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

Ahmad Ibrahim*
CISPA Helmholtz Center for Information Security
ahmad.ibrahim@cispa.de

Hamed Nemati*
Stanford University
CISPA Helmholtz Center for Information Security
hnnemati@stanford.edu

Till Schlüter
CISPA Helmholtz Center for Information Security
till.schlueter@cispa.de

Nils Ole Tippenhauer
CISPA Helmholtz Center for Information Security
tippenhauer@cispa.de

Christian Rossow
CISPA Helmholtz Center for Information Security
rossow@cispa.de

ABSTRACT

The complexity of modern processor architectures has given rise to sophisticated interactions among their components. Such interactions may result in potential attack vectors in terms of side channels, possibly available to userland exploits to leak secret data. Exploitation and countering of such side channels requires a detailed understanding of the target component. However, such detailed information is commonly unpublished for many CPUs.

In this paper, we introduce the concept of Leakage Templates to abstractly describe specific side channels and identify their occurrences in binary applications. We design and implement Plumber, a framework to derive the generic Leakage Templates from individual code sequences that are known to cause leakage (e.g., found by prior work). Plumber uses a combination of *instruction fuzzing*, *instructions' operand mutation* and *statistical analysis* to explore undocumented behavior of microarchitectural optimizations and derive sufficient conditions on vulnerable code inputs that if hold can trigger a distinguishing behavior. Using Plumber we identified novel leakage primitives based on Leakage Templates (for ARM Cortex-A53 and -A72 cores), in particular related to *previction* (a new premature cache eviction), and prefetching behavior. We show the utility of Leakage Templates by re-identifying a prefetcher-based vulnerability in OpenSSL 1.1.0g first reported by Shin et al. [40].

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**.

KEYWORDS

microarchitecture, side channel, leakage templates

*Both authors contributed equally to the paper

ACM Reference Format:
Ahmad Ibrahim, Hamed Nemati, Till Schlüter, Nils Ole Tippenhauer, and Christian Rossow. 2022. Microarchitectural Leakage Templates and Their Application to Cache-Based Side Channels. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3548606.3560613

1 INTRODUCTION

The past decade has witnessed a surge in side-channel attacks that exploit underspecified or undocumented hardware features [3, 16, 18, 23, 30, 44, 55], mostly focusing on cache-related leakage. The hidden nature of microarchitectural features has led to the development of techniques to test for the presence in hardware, and semi-automatically identify new vulnerabilities [11, 27, 28, 31, 37, 42, 43, 51] that allow the attacker to violate process isolation to obtain secret data, or to manipulate the victim's execution. Existing approaches to identify side channels commonly yield architecture-specific *distinguishing examples*, i.e., concrete code examples that represent side-channel leakage. Generalization from such concrete examples is a known hard problem, as it requires a detailed understanding of the processor component that introduces the channel. Details on information flow properties of microarchitectures are generally scarce, not publicly available, or depend on industrial secrets. As result, determining whether a given application is vulnerable to cache-related side-channel leakage is challenging.

Recent approaches already demonstrate the power of automation in the context of side channel analysis [11, 27, 37, 42, 51]. Gras et al. propose ABSynthe [11] to automatically infer *leakage maps* that show how instructions influence each other's contention behavior. Their system can identify and optimize for microarchitecture-specific side channels exploiting hyperthreading. In addition, Weber et al. introduce a fuzzing framework Osiris that synthesizes instruction sequences to identify timing-based side channels [51]. To this end, Osiris proposed an automated system to identify sequences that trigger and reset certain microarchitectural states. Finally, with Transynther [27], Moghimi et al. present an approach to synthesize Meltdown-type attacks. Transynther varies known attack patterns to create candidate attack code snippets and then evaluates whether these snippets leak data. All of these approaches have in common that they automate the search for new (variants of) side channels, a big leap towards automation. Having said this, they (i) are specific towards their use case (e.g., contention, Meltdown), (ii) limit their search space (e.g., instruction operands are largely ignored), and (iii) focus on attack code generation instead of finding a generic pattern for vulnerabilities that can be matched in existing code.

In this work, we introduce the concept of *Leakage Templates* (or LTs for short), which consist of a generalized code sequence and a set of relations on input parameters that, when satisfied, can trigger specific leakage behavior in victim code (see § 3.4 for more details). Given such an LT for a target hardware platform, we can identify code sequences (and required input values) that expose a specific side channel behavior (see § 7 for an example). We thereby address two research questions: (1) How can we learn generic LTs for largely-undocumented leakage behavior? (2) How can we use the LTs to find side channel vulnerabilities?

To this end, we design and implement Plumber (source is available at [36]) to facilitate generating such LTs, leveraging *instruction fuzzing*, *instructions' operands mutation*, and *statistical analysis*. The design of Plumber is based on exploring the architectural space through the execution of program-input pairs, and analyzing the resulting microarchitectural states (focusing on caching). We design a domain-specific language that simplifies the generation of a large number of instruction sequences (i.e., programs) and mutating their operands. Further, we use a statistical analysis approach to classify microarchitectural states and to extract relations on inputs. To validate our approach, we studied the cases of the cache replacement policy of the ARM Cortex-A72 and two microarchitectural features of the ARM Cortex-A53 processors: *previction* and *prefetching*.

