CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation

Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, Mahmut Kandemir
Pennsylvania State University
Cache Side Channels

- Side Channel
  - Unintentional information transfer
Cache Side Channels

- Side Channel
  - Unintentional information transfer
How Severe is the Problem?

- High bandwidth attack
- Work on secure enclaves
- Can be launched across VM’s in the cloud

Finding vulnerabilities in code is challenging!

Security

Intel shrugs off ‘new’ side-channel attacks on branch prediction units and SGX

Researchers show how side-channel attacks can be used to steal encryption keys on ARM processors

Spectre and Meltdown: Powerful Reminders of Side Channel Attacks
Prior Work

• **CacheAudit** (Doychev et al. Security ‘13)
  • Uses abstract interpretation
  • Computes upper bound on leakage
  • Does **not** provide location of leakage

• **CacheD** (Wang et al. Security ‘17)
  • Uses symbolic execution
  • Can detect where leakage happens
  • May **miss** side channels (not sound)
    • Requires concrete inputs
    • Does not provide fixes
Introducing CaSym

- Uses **cache-aware** symbolic execution
- **Soundly** models cache side channels
  - Memory accesses
  - Branches
- Detects **cause** of side channel
  - Provides simple **fix** mechanisms
- **Flexible** cache models
  - Infinite
  - Age
  - LRU
CaSym: Overview

Source Code → Clang → LLVM IR Code → Cache Model:
- Infinite Age
- LRU

Attack Model:
- Access Trace

Z3 → Cache Formula → Cache Analysis

Apply Mitigations

Localization Report
Example: Square & Multiply

- Does modular exponentiation
- Used in asymmetric encryption
  - RSA, ElGamal, etc

1: result = 0;
2: for(int i = expLen-1; i > 0; i--)
3: {
4:   result = result * result;
5:   result = result % mod;
6:   if((1 << i) & exp)
7:     {
8:       result = base * result;
9:       result = result % mod;
10:   }
11: }

Localization Report
Problem: Key Dependent Branch
Detected at: Line 6
Witnesses: ...

Iterates over each bit of key

Key

Causes different observable cache states
Symbolic Execution

- Program variables
  - Treats all program variables symbolically

- Cache variables
  - Creates cache variable for each program variable
  - Cache variables values are determined by cache model

Toy Program

```c
int a, b;
int PRIV key;

if (key == 1)
{
    a = 0;
}
else
{
    b = 0;
}
```

Cache Variables

- $a_{cache}$
- $b_{cache}$
- $key_{cache}$
Verification

- Run program twice
- Cache and public variables are same between runs
- Sensitive variables must be different
- Vulnerability reported when two different cache states are achieved

Toy Program

```c
int a, b;
int PRIV key;

if (key == 1)
{
    a = 0;
}
else
{
    b = 0;
}
```

Cache Variables

- `a_cache`
- `b_cache`
- `key_cache`

Cache Variables

- `a'_cache`
- `b'_cache`
- `key'_cache`

Toy Program

```c
int a', b';
int PRIV key';

if (key' == 1)
{
    a' = 0;
}
else
{
    b' = 0;
}
```
## Cache Models

### Motivation
- Cache implementations are complex
  - Replacement policies, hierarchies, inclusivity, etc.
  - Vary amongst processors

<table>
<thead>
<tr>
<th>Infinite</th>
<th>Age</th>
<th>LRU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Treats cache as an infinite set</td>
<td>Assigns an age to all variables</td>
<td>Also assigns ages to all variables</td>
</tr>
<tr>
<td>Never evicts data from cache</td>
<td>Overapproximates real replacement policies</td>
<td>Youngest $n$ variables are cached</td>
</tr>
</tbody>
</table>
Infinite Model Demo

Toy Program

```c
int a, b;
int PRIV key;

if (key == 1)
{
    a = 0;
}
else
{
    b = 0;
}
```

Abstract Cache

- `key → 1`
  - `Used(key) → true`
  - `Used(a) → true`
  - `Used(b) → false`

Abstract Cache

- `key := 0`
  - `Used(key) → false`
  - `Used(a) → false`
  - `Used(b) → false`
Toy Program

```
int a, b;
int PRIV key;

if (key == 1)
  { a = 0;

  }
else
  { b = 0;
  }
```

Abstract Cache

| Used(key) → 1 |
|______________|
| Used(a) → 0   |
| Used(b) → ∞  |

Used(key) → 1

Cache

| Used(key) → ∞ |
|______________|
| Used(a) → ∞  |
| Used(b) → ∞  |

≠
• Array reads are **unconstrained**
  • Uses taint analysis to check if read is sensitive

• **Reset** constraints
  • Breaks program into smaller chunks
  • Recomputes sensitive variables
  • Useful for loops

