

CacheD: Identifying Cache-Based Timing Channels in Production Software

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

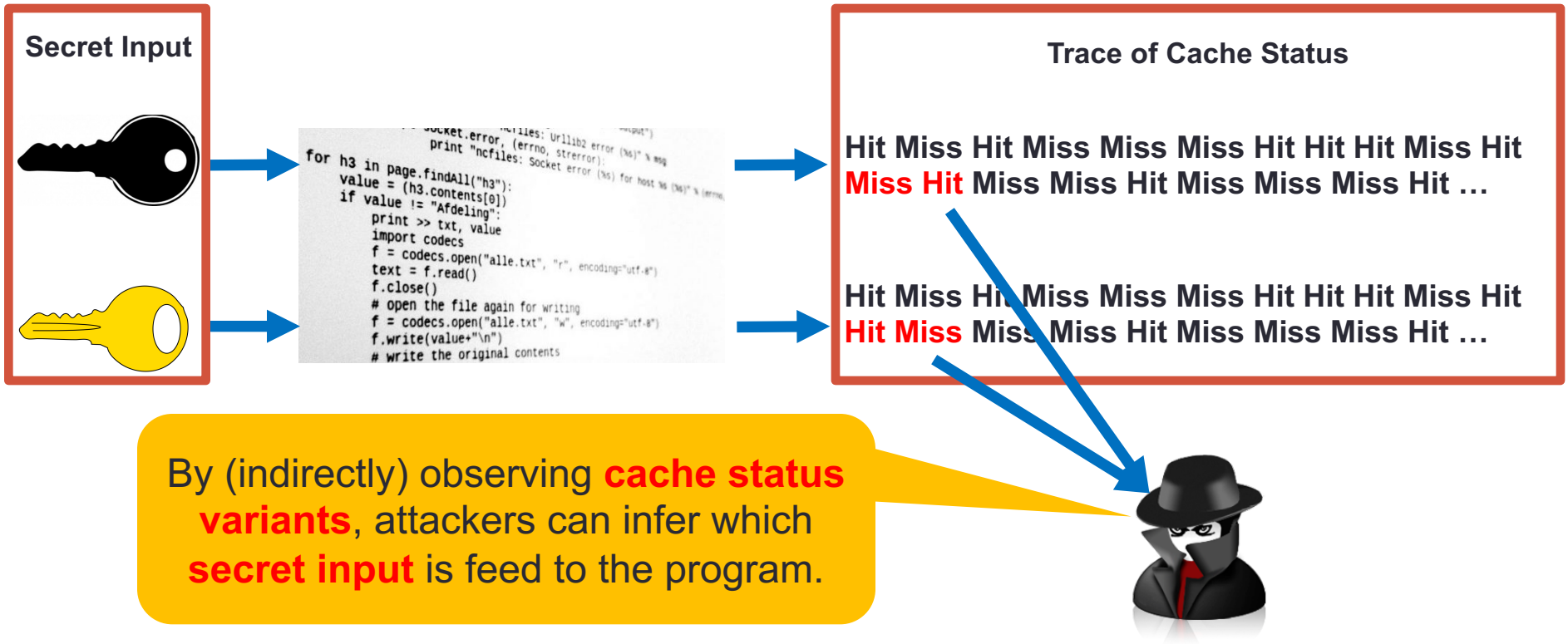
The Pennsylvania State University

USENIX Security 2017

Outline

- **Background**
- Overview
- Design
- Evaluation

Cache-based Timing Channel Attack



Some pictures are from http://www.securecontenttechnologies.com/paloalto.html and http://clipartix.com/key-clipart/.