Previction is a recently discovered yet widely undocumented processor behavior of evicting cache lines *before* the corresponding cache set is full [28]. Since previction behaves differently for two addresses that only differ in their cache lines' offset, it may violate existing assumptions used to secure software [30]. Prefetching, on the other hand, is a partially documented feature that allows the processor to detect regular memory access patterns and to fill cache lines with anticipated addresses by proactively continuing the pattern. However, many details of prefetching (e.g., the number of prefetched lines) are undocumented. In addition, the cache replacement policy of ARM processors is not well-documented.

Leveraging Plumber, we analyze cache replacement policy, previction and prefetching behavior of the processor's core and derive related LTs. For the replacement policy experiment, the derived LT establishes eviction strategies of L1 data cache. In the case of previction, the extracted LTs reveal conditions under which bits of memory address loads are leaked.

For prefetching, we leverage Plumber to discover parameters such as the minimal number of loads to trigger prefetching, the impact of intermediate instructions, the impact of page boundaries on prefetching, and the impact of cache hits. In other words, we show how the obtained LTs expose prefetching side channels that allow to infer the control flow of a program and to leak secret information. Those channels are different from existing prefetching-based attacks discussed in the literature [6, 12, 40, 49], which target the x86 architecture and either attack a software-based prefetcher [12] or use a simulated CPU [6] for their analysis. Most similar to our approach is the work of Shin et al. [40] which exploits the hardware prefetcher to attack a constant-time Elliptic Curve Diffie-Hellman (ECDH) implementation from OpenSSL. However, compared to ours, they used a different side channel, i.e. the effect of the order of accessed lookup table entries on the behavior of the prefetcher.

Although our focus in this paper is on the ARM architecture and cache-based side channels, the Plumber design is generic and can be used to detect LTs for other features like the *cache slice mapping* of Intel processors (see Appendix B for a discussion). Also, while the main goal of Plumber is to ease generating LTs, it can also be used to help reverse engineering of undocumented microarchitectural features. For example, § 10 shows how Plumber can be applied to discover the structure of the Cortex-A53's branch predictor.

Contributions. Our main contributions are as follows:

- We introduce the concept of *Leakage Templates*, which allow to identify code sequences (and required values) that expose a specific side channel behavior in a binary application executed on a specific architecture (§ 3).
- We design and implement Plumber, a framework that generates LTs, and allows to obtain a deeper understanding of hidden behavior of microarchitectures (§ 4, § 5, and § 10).
- We show Plumber's efficacy by investigating the undocumented eviction policy of Cortex-A72's L1 cache and previction and prefetching behaviors of the Cortex-A53 processors (§ 6), and identify five novel side channels (§ 8).
- We demonstrate how a derived LT can be used to identify a side channel in a binary application (§ 7).

2 BACKGROUND

Side channels are hidden information flow paths, which are potentially exploitable by an attacker to leak data. The number of attacks exploiting microarchitectural features, like caches [3, 30, 44, 45, 55], continues unabated. Therefore, the study of information flow analysis techniques to ensure the absence of side channel leakages is a topic of increasing relevance. Central to such an analysis is a model capturing the channel. However, the complexity of modern processors and the lack of information about their features make infeasible to explicitly model all the relevant, complex, and intertwined features like cache hierarchies and out-of-order execution. Abstract *observational models* tackle this problem by over-approximating attacker capabilities to observe flow of information via side channels.

2.1 Information Flow Analysis Tools – Scam-V

A key requirement of observational models is their *soundness*, i.e. observationally equivalent states should lead to executions that cannot be distinguished by an attacker on real hardware. Scam-V [28] automates validation of observational models' soundness. At high level, Scam-V generates well-formed random binary programs, which are denoted by $\mathcal{P}$. It then constructs pairs of initial states $\mathcal{I}$ s.t. executions of $\mathcal{P}$ from these states are observationally indistinguishable on the model. Scam-V then tests if the two states are also indistinguishable on hardware in the presence of undocumented microarchitectural components. Any experiment which enables the attacker to distinguish one microarchitecture-level execution from another represents a *counterexample* to the soundness of the observational model. We call a counterexample a *leakage witness* in this paper. A leakage witness can be seen as an instantiation of our Leakage Templates that can be used by an attacker to infer a function of the (secret) data via relevant side channels.

Code Template	Input 1 (Tag/Set)	Input 2 (Tag/Set)
`ldr x3, [x0]`	x0: **0x80100000**(T_1/S_1)	**0x80100020**(T_1/S_1)
`ldr x24, [x1]`	x1: 0x80100010(T_1/S_1)	0x80100010(T_1/S_1)
`ldr x9, [x2]`	x2: 0x80100000(T_1/S_1)	0x80100000(T_1/S_1)
`ldr x2, [x3]`	x3: 0x8013e000(T_2/S_1)	0x8013e000(T_2/S_1)
`ldr x15, [x4]`	x4: 0x80104000(T_3/S_1)	0x80104000(T_3/S_1)

Figure 1: A previction counterexample.

