C: CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

S: Stanford University

# Microarchitectural Leakage Templates and Their Application to Cache-Based Side Channels

CCS 2022

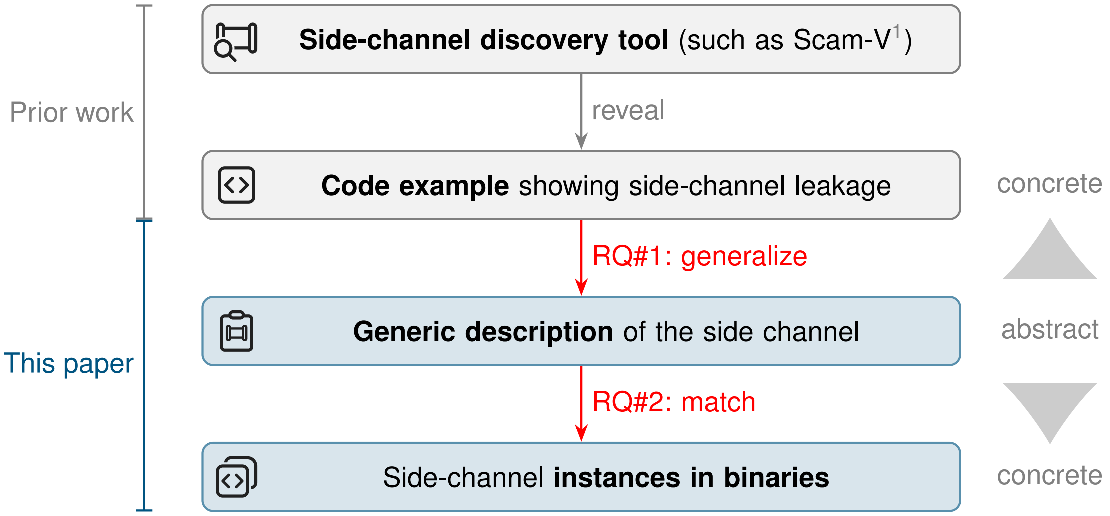
Ahmad Ibrahim[C] Hamed Nemati[S,C] Till Schlüter[C] Nils Ole Tippenhauer[C] Christian Rossow[C]

November 10, 2022

## Previously...

**Security of proprietary CPUs**

**Discovering new side channels**


Prior work
This paper
**Side-channel discovery tool** (such as Scam-V[1])
reveal
**Code example** showing side-channel leakage
RQ#1: generalize
**Generic description** of the side channel
RQ#2: match
Side-channel **instances in binaries**
concrete
abstract
concrete

[1] Nemati et al., "Validation of Abstract Side-Channel Models for Computer Architectures".

## Generic Description of a Side Channel

$\mathcal{P}(A)$: A **code** template

$\mathcal{B}$: Distinct **behaviors**

- e.g. timing: $\mathcal{B} = \{\bullet \text{ fast}, \circ \text{ slow}\}$

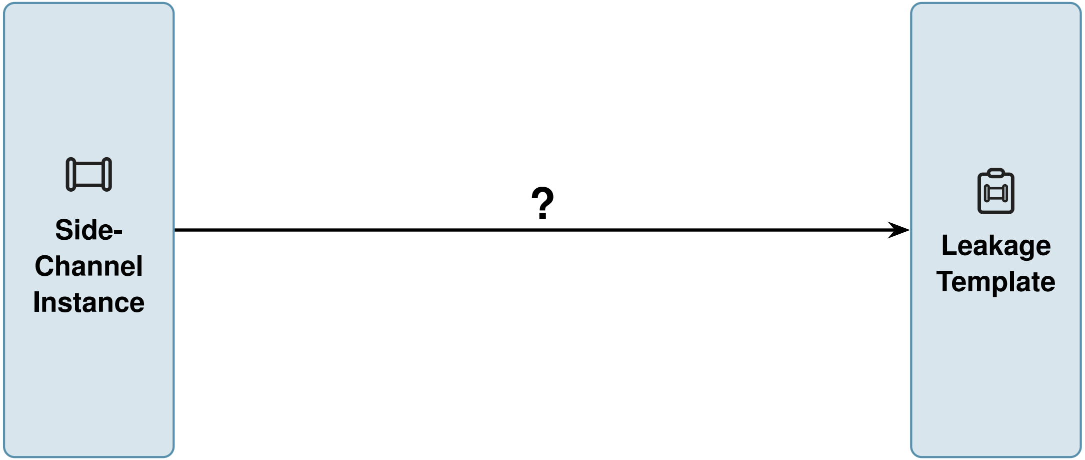
$\mathcal{R}(A, b)$: **Relations** between inputs, leading to a certain behavior