• Loop **transformation**
  • Soundly rewrite program to be loop free
  • Makes loop unrolling unnecessary
Attack Models

Access Model

Program

access(a);
access(b);
access(c);
access(d);
access(e);
access(f);

Set of Addresses:
{&a, &b, &c, &d, &e, &f}
Trace Model

Program

access(a);
access(b);
access(c);
access(d);
access(e);
access(f);

Sequence of Addresses:
[&a, &b, &c, &d, &e, &f]
## Crypto Results: Trace

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Infinite</th>
<th></th>
<th>Age</th>
<th></th>
<th>LRU (2k)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Found</td>
<td>Time</td>
<td>Found</td>
<td>Time</td>
<td>Found</td>
<td>Time</td>
</tr>
<tr>
<td>AES libgcrypt</td>
<td>64</td>
<td>8.9</td>
<td>64</td>
<td>16.7</td>
<td>64</td>
<td>635</td>
</tr>
<tr>
<td>AES mbed TLS</td>
<td>17</td>
<td>5.9</td>
<td>17</td>
<td>17.0</td>
<td>17</td>
<td>757</td>
</tr>
<tr>
<td>3DES libgcrypt</td>
<td>128</td>
<td>62.5</td>
<td>48</td>
<td>27.0</td>
<td>48</td>
<td>803</td>
</tr>
<tr>
<td>3DES mbed TLS</td>
<td>2</td>
<td>0.92</td>
<td>2</td>
<td>2.65</td>
<td>2</td>
<td>9.2</td>
</tr>
<tr>
<td>DES glibc</td>
<td>0</td>
<td>0.24</td>
<td>0</td>
<td>1.27</td>
<td>0</td>
<td>5.35</td>
</tr>
<tr>
<td>UFC glibc</td>
<td>0</td>
<td>0.24</td>
<td>0</td>
<td>1.27</td>
<td>0</td>
<td>5.35</td>
</tr>
<tr>
<td>Square &amp; Multiply libgcrypt</td>
<td>3</td>
<td>84.8</td>
<td>3</td>
<td>2618</td>
<td>3</td>
<td>275</td>
</tr>
<tr>
<td>Square &amp; Always Multiply libgcrypt</td>
<td>4</td>
<td>18.9</td>
<td>17</td>
<td>73.2</td>
<td>17</td>
<td>757</td>
</tr>
<tr>
<td>Left-to-Right Modular Exp</td>
<td>3</td>
<td>84.8</td>
<td>3</td>
<td>2618</td>
<td>3</td>
<td>275</td>
</tr>
<tr>
<td></td>
<td>Totals</td>
<td></td>
<td></td>
<td></td>
<td>Totals</td>
<td></td>
</tr>
<tr>
<td></td>
<td>269</td>
<td>217.36</td>
<td>270</td>
<td>3226.82</td>
<td>269</td>
<td>8881.85</td>
</tr>
</tbody>
</table>

*Order of accesses is still different*

*Can take significantly more time*

*Most realistic model*

*Finds one additional vulnerable location*
## Protected Results

### Preloading vs. Pinning

<table>
<thead>
<tr>
<th>Functions</th>
<th>Preloading</th>
<th></th>
<th>Pinning</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Infinite</td>
<td>Age</td>
<td>Infinite</td>
</tr>
<tr>
<td>AES libgcrypt</td>
<td>0</td>
<td>2.95</td>
<td>64</td>
<td>17.4</td>
</tr>
<tr>
<td>AES mbed TLS</td>
<td>0</td>
<td>1.68</td>
<td>17</td>
<td>17.4</td>
</tr>
<tr>
<td>3DES libgcrypt</td>
<td>0</td>
<td>84.0</td>
<td>128</td>
<td>170</td>
</tr>
<tr>
<td>3DES mbed TLS</td>
<td>0</td>
<td>1.53</td>
<td>48</td>
<td>65.5</td>
</tr>
<tr>
<td>DES glibc</td>
<td>0</td>
<td>0.56</td>
<td>2</td>
<td>3.15</td>
</tr>
<tr>
<td><strong>Totals</strong></td>
<td>0</td>
<td>90.72</td>
<td>259</td>
<td>273.45</td>
</tr>
</tbody>
</table>

- **Preloading**: Data cached at beginning of function
- **Pinning**: Data cached throughout function
Conclusions

• Built CaSym to **automatically** identify vulnerabilities in programs
• CaSym supports a **variety** of cache models
  • Easy to get different **precision** and **efficiency**
• Tested on an **assortment** of benchmarks
  • Confirm many existing **vulnerabilities** in crypto benchmarks
  • Verified **mitigations** strategies on crypto benchmarks
  • Found over **20 new** potential vulnerabilities in the PostgreSQL database
Thank You!