Previction. Previction is an undocumented behavior of the ARM Cortex-A53 processors which invalidates cache-related observational models. Previction causes a cache line to be evicted *before* the corresponding cache set is full. Scam-V [28] discovers a handful instances of this behavior. However, the real cause of *previction* is unknown. The authors conjecture that the processor detects a short sequence of loads to the same cache set and anticipates more loads to the same set with no reuse of previously loaded values. It evicts the valid cache line in order to make space for more colliding lines.

Fig. 1 depicts a previction counterexample. The program consists of five loads. The given inputs are observationally equivalent: they only differ for the value of x0, which affects the address used for the first load. However, the addresses 0x80100000 and 0x80100020 in x0 have the same tag and cache set index and only differ in the offset within the same cache line. The addresses of all load instructions are mapped to the same cache set, i.e., set 0. Since the cache is 4-way associative and is initially empty, one expects no eviction to occur. Executing the program with the given inputs on the real hardware, however, results in two different cache states. In one case x0 is present in the cache, while in the other case it is not.

2.2 ARMv8 internal memory subsystem

ARMv8 processors have two levels of caches: (1) a level one (L1) cache per each core; and (2) a last level cache (L2) shared between cores. When the CPU needs to read a memory location that is currently not cached, it fetches the requested data from memory into a cache line and tags that line with the memory location where the data was read from. When a line is loaded from memory and all potential destination lines in the cache are occupied, the CPU uses a specific replacement policy to decide which colliding line(s) should be evicted. The eviction policy ensures that the cache always stores the most popular content, thus using the available space efficiently.

Table 1 shows our notation to extract cache-related information from an address. `sameTag`, `sameSet` and `samePage` are predicates checking for equality of cache tag, set, or page indices of addresses.

Prefetcher. The L1 cache implements a prefetcher, for some configurable $k \in \mathbb{N}$. When the prefetcher detects k cache misses whose set indices are separated by a fixed stride, the prefetcher starts to fill the cache with a sequence of lines from memory locations whose addresses match the stride of the initial cache misses. We call such sequences *prefetch streams*. An exception happens when prefetching crosses a small page (4K) boundary. In this case the prefetcher stops fetching data from the adjacent page.

3 LEAKAGE TEMPLATES

We now introduce *Leakage Templates* and motivate their utility.

Table 1: Summary of notations.

Notation	Description
a_i	A physical address
${a_i}^{m-n}$	Bits m through n of a_i
$\texttt{set}(a_i) = {a_i}^{6-12}$	a_i cache set index
$\texttt{tag}(a_i) = {a_i}^{13-31}$	a_i cache tag
$\texttt{word}(a_i) = {a_i}^{2-5}$	a_i word offset
$\texttt{bus}(a_i) = {a_i}^{4-5}$	a_i bus round
$\texttt{page}(a_i) = {a_i}^{12-31}$	a_i page index
$\texttt{sameTag}(a_i, a_j, \ldots)$	Cache tag equality predicate
$\texttt{sameSet}(a_i, a_j, \ldots)$	Cache set equality predicate
$\texttt{samePage}(a_i, a_j, \ldots)$	Page index equality predicate

3.1 Goal and Motivation for Leakage Templates

Goal of the analyst. The overall setting is depicted in Fig. 3. We assume that the analyst has concrete examples of (artificial) code (for a specific hardware architecture) that behave distinctively on the microarchitectural level for different inputs, thus exposing an undocumented behavior. We call these examples *distinguishing examples*. Plumber takes as input an abstract description of a leaking code snippet in terms of a *Generative Testcase Specification* (GTS) (see § 4.1 for more details). The goal of the analyst is to utilize this behavior to leak information from a real-world application. To achieve this, the analyst needs to identify code segments in the target application that *under certain analyst-controllable conditions* trigger the undocumented microarchitectural behavior.

Motivation for Leakage Templates. As the analyst starts with concrete code sequences (in form of distinguishing examples), it is unlikely that the exact same code sequences will appear in another target application. Therefore, we need to abstract from the concrete code sequences and find: (1) a generalized code sequence, (2) a set of relevant attributes, and (3) relations between those attributes that expose the specific side channel. We call such information a *Leakage Template* or *LT*. The LT abstractly defines the conditions for code segments under which the leakage is observed. Given an LT, an analyst can identify code segments in a target application that expose a side channel. We demonstrate in § 7 how binaries can be scanned for code sections that match the code pattern from a LT. In addition, we show how dynamic binary analysis techniques can be used to analyze a matching code section for the presence of side-channel leakage based on the relations from a LT.

Sources of distinguishing examples. There are at least two general ways of finding distinguishing examples. Such examples can be derived from abstract (natural language) descriptions of the architecture's behavior, e.g., in manuals. Moreover, one can extract these examples from concrete code traces that expose the intended side channel, e.g., out of tools like Scam-V or zero-day exploits.

3.2 Motivating Example I: Caching

Assume that simple caching behavior of a CPU was insufficiently documented, and the analyst tries to understand in which parts of a target application (and under which conditions) caching occurs.