- *"When inputs X and Y are in relation, then behavior ●"*

**Leakage Template**

| Code $\mathcal{P}(A)$ | Behavior and Relations | |
|---|---|---|
| `ldr x0, [x1]` | $\mathcal{B}$ | $\mathcal{R}(A, b)$ |
| `; ...` | $(\bullet)$ fast | $\texttt{sameTag}(x_1, x_2) \land \texttt{sameSet}(x_1, x_2)$ |
| `ldr x0, [x2]` | $(\circ)$ slow | $\neg\texttt{sameTag}(x_1, x_2) \lor \neg\texttt{sameSet}(x_1, x_2)$ |

Figure: Leakage Template: Cache-Timing Side Channel


Side-Channel Instance
?
Leakage Template

| Code $\mathcal{P}(A)$ | Behavior and Relations | |
|---|---|---|
| `ldr x0, [x1]` | $\mathcal{B}$ | $\mathcal{R}(A,b)$ |
| `; ...` | (●) fast | sameTag$(x_1,x_2)$ $\wedge$ sameSet$(x_1,x_2)$ |
| `ldr x0, [x2]` | (○) slow | $\neg$sameTag$(x_1,x_2)$ $\vee$ $\neg$sameSet$(x_1,x_2)$ |

Figure: Leakage Template: Cache-Timing Side Channel

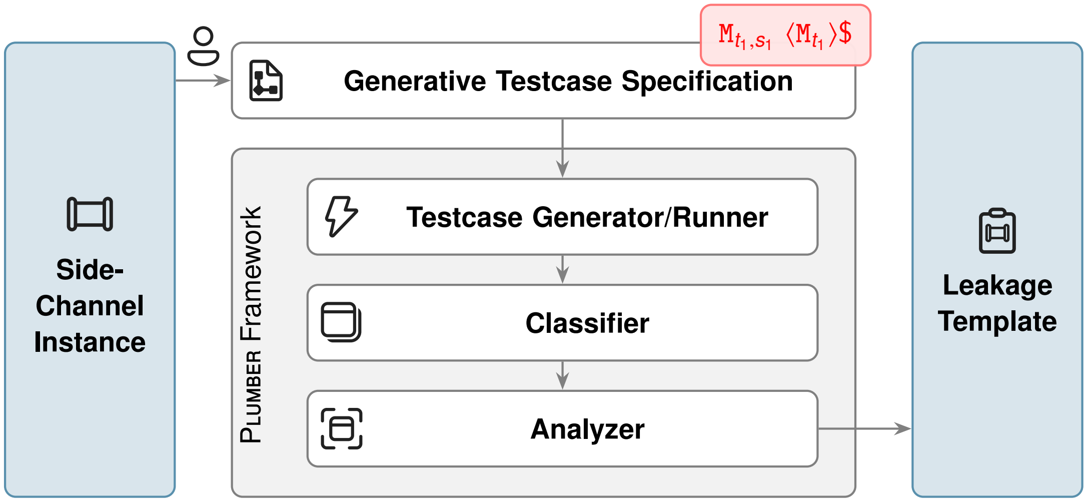
**Memory address:** $\cdots\underbrace{11101010010}_{\text{tag}}\,\underbrace{0010010}_{\text{set index}}\,010010$

## Testcases

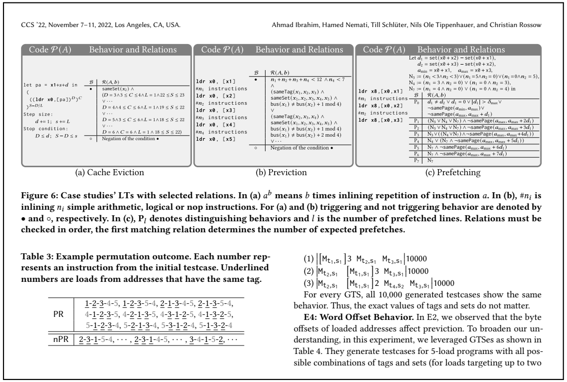
| | | |
|---|---|---|
| **TC0** | $t_1, s_1$ | $t_1, s_0$ |
| **TC1** | $t_1, s_1$ | $t_1, s_1$ |
| ... | ... | ... |
| **TC127** | $t_1, s_1$ | $t_1, s_{127}$ |
| | `x1`: fixed tag and set | `x2`: fixed tag, iterate over all sets |

## Classification

| (●) fast | | (○) slow | |
|---|---|---|---|
| **x1** | **x2** | **x1** | **x2** |
| $t_1, s_1$ | $t_1, s_1$ | $t_1, s_1$ | $t_1, s_0$ |
| | | $t_1, s_1$ | $t_1, s_2$ |
| | | ... | ... |
| | | $t_1, s_1$ | $t_1, s_{127}$ |