Cache-based Timing Channel Attack

Leakage due to secret-dependent control flow

```
r = 1;
for i from n−1 to 0 {
    r = r*r mod m;
    if (e[i] == 1) {
        r = r*b mod m;
    }
}
```

n-bit secret exponent

Prime+Probe corresponding cache line

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

Leakage due to secret-dependent memory access

```
idx = 0;
for i from 0 to n-1 {
    idx = window_ith(key, i);
    t = table[idx];
}
```

window size of key

Prime+Probe corresponding cache line

- *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

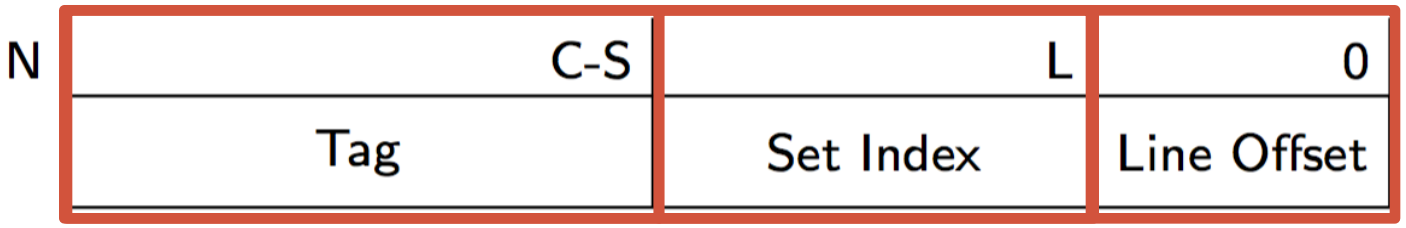
Existing work

- 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

```
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];
  }
}
```

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 (whe
➔ **Capture program secret dependence**
➔ **Construct symbolic formula**

F_1 =10+4·*secret* mod 128
F_2 =10+4·(*secret* mod 128) mod 4

Checking Cache Line Access Variants

$$F(\vec{k}) \gg L \neq F(\overrightarrow{k'}) \gg L \wedge C$$

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

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

***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 $\overrightarrow{k'}$) lead to different cache line access?

Satisfiable solution: cache line access variants would depend on program secrets.

Return to the Motivating Example

```
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];
  }
}
```

L equals 6 is a standard configuration.

ache line be able to ... secrets.

t = table[index]:

memory access formula ➔ $F(k) \equiv 10+4 \cdot k \bmod 128$

cache line access formula ➔ $F(k) >> 6 \equiv 10+4 \cdot k \bmod 128 >>$

satisfiability of $F(\vec{k}) \gg 6 \neq F(\vec{k'}) \gg 6$ ➔ **[$\vec{k}$=1, $\vec{k'}$ =30]**

interpretation of the results: ➔ **different k can access different cache lines**

Return to the Motivating Example

```
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 ➔ $F(k) \equiv 10+4\cdot(k \bmod 128) \bmod$

cache line access formula ➔ $F(k) >> 6 \equiv 10+4\cdot(k \bmod 128) \; n$ $4 >> 6$

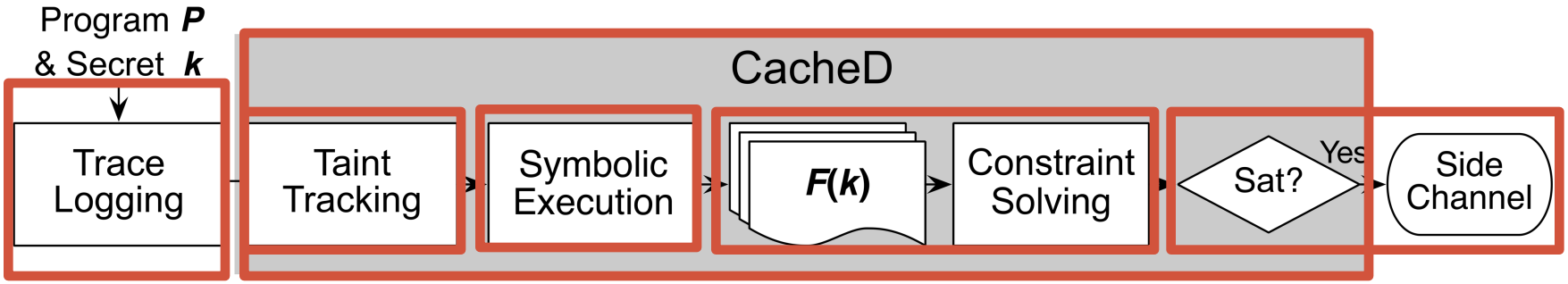
satisfiability of $F(\vec{k}) \gg 6 \neq F(\vec{k'}) \gg 6$ ➔ **NO satisfiable solution**

interpretation of the results: ➔ **cache line access is independent with k**

Outline

- Background
- Overview
- **Design**
- Evaluation

Design Overview



Programmers can provide inputs to cover security-sensitive code components.

- Online: execution trace-logging 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.

load reg, [F(k)]

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

Algorithm	Implementation	Versions
RSA	Libgcrypt OpenSSL Botan	1.6.1; 1.7.3 0.9.7c; 1.0.2f 1.10.13
ElGamal	Libgcrypt	1.6.1; 1.7.3
AES	OpenSSL	0.9.7c; 1.0.2f

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

Evaluation Results Overview

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

- 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

Algorithm	Implementation	Observe the Access of Different Cache Lines	Observe Different Cache Status (hit vs. miss)
RSA	Libgcrypt 1.6.1	✓	✓
ElGamal	Libgcrypt 1.6.1	✓	✓
RSA	OpenSSL 0.9.7c	✓	✓
RSA	OpenSSL 1.0.2f	✓	×
AES	OpenSSL 0.9.7c	✓	✓
AES	OpenSSL 1.0.2f	✓	×
RSA	Botan 1.10.13	✓	✓

*Conservatively check the cache status **only** at the vulnerable program points!*

- 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.

Case Study of RSA Vulnerabilities

