<td>1.6.1; 1.7.3</td>
</tr>
<tr>
<td></td>
<td>OpenSSL</td>
<td>0.9.7c; 1.0.2f</td>
</tr>
<tr>
<td></td>
<td>Botan</td>
<td>1.10.13</td>
</tr>
<tr>
<td>ElGamal</td>
<td>Libgcrypt</td>
<td>1.6.1; 1.7.3</td>
</tr>
<tr>
<td>AES</td>
<td>OpenSSL</td>
<td>0.9.7c; 1.0.2f</td>
</tr>
</tbody>
</table>

- Write sample code to invoke the decryption routine of each implementation.
- Default compilation and configuration settings when building each cryptosystem.
Evaluation Results Overview

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Implementation</th>
<th>Vulnerable Program Points (known/unknown)</th>
<th>Independent Vulnerable Points (known/unknown)</th>
<th># of Instructions on the Traces</th>
<th>Processing Time (CPU Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA</td>
<td>Libgcrypt 1.6.1</td>
<td>2/20</td>
<td>2/0</td>
<td>26,848,103</td>
<td>11542.3</td>
</tr>
<tr>
<td>RSA</td>
<td>Libgcrypt 1.7.3</td>
<td>0/0</td>
<td>NA</td>
<td>27,775,053</td>
<td>10788.9</td>
</tr>
<tr>
<td>ElGamal</td>
<td>Libgcrypt 1.6.1</td>
<td>2/19</td>
<td>2/0</td>
<td>31,077,760</td>
<td>17044.8</td>
</tr>
<tr>
<td>ElGamal</td>
<td>Libgcrypt 1.7.3</td>
<td>0/0</td>
<td>NA</td>
<td>31,407,882</td>
<td>12463.1</td>
</tr>
<tr>
<td>RSA</td>
<td>OpenSSL 0.9.7c</td>
<td>0/2</td>
<td>0/1</td>
<td>674,797</td>
<td>199.3</td>
</tr>
<tr>
<td>RSA</td>
<td>OpenSSL 1.0.2f</td>
<td>0/2</td>
<td>0/1</td>
<td>473,392</td>
<td>165.6</td>
</tr>
<tr>
<td>AES</td>
<td>OpenSSL 0.9.7c</td>
<td>48/0</td>
<td>48/0</td>
<td>791</td>
<td>43.4</td>
</tr>
<tr>
<td>AES</td>
<td>OpenSSL 1.0.2f</td>
<td>32/0</td>
<td>32/0</td>
<td>2,410</td>
<td>48.5</td>
</tr>
<tr>
<td>RSA</td>
<td>Botan 1.10.13</td>
<td>0/29</td>
<td>0/2</td>
<td>2,005,124</td>
<td>7527.0</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>84/72</td>
<td>84/4</td>
<td>120,265,312</td>
<td>59822.9</td>
</tr>
</tbody>
</table>

- We identified **known vulnerabilities**, and discovered **new program points** that can potentially lead to cache access variants.
- **Known/unknown independent vulnerable points** are also identified, which indicates most-likely attack surface.
- **Scalability**: CacheD processes over **120 million instructions** within **17 CPU hours**.
Exploring Independent Vulnerabilities

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Implementation</th>
<th>Observe the Access of Different Cache Lines</th>
<th>Observe Different Cache Status (hit vs. miss)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA</td>
<td>Libgcrypt 1.6.1</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ElGamal</td>
<td>Libgcrypt 1.6.1</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RSA</td>
<td>OpenSSL 0.9.7c</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RSA</td>
<td>OpenSSL 1.0.2f</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>AES</td>
<td>OpenSSL 0.9.7c</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>AES</td>
<td>OpenSSL 1.0.2f</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>RSA</td>
<td>Botan 1.10.13</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

- Instrument the source code and modify secrets with $(k, k')$.
- Monitor the execution of instrumented code and intercept cache access from CPU to L1 Data Cache using a hardware simulator (gem5).
- **Memory accesses** at all vulnerable points visit different cache lines.
- **Cache status** are different at the vulnerable program points for most of the cases.

Conservatively check the cache status **only at the vulnerable program points!**
Case Study of RSA Vulnerabilities

- Case study of two RSA vulnerable program points in Libgcrypt (v1.6.1).
- Program secrets are marked as red.
- Vulnerable program points in the source and traces are bold.
- For each vulnerable program point, the constraint solver gives a pair of counter examples (k, k’).
- Observation in the hardware simulator shows cache behavior variants.
Unknown Vulnerabilities in OpenSSL RSA Implementation

```c
int BN_num_bits(const BIGNUM *a) {
    BN_ULONG l;
    int i;

    bn_check_top(a);

    if (a->top == 0) return(0);
    l=a->d[a->top-1];
    assert(l != 0);
    i=(a->top-1)*BN_BITS2;
    return(i+BN_num_bits_word(l));
}

int BN_num_bits_word(BN_ULONG l) {
    static const char bits[256]= {
    0,1,2,2,3,3,3,3,3,3,4,4,4,4,4,4,4,4,
    5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
    ...
    8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,
    8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,
    ...
    return bits[l];
}
```
Summary

Identify cache-based timing channels in production software

- A trace-based analysis method that models the cache line access variants.
- A conceptually simple yet general enough model to capture most adopted threats.
- Perform precise and scalable analysis to identify known vulnerabilities as well as discover previously unknown issue.
Thank You

Q & A

Shuai Wang
http://www.personal.psu.edu/szw175/