Initial Information. A starting *leakage witness* (i.e. a distinguishing code example with a pair of inputs) could contain a sequence of two load instructions (for variable addresses), and two instances (i.e., pairs of input addresses) where different behavior was observed. In the first instance, both loads refer to the same address, while in the second instance, two distinct addresses (on different cache lines) are accessed. When executing the two instances on a clean cache state, the second load will be considerably faster for the first instance compared to the second one.

Leakage Template. An abstract LT that describes this side channel would specify: (1) an abstract code template (i.e., at least two loads from symbolic addresses with potentially other instructions in between); (2) the two possible behaviors (i.e., fast and slow execution time of the second load); and (3) the relations over the loaded addresses which lead to the respective behaviors.

Benefit of the Leakage Template. The starting leakage witness just provides one concrete code instance that leads to the leakage behavior, while the LT ideally covers every possible sequence of instructions in which two loads lead to caching. Consequently, one would expect that the leakage witness alone will not enable the analyst to identify instances of the side channel leakage in an arbitrary target program, while the LT is expected to have much higher chances to discovering related code fragments.

3.3 Motivating Example II: Prevection

For previction, the leakage behavior appears to require a more complex code sequence (see § 2.1).

Initial information. In this case, a starting leakage witness could contain a sequence of five loads, and two sets of addresses which lead to different behaviors, as summarized in Fig. 1.

Leakage Template. An LT that describes this side channel would specify: (1) an abstract code template (e.g., at least five loads from symbolic addresses with potentially a number of other instructions in between); (2) the two possible behaviors (i.e., previction or no previction); and (3) the abstract relations over the loaded addresses which lead to the respective behaviors.

3.4 Definition of Leakage Templates

Based on the provided motivation for LTs, we now describe the components of LTs themselves. An LT is a triple $(\mathcal{P}(A), \mathcal{B}, \mathcal{R}(A, b))$; with $\mathcal{P}(A)$ being a *code template* with a set of attributes $A \subseteq \mathbb{A}$, $\mathcal{B}$ a set of observed distinct behaviors, and $\mathcal{R} : \mathbb{A} \times \mathcal{B} \rightarrow 2^{\mathbb{P}_A}$ maps a behavior $b \in \mathcal{B}$ to a (set of) predicate(s) on attributes, where $\mathbb{P}_A$ is the set of predicates on A. We note that our definition of behavior is generic. For instance, it may refer to temporal (e.g., measuring execution time) or spatial (e.g., monitoring cache content) behavior. For the latter, the behavior refers to the difference between the initial (before execution of the code template) and final (after execution of the code template) state of the monitored component (e.g., cache). Given this description, we provide additional details on the resulting LT for our caching example in Fig. 2. In § 6.2 we show how an LT for the previction example can be derived.

4 PLUMBER

We now present our design of Plumber, a framework to automatically derive LTs (focusing on caching behavior). In particular, in our design we had to address the following challenges:

- **C1**: We need to construct efficient specifications to steer testcase generation towards generating inputs which are likely to trigger a specific microarchitectural behavior.

Code $\mathcal{P}(A)$	Behavior and Relations	
`ldr x0, [x1]`	$\mathcal{B}$	$\mathcal{R}(A, b)$
`; 0-n NIIs`	(•)	$\texttt{sameTag}(x_1, x_2) \wedge \texttt{sameSet}(x_1, x_2)$
`ldr x0, [x2]`	(◦)	$\neg\texttt{sameTag}(x_1, x_2) \vee \neg\texttt{sameSet}(x_1, x_2)$

Figure 2: Caching LT. • (fast), ◦ (slow) are distinguishing behaviors. NIIs are Non-Interfering Instructions.

- **C2**: Size of input space (number of possible code sequences and input values) is too large. Thus, we need an approach to explore the input space and ensure high test coverage which in turn increases accuracy of the derived relations.
- **C3**: We need to get accurate measurements with minimal noise, close to 'ground truth' of the respective channel.
- **C4**: The relations between attributes in code sequences and triggered behaviors can be very complex and counter-intuitive, and manual derivation of such relations (if not impossible) is highly error prone. So, we need to develop a statistical analysis technique to automate finding the correlation between attributes and observed behavior.

4.1 Abstract Framework Design

Plumber's input is a *Generative Testcase Specification* (GTS), which abstractly describes programs to analyze in a domain-specific language, and whose leakage effects are to be monitored. The detailed description of GTS and the language (and how it addresses **C1**) is provided in § 4.2 and 4.3. The framework outputs the LT, including the behavior of the monitored microarchitectural components, and the relations between attributes and the behaviors.

As shown in Fig. 3, the framework consists of two parts. The *Backend* instantiates and executes testcases (i.e., program-input pairs) from a (preprocessed) GTS. The *Frontend* encapsulates the Backend, towards the outside it receives a GTS and outputs LTs.