```
1  void _gcry_mpi_powm (gcry_mpi_t res,
2    gcry_mpi_t base,
3    gcry_mpi_t expo, gcry_mpi_t mod) {
4    ...
5    e = ep[i];
6    count_leading_zeros (c, e);
7    e = (e << c) << 1;
8    ...
9    e0 = (e >> (BITS_PER_MPI_LIMB - W));
10   count_trailing_zeros (c0, e0);
11   e0 = (e0 >> c0) >> 1;
12   ...
13   base_u = b_2i3[e0 - 1];
14   base_u_size = b_2i3size[e0 - 1];
15   ...
16 }
```

(a) Source code.

```
1  ...
2  mov edx, [esp+0x38]
3  sub edx, 0x1
4  mov ecx, [esp+edx*4+0xb8]
5  mov edx, [esp+edx*4+0xf4]
6  mov esi, [esp+0x24]
7  ...
```

```
1  Solution of the first mem access (line 4):
2  e = 0x10000016
3  e'= 0x1000003e
4
5  Solution of the second mem access (line 5):
6  e = 0x400500d
7  e'= 0x1007c
```

```
1  0x44156a: 0x7fffffffe440 hit
2  0x441572: 0x7fffffffe3c8 hit
3  0x44156a: 0x7fffffffe438 hit
4  0x441572: 0x7fffffffe3c4 hit
5  0x44156a: 0x7fffffffe3d8 hit
6  0x441572: 0x7fffffffe394 hit
7  0x44156a: 0x7fffffffe408 hit
8  0x441572: 0x7fffffffe3ac hit
```

```
1  0x44156a: 0x7fffffffe3f0 hit
2  0x441572: 0x7fffffffe3a0 hit
3  0x44156a: 0x7fffffffe438 hit
4  0x441572: 0x7fffffffe3c4 hit
5  0x44156a: 0x7fffffffe3d8 miss
6  0x441572: 0x7fffffffe394 miss
7  0x44156a: 0x7fffffffe408 miss
8  0x441572: 0x7fffffffe3ac hit
```

(b) Instructions on the dumped trace and solutions provided by the solver.

- 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

```
1   int BN_num_bits(const BIGNUM *a) {
2     BN_ULONG l;
3     int i;
4
5     bn_check_top(a);
6
7     if (a->top == 0) return(0);
8     l=a->d[a->top-1];
9     assert(l != 0);
10    i=(a->top-1)*BN_BITS2;
11    return(i+BN_num_bits_word(l));
12  }
13
14  int BN_num_bits_word(BN_ULONG l) {
15    static const char bits[256]={
16      0,1,2,2,3,3,3,3,4,4,4,4,4,4,4,4,
17      5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
18      ...
19      8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,
20      8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,
21    };
22    ....
23    return bits[l];
24  }
```

Calculate the length of the secret key by bit.

A lookup table to determine the exact number of bits in the last entry of a->d.

Previously unknown vulnerable program point.

We have also identified unknown vulnerabilities in Botan (1.10.13) RSA decryption procedures.

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/