$M_{t_1,s_1}$ $\langle M_{t_1} \rangle\$$
Generative Testcase Specification
Side-Channel Instance
Plumber Framework
Testcase Generator/Runner
Classifier
Analyzer
Leakage Template

## Case Studies



| Code $\mathcal{P}(A)$ | Behavior and Relations |
|---|---|

```
let pa = x1+s+d in
 (
  ((ldr x0,[pa])^D)^C
 )^(S−D/L)
Step size:
    d += 1;  s += L
Stop condition:
    D ≤ d;  S − D ≤ s
```

| $\mathcal{B}$ | $\mathcal{R}(A, b)$ |
|---|---|
| • | sameSet($x_i$) $\wedge$ ($D = 3 \wedge 3 \le C \le 4 \wedge L = 1 \wedge 22 \le S \le 23$ $\vee \cdots$ $D = 4 \wedge 4 \le C \le 6 \wedge L = 1 \wedge 19 \le S \le 22$ $\vee \cdots$ $D = 5 \wedge 3 \le C \le 6 \wedge L = 1 \wedge 18 \le S \le 22$ $\vee \cdots$ $D = 6 \wedge C = 6 \wedge L = 1 \wedge 18 \le S \le 22$) |
| ◦ | Negation of the condition • |

(a) Cache Eviction

| Code $\mathcal{P}(A)$ | Behavior and Relations |
|---|---|

```
ldr x0, [x1]
#n1 instructions
ldr x0, [x2]
#n2 instructions
ldr x0, [x3]
#n3 instructions
ldr x0, [x4]
#n4 instructions
ldr x0, [x5]
```

| $\mathcal{B}$ | $\mathcal{R}(A, b)$ |
|---|---|
| • | $n_1 + n_2 + n_3 + n_4 < 12 \wedge n_4 < 7$ $\wedge$ (sameTag($x_1, x_2, x_3$) $\wedge$ sameSet($x_1, x_2, x_3, x_4, x_5$) $\wedge$ bus($x_1$) $\neq$ bus($x_2$) + 1 mod 4) $\vee$ (sameTag($x_2, x_3, x_4$) $\wedge$ sameSet($x_1, x_2, x_3, x_4, x_5$) $\wedge$ bus($x_1$) $\neq$ bus($x_2$) + 1 mod 4) bus($x_1$) $\neq$ bus($x_2$) + 2 mod 4) $\vee \cdots$ |
| ◦ | Negation of the condition • |

(b) Prevision

| Code $\mathcal{P}(A)$ | Behavior and Relations |
|---|---|

```
ldr x8,[x0,x1]
#n1 instructions
ldr x8,[x0,x2]
#n2 instructions
ldr x8,[x0,x3]
```

Let $d_1$ = set(x0 + x2) − set(x0 + x1),
$d_2$ = set(x0 + x3) − set(x0 + x2),
$a_{\min}$ = x0 + x1, $a_{\max}$ = x0 + x3,
$N_3 := (n_1 < 3 \wedge n_2 < 3) \vee (n_1 = 5 \wedge n_2 = 0) \vee (n_1 = 0 \wedge n_2 = 5)$,
$N_4 := (n_1 = 3 \wedge n_2 = 0) \vee (n_1 = 0 \wedge n_2 = 3)$,
$N_7 := (n_1 = 4 \wedge n_2 = 0) \vee (n_1 = 0 \wedge n_2 = 4)$ in

| $\mathcal{B}$ | $\mathcal{R}(A, b)$ |
|---|---|
| $P_0$ | $d_1 \neq d_2 \vee d_1 = 0 \vee \lvert d_1 \rvert > \delta_{\max} \vee$ $\neg$samePage($a_{\min}, a_{\max}$) $\vee$ $\neg$samePage($a_{\max}, a_{\max} + d_1$) |
| $P_1$ | $(N_3 \vee N_4 \vee N_7) \wedge \neg$samePage($a_{\max}, a_{\max} + 2d_1$) |
| $P_2$ | $(N_3 \vee N_4 \vee N_7) \wedge \neg$samePage($a_{\max}, a_{\max} + 3d_1$) |
| $P_3$ | $N_3 \vee ((N_4 \vee N_7) \wedge \neg$samePage($a_{\max}, a_{\max} + 4d_1$)) |
| $P_4$ | $N_4 \vee (N_7 \wedge \neg$samePage($a_{\max}, a_{\max} + 5d_1$)) |
| $P_5$ | $N_7 \wedge \neg$samePage($a_{\max}, a_{\max} + 6d_1$) |
| $P_6$ | $N_7 \wedge \neg$samePage($a_{\max}, a_{\max} + 7d_1$) |
| $P_7$ | $N_7$ |