We present each component by describing the steps required to derive an LT from a GTS. ① The *Preprocessor* parses the GTS and forwards the result to the *Testcase Instantiator.* ② The Instantiator then systematically generates testcases (addressing **C2**) based on the parsed preprocessed GTS. ③ The *Runner* executes every testcase in a controlled environment (addressing **C3**) and returns the behavior to the *Classifier.* ④ The Classifier classifies the programs (if necessary) based on their behavior and stores them. ⑤ The *Analyzer* interprets classified behaviors (addressing **C4**) and returns relations on the attributes which trigger specific microarchitectural behavior (i.e., $\mathcal{B}$ and $\mathcal{R}(A, b)$ of the LT).

A possible application of an LT is to identify instances of it in binaries. This can be done using a *Template Matcher* ⑥, which may use static and dynamic binary analysis techniques to find code sections matching the LT's code pattern as well as its relations. We discuss a proof-of-concept implementation in § 7.

4.2 Definition of GTS

The GTS (used as input of Plumber) intuitively defines sequence(s) of instructions to be executed over (mutated) operands, and microarchitectural component(s) to be monitored (e.g., content of the cache). The GTS can also specify the initial state of (monitored) component(s) before the execution of its instruction sequences.

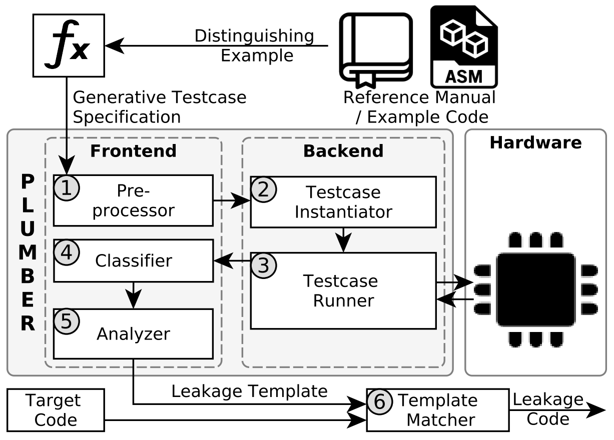


Figure 3: Overview of Plumber components.

To address **C1** as outlined earlier, we introduce a domain-specific generative language. The language provides three main features. (i) It allows us to abstractly specify possible mutations of the program (i.e. enabling very generic code templates), (ii) it allows us to specify which data in the programs should be fuzzed or mutated (i.e. defining domains for attributes), and (iii) it allows us to specify a set of relations to apply to non-fuzzed data in the testcases.

4.3 Specification Language

A Generative Testcase Specification (GTS) is formed of a sequence of directives that specify different operations, e.g., arithmetic operations or nop instructions. The proposed language defines an extensible set of directives, which includes M, A and N. The directive M denotes memory loads, the directive A denotes arithmetic or logical operations, and the directive N denotes nop instructions. Additionally, the language defines two directives that allow reverse engineering the branch predictor (see § 10). $S_{c_1,bool}$ denotes an instruction that sets a variable (identified by c_1) to a boolean value bool, and $B_{c_1,bool,step}$ denotes a branch instruction that jumps step steps when the value of the variable identified by c_1 is bool. The default value of a variable is false (F).

The addresses of load operations are mapped to a certain cache set and have specific tags. It is possible to define the tag and set attributes of a memory directive. We use M_{t_1,s_1} to refer to a load from an address with tag t_1 and cache set s_1. The values of these attributes allow determining the relation between tags or sets of different loads. When omitted, these attributes acquire default values which are identical for the same request. The language allows defining arithmetic relations between the tag (and set) attributes of different memory directives. For example, the GTS M M M_{t_1,s_1} M_{t_1,s_1} M_{t_1+1,s_1+5} A A N represents two loads of memory addresses with the same set and the same tag (default), followed by two other loads with a different tag t_1 and a different set s_1 and a fifth load with tag ($t_2 = t_1 + 1$) and set ($s_2 = s_1 + 5$), two arithmetic or logical operations, and a nop instruction. The values of the operands of other instructions, e.g., arithmetic operations, can be defined in the same manner, e.g., A_{v_1,v_2} represents an arithmetic operation with two operands v_1 and v_2.

The language also provides an extensible set of macros and operators that allow constructing a meaningful GTS as well as defining the initial state of hardware components. Note that this language is extensible, i.e., it allows defining new macros and operators for investigating various processor components.

Power $[\,]\mathtt{dir}_{\mathtt{attr}}, n, i$**.** This macro allows repeating directive(s) *n* times, while incrementing the attribute attr of the directive dir by a value of *i*. For example, the GTS $\left[M_{t_1,s_1}\right]M_s, 2, 1$ can be used to refer to the GTS: $M_{t_1,s_1}\ \ M_{t_1,s_1+1}$. The power macro can be also used with a single input *n*. In this case the directive(s) are repeated *n* times. For example, the GTS presented above can also be expressed as: $[M]2\ \ \left[M_{t_1,s_1}\right]2\ \ [A]2\ \ N$.

Wildcard #n**.** This macro expands to *n* arbitrary directives that do not perform memory operations. For example, one possible expansion of M #3 M is M N N N M.

Shuffle ()!**.** This operator generates all possible permutations of a GTS while omitting those with similar directives. For example, $\left([M]2\ \ M_{t_1,s_1}\right)!$ refers to the set: $\{M\ M\ M_{t_1,s_1}; M\ M_{t_1,s_1}\ M; M_{t_1,s_1}\ M\ M\}$.