(c) Prefetching

**Figure 6: Case studies' LTs with selected relations. In (a) $a^b$ means $b$ times inlining repetition of instruction $a$. In (b), $\#n_i$ is inlining $n_i$ simple arithmetic, logical or nop instructions. For (a) and (b) triggering and not triggering behavior are denoted by • and ◦, respectively. In (c), $P_l$ denotes distinguishing behaviors and $l$ is the number of prefetched lines. Relations must be checked in order, the first matching relation determines the number of expected prefetches.**

**Table 3: Example permutation outcome. Each number represents an instruction from the initial testcase. Underlined numbers are loads from addresses that have the same tag.**

| | |
|---|---|
| PR | $\underline{1}$-$\underline{2}$-$\underline{3}$-4-5, $\underline{1}$-$\underline{2}$-$\underline{3}$-5-4, $\underline{2}$-$\underline{1}$-$\underline{3}$-4-5, $\underline{2}$-$\underline{1}$-$\underline{3}$-5-4, 4-$\underline{1}$-$\underline{2}$-$\underline{3}$-5, 4-$\underline{2}$-$\underline{1}$-$\underline{3}$-5, 4-$\underline{3}$-$\underline{1}$-$\underline{2}$-5, 4-$\underline{1}$-$\underline{3}$-$\underline{2}$-5, 5-$\underline{1}$-$\underline{2}$-$\underline{3}$-4, 5-$\underline{2}$-$\underline{1}$-$\underline{3}$-4, 5-$\underline{3}$-$\underline{1}$-$\underline{2}$-4, 5-$\underline{1}$-$\underline{3}$-$\underline{2}$-4 |
| nPR | $\underline{2}$-$\underline{3}$-$\underline{1}$-5-4, ··· , $\underline{2}$-$\underline{3}$-$\underline{1}$-4-5, ··· , $\underline{3}$-4-$\underline{1}$-5-$\underline{2}$, ··· |

(1) $\left|[\mathsf{M}_{t_1,s_1}]3\ \ \mathsf{M}_{t_2,s_1}\ \ \mathsf{M}_{t_3,s_1}\right|10000$
(2) $\left|\mathsf{M}_{t_2,s_1}\ \ [\mathsf{M}_{t_1,s_1}]3\ \ \mathsf{M}_{t_3,s_1}\right|10000$
(3) $\left|\mathsf{M}_{t_2,s_1}\ \ [\mathsf{M}_{t_1,s_1}]2\ \ \mathsf{M}_{t_4,s_2}\ \ \mathsf{M}_{t_3,s_1}\right|10000$

For every GTS, all 10,000 generated testcases show the same behavior. Thus, the exact values of tags and sets do not matter.

**E4: Word Offset Behavior.** In E2, we observed that the byte offsets of loaded addresses affect prevision. To broaden our understanding, in this experiment, we leveraged GTSes as shown in Table 4. They generate testcases for 5-load programs with all possible combinations of tags and sets (for loads targeting up to two

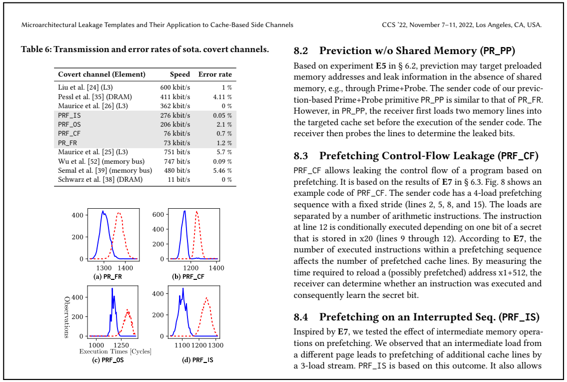
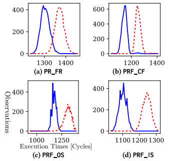


**Table 6: Transmission and error rates of sota. covert channels.**

| Covert channel (Element) | Speed | Error rate |
|---|---|---|
| Liu et al. [24] (L3) | 600 kbit/s | 1 % |
| Pessl et al. [35] (DRAM) | 411 kbit/s | 4.11 % |
| Maurice et al. [26] (L3) | 362 kbit/s | 0 % |
| PRF_IS | 276 kbit/s | 0.05 % |
| PRF_OS | 206 kbit/s | 2.1 % |
| PRF_CF | 76 kbit/s | 0.7 % |
| PR_FR | 73 kbit/s | 1.2 % |
| Maurice et al. [25] (L3) | 751 bit/s | 5.7 % |
| Wu et al. [52] (memory bus) | 747 bit/s | 0.09 % |
| Semal et al. [39] (memory bus) | 480 bit/s | 5.46 % |
| Schwarz et al. [38] (DRAM) | 11 bit/s | 0 % |



### 8.2 Prevision w/o Shared Memory (PR_PP)

Based on experiment **E5** in § 6.2, prevision may target preloaded memory addresses and leak information in the absence of shared memory, e.g., through Prime+Probe. The sender code of our prevision-based Prime+Probe primitive PR_PP is similar to that of PR_FR. However, in PR_PP, the receiver first loads two memory lines into the targeted cache set before the execution of the sender code. The receiver then probes the lines to determine the leaked bits.