Subset ()⊂**.** This operator generates all possible subsets of a GTS while omitting those with similar directives. For example, $\left([M]2\ \ M_{t_1,s_1}\right)\subset$ refers to the set: $\{M\ M; M\ M_{t_1,s_1}; M; M_{t_1,s_1}\}$.

Slide ()n**.** For a given GTS, this operator shifts all loaded addresses one set at a time up to *n* times. For example, $\left(M_{t_1,s_1}\ \ M_{t_2,s_2}\right)3$ refers to the set: $\{M_{t_1,s_1}\ \ M_{t_1,s_1}; M_{t_1,s_1+1}\ \ M_{t_1,s_1+1}; M_{t_1,s_1+2}\ \ M_{t_1,s_1+2}\}$.

Merge (:)+**.** This operator merges two requests by sliding the directives of the first over the second as demonstrated by the following example. $\left(M_{t_1,s_1}\ \ M_{t_2,s_2} : M_{t_3,s_3}\ \ M_{t_4,s_4}\right)+$ refers to the set:

$$\left\{\begin{matrix} M_{t_1,s_1}\ M_{t_2,s_2}\ \ M_{t_3,s_3}\ \ M_{t_4,s_4};\ M_{t_1,s_1}\ \ M_{t_3,s_3}\ \ M_{t_2,s_2}\ M_{t_4,s_4}; \\ M_{t_3,s_3}\ M_{t_1,s_1}\ \ M_{t_4,s_4}\ \ M_{t_2,s_2};\ M_{t_3,s_3}\ \ M_{t_4,s_4}\ \ M_{t_1,s_1}\ M_{t_2,s_2} \end{matrix}\right\}$$

Load offset mutation ⟨⟩@**.** For every load instruction, the operator signals generation of a testcase for all possible addresses with the indicated tag and set, i.e., it brute forces word offsets. For example, ⟨M M⟩@ generates a set formed of all two loads having the same tag and set with all possible combinations of word offsets.

Cache line mutation ⟨⟩$**.** For every load instruction, this operator signals the generation of a testcase for every possible memory address having the indicated tag and word offset, i.e., it brute forces all possible sets. For example, ⟨M M⟩$ generates a set formed of all two loads that have the same tag for all possible combinations of sets, i.e., $\{M_{t_1,s_1}\ \ M_{t_1,s_2}\}$ for every set s_1 and s_2.

Repetition ||n**.** This operator repeats the GTS *n* times, e.g., the GTS $\left|M\ \ M_{t_1,s_1}\right|3$ corresponds to: $\{M\ \ M_{t_1,s_1}\ ;\ M\ \ M_{t_1,s_1}\ ;\ M\ \ M_{t_1,s_1}\}$.

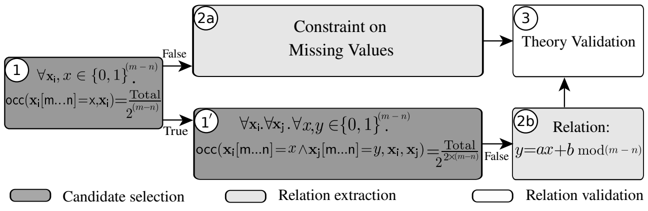
Precondition P()**.** This operator allows setting up the state of different hardware components before the execution of testcase. For instance, the GTS $P\left(M_{t_1,s_1}\ \ M_{t_2,s_1}\right)$ ⟨M M⟩$ generates cache line mutation testcases where two lines in s_1 are already occupied.

5 DESIGN & IMPLEMENTATION

Plumber currently targets the ARM architecture. It is implemented in C and Python as well as ARM assembly. We exploit ARM assembly for: (a) implementing testcases, (b) setting up and (accurately) reading architectural components, and (c) increasing the performance. However, our design is applicable to other architectures. In the following, we present the implementation details for Plumber's components presented in Fig. 3.

	Load Register				
Output	x0	x1	x2	x3	x4
1	1 · · · 000000000**0000**00	1 · · · 000000000**0000**00	· · ·	· · ·	1 · · · 100000000**0000**00
2	1 · · · 000000000**0001**00	1 · · · 000000000**0000**00	· · ·	· · ·	1 · · · 100000000**0000**00
3	1 · · · 000000000**0010**00	1 · · · 000000000**0000**00	· · ·	· · ·	1 · · · 100000000**0000**00
4	1 · · · 000000000**0011**00	1 · · · 000000000**0000**00	· · ·	· · ·	1 · · · 100000000**0000**00
5	1 · · · 000000000**0100**00	1 · · · 000000000**0000**00	· · ·	· · ·	1 · · · 100000000**0000**00
6	1 · · · 000000000**0101**00	1 · · · 000000000**0000**00	· · ·	· · ·	1 · · · 100000000**0000**00
7	1 · · · 000000000**0110**00	1 · · · 000000000**0000**00	· · ·	· · ·	1 · · · 100000000**0000**00
8	1 · · · 000000000**0111**00	1 · · · 000000000**0000**00	· · ·	· · ·	1 · · · 100000000**0000**00
9	1 · · · 000000000**1000**00	1 · · · 000000000**0000**00	· · ·	· · ·	1 · · · 100000000**0000**00
			· · ·		
2^{20}	1 · · · 000000000111100	1 · · · 000000000111100	· · ·	· · ·	1 · · · 100000000111100

Figure 4: Input bit table from a class causing previction. Shaded columns are registers responsible for previction. Shaded rows are inputs missing from the table. Mutated bits are in bold and the bits causing previction are underlined.



1
$\forall \mathbf{x}_i, x \in \{0,1\}^{(m-n)}.$
$\mathrm{occ}(\mathbf{x}_i[m...n]=x,\mathbf{x}_i)=\frac{\mathrm{Total}}{2^{(m-n)}}$
False
True
2a
Constraint on Missing Values
3
Theory Validation
1′
$\forall \mathbf{x}_i.\forall \mathbf{x}_j.\forall x,y \in \{0,1\}^{(m-n)}.$
$\mathrm{occ}(\mathbf{x}_i[m...n]=x \wedge \mathbf{x}_j[m...n]=y, \mathbf{x}_i, \mathbf{x}_j)=\frac{\mathrm{Total}}{2^{2\times(m-n)}}$
False
2b
Relation:
$y=ax+b \bmod (m-n)$
Candidate selection
Relation extraction
Relation validation

Figure 5: The analyzer function flowchart for previction.

5.1 Frontend

The Frontend handles user interaction and has three components:

Preprocessor. The preprocessor interprets a given GTS and forwards the results to the Backend. It expands repetition, power and wildcard macros. Wildcard is expanded by randomly picking directives from the set of defined operations. The Preprocessor is also responsible for expanding the shuffle, subset, slide and merge operators, i.e., it generates permutations of a given GTS by applying the respective operations and forwards them to the Backend.

Classifier. This component classifies the output of the Backend based on the behavior of the monitored component, e.g., for previction/prefetching, output is classified based on the occurrence of previction/prefetching (or the previcted/prefetched addresses). For every behavior, the classifier generates a *bit table* containing the *binary representation* of mutated instruction operands, e.g., accessed addresses for cache line mutation. Fig. 4 shows an example of a bit table generated for previction. Each testcase is represented by one row in the table, each column represents a loaded address.

Analyzer. The analyzer extracts relations between inputs and their effect on the monitored microarchitectural behavior of the component. It exposes a set of primitive operations over bit tables to generate and validate such relations. Our primitives have similar meaning as SQL statements. Examples include: (a) *count*() which counts the number of rows (or columns) in a table; (b) the variadic function $select(cond(\mathbf{x}_i[m_i \dots n_i], ...), \mathbf{x}_i[m_i \dots n_i], ...)$ that returns all rows whose fields are in the relation *cond*, e.g., $select(\mathbf{x}_i[j] = 1, \mathbf{x}_i)$ returns rows where the jth bit of the register $\mathbf{x}_i$ is 1; (c) the variadic function $relation(\mathbf{x}_i[m_i \dots n_i], \mathbf{x}_j[m_j \dots n_j], \dots)$ which takes two or more inputs and returns a linear relation over specific bits of the inputs. We define an example analyzer function for previction. Other analyzer functions typically follow the same strategy. As shown in Fig. 5, this function has three phases:

Candidate selection. This step pinpoints parts of the inputs (i.e., specific bits from every address) that are correlated with the observed behavior (e.g., previction). Let $\circ$ denote function composition; occ = *count* $\circ$ *select* defines the composition of *count* and *select*. The analyzer determines the candidate bits as follows:

For every address $\mathbf{x}_i$, the analyzer uses $\mathrm{occ}(\mathbf{x}_i[m \dots n] = x, \mathbf{x}_i)$ to find the number of occurrences of each possible value $x \in \{0,1\}^{(m-n)}$ for the *non-constant bit sequence* indexed by m and n, i.e., mutated bits (step ①). It compares these actual occurrences to the expected number of occurrences of this value, i.e., it checks whether $\mathrm{occ}(\mathbf{x}_i[m \dots n] = x, \mathbf{x}_i) = \frac{\mathrm{Total}}{\mathrm{n_{pos}}}$; where Total denotes the size of the bit table; and $\mathrm{n_{pos}} = 2^{(m-n)}$ is the number of different possible values for the bit sequence indexed by m and n. Every address where the number of occurrences of (some or all) values deviates from the expectation is marked as a *candidate address*.