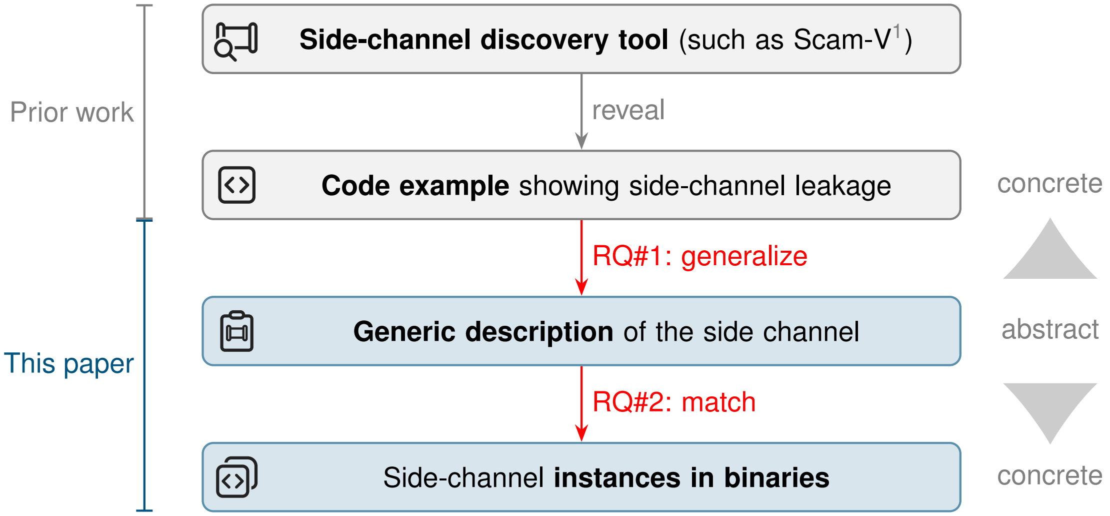
### 8.3 Prefetching Control-Flow Leakage (PRF_CF)

PRF_CF allows leaking the control flow of a program based on prefetching. It is based on the results of **E7** in § 6.3. Fig. 8 shows an example code of PRF_CF. The sender code has a 4-load prefetching sequence with a fixed stride (lines 2, 5, 8, and 15). The loads are separated by a number of arithmetic instructions. The instruction at line 12 is conditionally executed depending on one bit of a secret that is stored in x20 (lines 9 through 12). According to **E7**, the number of executed instructions within a prefetching sequence affects the number of prefetched cache lines. By measuring the time required to reload a (possibly prefetched) address x1+512, the receiver can determine whether an instruction was executed and consequently learn the secret bit.

### 8.4 Prefetching on an Interrupted Seq. (PRF_IS)

Inspired by **E7**, we tested the effect of intermediate memory operations on prefetching. We observed that an intermediate load from a different page leads to prefetching of additional cache lines by a 3-load stream. PRF_IS is based on this outcome. It also allows

**In the paper:** 3 Leakage Templates, 4 Covert Channels


Prior work
This paper
Side-channel discovery tool (such as Scam-V[1])
reveal
Code example showing side-channel leakage
RQ#1: generalize
Generic description of the side channel
RQ#2: match
Side-channel instances in binaries
concrete
abstract
concrete

[1] Nemati et al., "Validation of Abstract Side-Channel Models for Computer Architectures".

## Searching for Instances of a Leakage Template

1. **Static Analysis**

   Search for candidate code sections matching $\mathcal{P}(A)$

2. **Dynamic Analysis**

   For each candidate section:
   Check whether different inputs fulfill relations for different behaviors
   (= are distinguishable based on behavior)

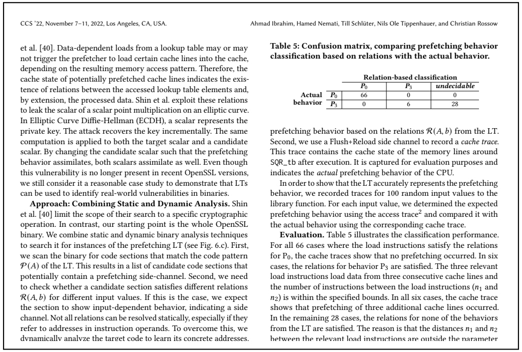
**Recall:**

$\mathcal{P}(A)$: A **code** template

$\mathcal{B}$: Distinct **behaviors**

$\mathcal{R}(A, b)$: **Relations** between inputs, leading to a certain behavior

**Leakage Template**



CCS '22, November 7–11, 2022, Los Angeles, CA, USA. Ahmad Ibrahim, Hamed Nemati, Till Schlüter, Nils Ole Tippenhauer, and Christian Rossow

et al. [40]. Data-dependent loads from a lookup table may or may not trigger the prefetcher to load certain cache lines into the cache, depending on the resulting memory access pattern. Therefore, the cache state of potentially prefetched cache lines indicates the existence of relations between the accessed lookup table elements and, by extension, the processed data. Shin et al. exploit these relations to leak the scalar of a scalar point multiplication on an elliptic curve. In Elliptic Curve Diffie-Hellman (ECDH), a scalar represents the private key. The attack recovers the key incrementally. The same computation is applied to both the target scalar and a candidate scalar. By changing the candidate scalar such that the prefetching behavior assimilates, both scalars assimilate as well. Even though this vulnerability is no longer present in recent OpenSSL versions, we still consider it a reasonable case study to demonstrate that LTs can be used to identify real-world vulnerabilities in binaries.

**Approach: Combining Static and Dynamic Analysis.** Shin et al. [40] limit the scope of their search to a specific cryptographic operation. In contrast, our starting point is the whole OpenSSL binary. We combine static and dynamic binary analysis techniques to search it for instances of the prefetching LT (see Fig. 6.c). First, we scan the binary for code sections that match the code pattern $\mathcal{P}(A)$ of the LT. This results in a list of candidate code sections that potentially contain a prefetching side-channel. Second, we need to check whether a candidate section satisfies different relations $\mathcal{R}(A,b)$ for different input values. If this is the case, we expect the section to show input-dependent behavior, indicating a side channel. Not all relations can be resolved statically, especially if they refer to addresses in instruction operands. To overcome this, we dynamically analyze the target code to learn its concrete addresses.

**Table 5: Confusion matrix, comparing prefetching behavior classification based on relations with the actual behavior.**

| | | Relation-based classification | | |
|---|---|---|---|---|
| | | $P_0$ | $P_3$ | *undecidable* |
| Actual behavior | $P_0$ | 66 | 0 | 0 |
| | $P_3$ | 0 | 6 | 28 |

prefetching behavior based on the relations $\mathcal{R}(A,b)$ from the LT. Second, we use a Flush+Reload side channel to record a *cache trace*. This trace contains the cache state of the memory lines around `SQR_tb` after execution. It is captured for evaluation purposes and indicates the *actual* prefetching behavior of the CPU.

In order to show that the LT accurately represents the prefetching behavior, we recorded traces for 100 random input values to the library function. For each input value, we determined the expected prefetching behavior using the access trace[2] and compared it with the actual behavior using the corresponding cache trace.

**Evaluation.** Table 5 illustrates the classification performance. For all 66 cases where the load instructions satisfy the relations for $P_0$, the cache traces show that no prefetching occurred. In six cases, the relations for behavior $P_3$ are satisfied. The three relevant load instructions load data from three consecutive cache lines and the number of instructions between the load instructions ($n_1$ and $n_2$) is within the specified bounds. In all six cases, the cache trace shows that prefetching of three additional cache lines occurred. In the remaining 28 cases, the relations for none of the behaviors from the LT are satisfied. The reason is that the distances $n_1$ and $n_2$ between the relevant load instructions are outside the parameter

**In the paper:** Re-identifying a known vulnerability *(Shin et al.*[1]*, CCS'18)*: Prefetching-based side channel in Elliptic Curve Diffie-Hellman (ECDH) code in OpenSSL 1.1.0g

[1] Shin et al., "Unveiling Hardware-Based Data Prefetcher, a Hidden Source of Information Leakage".

## Plumber Use Cases and Limitations

**Additional Use Cases**

- Facilitate reverse engineering of microarchitectural components
  - Examples in the paper: branch predictor, cache slice mapping

**Limitations**

- Focus on cache-based side channels
- Implemented for ARM architecture

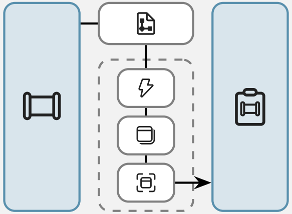
CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Stanford University

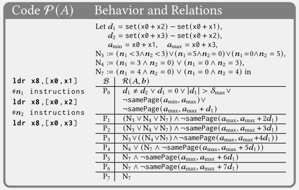
## Leakage Template

- Code
- Behaviors
- Relations

## Plumber Framework



## Case Studies



| Code $\mathcal{P}(A)$ | Behavior and Relations |
|---|---|
| | Let $d_1 = \mathtt{set}(\mathtt{x0}+\mathtt{x2}) - \mathtt{set}(\mathtt{x0}+\mathtt{x1})$, $d_2 = \mathtt{set}(\mathtt{x0}+\mathtt{x3}) - \mathtt{set}(\mathtt{x0}+\mathtt{x2})$, $a_{\min} = \mathtt{x0}+\mathtt{x1}$, $a_{\max} = \mathtt{x0}+\mathtt{x3}$, $\mathrm{N}_3 := (n_1 < 3 \wedge n_2 < 3) \vee (n_1 = 5 \wedge n_2 = 0) \vee (n_1 = 0 \wedge n_2 = 5)$, $\mathrm{N}_4 := (n_1 = 3 \wedge n_2 = 0) \vee (n_1 = 0 \wedge n_2 = 3)$, $\mathrm{N}_7 := (n_1 = 4 \wedge n_2 = 0) \vee (n_1 = 0 \wedge n_2 = 4)$ in |

```
ldr x8,[x0,x1]
#n1 instructions
ldr x8,[x0,x2]
#n2 instructions
ldr x8,[x0,x3]
```

| $\mathcal{B}$ | $\mathcal{R}(A,b)$ |
|---|---|
| $\mathrm{P}_0$ | $d_1 \neq d_2 \vee d_1 = 0 \vee \lvert d_1 \rvert > \delta_{\max} \vee \neg\mathtt{samePage}(a_{\min}, a_{\max}) \vee \neg\mathtt{samePage}(a_{\max}, a_{\max} + d_1)$ |
| $\mathrm{P}_1$ | $(\mathrm{N}_3 \vee \mathrm{N}_4 \vee \mathrm{N}_7) \wedge \neg\mathtt{samePage}(a_{\max}, a_{\max} + 2d_1)$ |
| $\mathrm{P}_2$ | $(\mathrm{N}_3 \vee \mathrm{N}_4 \vee \mathrm{N}_7) \wedge \neg\mathtt{samePage}(a_{\max}, a_{\max} + 3d_1)$ |
| $\mathrm{P}_3$ | $\mathrm{N}_3 \vee ((\mathrm{N}_4 \vee \mathrm{N}_7) \wedge \neg\mathtt{samePage}(a_{\max}, a_{\max} + 4d_1))$ |
| $\mathrm{P}_4$ | $\mathrm{N}_4 \vee (\mathrm{N}_7 \wedge \neg\mathtt{samePage}(a_{\max}, a_{\max} + 5d_1))$ |
| $\mathrm{P}_5$ | $\mathrm{N}_7 \wedge \neg\mathtt{samePage}(a_{\max}, a_{\max} + 6d_1)$ |
| $\mathrm{P}_6$ | $\mathrm{N}_7 \wedge \neg\mathtt{samePage}(a_{\max}, a_{\max} + 7d_1)$ |
| $\mathrm{P}_7$ | $\mathrm{N}_7$ |

## Matching Binaries

1. Static
2. Dynamic

**Code**

github.com/scy-phy/plumber

**Till Schlüter**

till.schlueter@cispa.de

tschlueter.com

(List of all resources related to this paper)

## Generative Testcase Specification (GTS)

### Directives

| Directive | Description |
|---|---|
| `M` | Memory Access |
| `A` | Arithmetic/Logic Instruction |
| `N` | NOP |
| `B` | Branch |
| ... | ... |

### Operators

| Operator | Description |
|---|---|
| $[\cdot]n$ | Power |
| $\#n$ | Wildcard |
| $\langle\cdot\rangle\$$ | Cache line (set) mutation |
| $\mathrm{P}(\cdot)$ | Precondition |
| ... | ... |

### Example: Cache-Timing Side Channel

$$\underbrace{\mathrm{P}(\mathtt{M}_{t_1,s_1})}_{\text{Precondition: Prime cache with a cache line in set } s_1} \quad \underbrace{\langle\mathtt{M}_{t_1}\rangle\$}_{\text{Generate one test case for each possible set index, keep the tag index constant}}$$

## Classifier

- Classifies test cases based on the observed behavior
- For each behavior: produce a *bit table*
  - Bit table: List of all test cases that trigger a certain behavior

**Bit Table**

**Behavior ○**

| **Test Case #** | **x1** | **x2** |
|---|---|---|
| 1 | 00 | 01 |
| 2 | 00 | 10 |
| ... | ... | ... |

## Analyzer

- For each bit table (= behavior): Identify common features

$\Rightarrow$ Extracts relations that trigger a certain behavior

## Prefetching on ARM Cortex-A53

- Loads cache lines in advance that are likely to be needed soon

## Steps to Create the Leakage Template

1. Number of sequential loads
2. Intermediate instructions
3. Respecting page boundary
4. Multiple prefetching sequences
5. Cache hits

Code $\mathcal{P}(A)$ Behavior and Relations

```
ldr x8,[x0,x1]
#n1 instructions
ldr x8,[x0,x2]
#n2 instructions
ldr x8,[x0,x3]
```

$$\begin{aligned}\text{Let } d_1 &= \texttt{set}(\texttt{x0}+\texttt{x2}) - \texttt{set}(\texttt{x0}+\texttt{x1}),\\ d_2 &= \texttt{set}(\texttt{x0}+\texttt{x3}) - \texttt{set}(\texttt{x0}+\texttt{x2}),\\ a_{\min} &= \texttt{x0}+\texttt{x1}, \quad a_{\max} = \texttt{x0}+\texttt{x3},\\ \mathrm{N}_3 &:= (n_1<3 \wedge n_2<3) \vee (n_1=5 \wedge n_2=0) \vee (n_1=0 \wedge n_2=5),\\ \mathrm{N}_4 &:= (n_1=3 \wedge n_2=0) \vee (n_1=0 \wedge n_2=3),\\ \mathrm{N}_7 &:= (n_1=4 \wedge n_2=0) \vee (n_1=0 \wedge n_2=4) \text{ in}\end{aligned}$$

| $\mathcal{B}$ | $\mathcal{R}(A,b)$ |
|---|---|
| $\mathrm{P}_0$ | $d_1 \neq d_2 \vee d_1 = 0 \vee \lvert d_1 \rvert > \delta_{\max} \vee \neg\texttt{samePage}(a_{\min}, a_{\max}) \vee \neg\texttt{samePage}(a_{\max}, a_{\max}+d_1)$ |
| $\mathrm{P}_1$ | $(\mathrm{N}_3 \vee \mathrm{N}_4 \vee \mathrm{N}_7) \wedge \neg\texttt{samePage}(a_{\max}, a_{\max}+2d_1)$ |
| $\mathrm{P}_2$ | $(\mathrm{N}_3 \vee \mathrm{N}_4 \vee \mathrm{N}_7) \wedge \neg\texttt{samePage}(a_{\max}, a_{\max}+3d_1)$ |
| $\mathrm{P}_3$ | $\mathrm{N}_3 \vee ((\mathrm{N}_4 \vee \mathrm{N}_7) \wedge \neg\texttt{samePage}(a_{\max}, a_{\max}+4d_1))$ |
| $\mathrm{P}_4$ | $\mathrm{N}_4 \vee (\mathrm{N}_7 \wedge \neg\texttt{samePage}(a_{\max}, a_{\max}+5d_1))$ |
| $\mathrm{P}_5$ | $\mathrm{N}_7 \wedge \neg\texttt{samePage}(a_{\max}, a_{\max}+6d_1)$ |
| $\mathrm{P}_6$ | $\mathrm{N}_7 \wedge \neg\texttt{samePage}(a_{\max}, a_{\max}+7d_1)$ |
| $\mathrm{P}_7$ | $\mathrm{N}_7$ |

Figure: Leakage Template: Prefetching.
$P_l$ means prefetching $l$ lines.

## Re-Identifying a Prefetching-Based Vulnerability in OpenSSL

**Vulnerability *(Shin et al.*[2] *CCS'18)*:**
Prefetching-based attack on Elliptic Curve Diffie-Hellman (ECDH) in OpenSSL 1.1.0g

1. **Static Analysis**
   - Search for the code template from the prefetching Leakage Template

   ⇒ Identified 429 matching sequences across 18 OpenSSL modules (including the target code section)
2. **Dynamic Analysis**
   - Run code with different inputs
   - Evaluate register contents against relations

   ⇒ Different inputs satisfy relations for different behaviors

**Conclusion:** Different classes of inputs are distinguishable based on prefetching behavior.

[2] Shin et al., "Unveiling Hardware-Based Data Prefetcher, a Hidden Source of Information Leakage".

## References

[1] Hamed Nemati et al. “Validation of Abstract Side-Channel Models for Computer Architectures”. In: *International Conference on Computer-Aided Verification (CAV)*. 2020. DOI: 10.1007/978-3-030-53288-8_12.

[2] Youngjoo Shin et al. “Unveiling Hardware-Based Data Prefetcher, a Hidden Source of Information Leakage”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS ’18)*. 2018. DOI: 10.1145/3243734.3243736.