If all addresses have the same number of occurrences equal to $\frac{\mathrm{Total}}{\mathrm{n_{pos}}}$, the analyzer proceeds to step ①′. In this step, the analyzer repeats the previous check on pairs of operands. For every pair of operands $\mathbf{x}_i$ and $\mathbf{x}_j$ the analyzer checks whether $\mathrm{occ}(\mathbf{x}_i[m \dots n] = x \wedge \mathbf{x}_j[m \dots n] = y, \mathbf{x}_i, \mathbf{x}_j) = \frac{\mathrm{Total}}{\mathrm{n'_{pos}}}$, for all possible values $x, y \in \{0,1\}^{(m-n)}$, where $\mathrm{n'_{pos}} = 2^{2\times(m-n)}$ is the number of possible values for the two bit sequences. Every pair of addresses for which the number of occurrences of (some or all) values deviates from the expectation is marked as a pair of *candidate interrelated addresses.*

Relation extraction. Next, the analyzer detects the constraints on certain bits in a candidate address or the interrelation between bits in candidate interrelated addresses. In step ⓶ⓐ, the analyzer uses occ to determine these constraints by checking every candidate address for the missing values. In step ⓶ⓑ, the analyzer uses the function $relation(\mathbf{x}_i[m \dots n], \mathbf{x}_j[m \dots n])$ to find the interrelation between the non-constant bits of interrelated candidate addresses. It finds a and b in the equation $y = ax + b \mod (m-n)$, where x and y represent the interrelated bits in the two addresses, respectively.

Relation validation. ③ This phase validates the generated constraints and relations. A constraint is validated by using the occ function to check whether all value combinations of unrelated bits occur in the bit table. A relation is validated by using $\mathrm{occ}(\text{"}y = ax + b\text{"}, \mathbf{x}_i[m \dots n], \mathbf{x}_j[m \dots n])$ to check whether: (i) every row in the bit table satisfies the extracted relation and (ii) every value combination of unrelated bits occurs in the bit table.

5.2 Backend

The Backend is responsible for generating concrete testcases (a set of programs and their inputs) from a GTS, setting up microarchitectural components, executing the programs over their respective inputs on real hardware, and returning the microarchitectural behavior of this execution. The Backend is formed of two components:

Instantiator. This component receives an expanded GTS from the Preprocessor and generates concrete testcases. It uses an instruction store to pick instructions for each operation when generating programs. It further uses an address store to generate inputs. In particular, for the first occurrence of every tag and/or set attribute value, the generator picks random values from the store and queries their corresponding address. For every consecutive occurrence of a

Table 2: Approximate total execution time for the experiments. d, m **and** s **stand for** ***day(s), minute(s)*** **and** ***second(s).***

Case study		Execution time				
	Eviction	$62m$				
	Previction	E1	E2	E3	E4	E5
		$0.7d$	$6s$	$7m$	$1.5d$	$> 1s$
	Prefetching	E6	E7	E8	E9	E10
		$4d$	$6m$	$> 1s$	$> 1s$	$0.2d$

tag or set, the generator uses the previously selected tag and/or set from the first occurrence. In the case of arithmetic relation between these attributes, the generator searches for addresses that satisfy this relation. Alternatively, the Testcase Instantiator can generate new addresses and add them to the address store. Finally, the Instantiator generates and/or mutates inputs of instructions as requested. For example, when the GTS includes an offset mutation operator, the generator generates inputs with all possible address combinations by brute forcing their word offset. The generated testcases are forwarded to the Runner. When the cache line mutation operator is used, the Instantiator generates inputs with all possible address combinations by brute forcing the set index.

Runner. The Testcase Runner receives a testcase from the Instantiator to be executed on the hardware. The Runner first inserts memory barriers between the program's instructions. It then connects to the hardware, and refreshes the microarchitectural state, e.g., clears the cache. The Runner then sets up microarchitectural components by executing the precondition part of the program. The program is then executed using ARM TrustZone. ARM TrustZone provides the highest level of privilege that allows the execution of all possible instructions as well as the inspection of microarchitectural states. Most importantly, TrustZone provides direct access to the cache state through privileged debug instructions. Having said that, Plumber may also leverage other techniques to infer the microarchitectural state (e.g., Reload+Time typically used in cache side-channel attacks [34, 53] to infer the content of the cache) if such special debug instructions are unavailable.

We conduct our experiments as bare-metal code—there are no background processes or interrupts which could induce noise in terms of cache content or timing. We still experience a low amount of noise due to the shared memory subsystem (such as the GPU) and because our experiments are not synchronized with the memory controller. We found that this noise could be safely ignored.

6 APPLICATION

We show the utility of LTs and the effectiveness of Plumber through three case studies: cache eviction policy, previction and prefetching. For each case the presented LT is only a fragment of the identified LT. Missing cases are omitted due to either (1) clarity, i.e., complex relations are omitted, or (2) inconclusive outcome, i.e., cases that generate random behavior and may not be reliably exploited. Experiments are done on Raspberry Pi 3 & 4— widely used ARMv8 platforms, which use Cortex-A53 and -A72 CPUs, respectively. On these processors, data is transferred between memory and cache in blocks of 64 bytes and the L1 data cache is 4-way set associative.

Cortex-A53 cores show the previction behavior, and although the CPU is well-documented, the *exact* behavior of the prefetcher on these cores is unknown. For example, the ARM manuals do not answer any of the following questions: (**Q1**) How much data is prefetched? (**Q2**) Do non-memory operations influence the prefetching behavior? And (**Q3**), do load operations in one page affect prefetching due to memory loads from a different page?

To perform the experiments, we used a cluster of five Raspberry Pi 3 and one Pi 4 boards. Table 2 shows the approximate total execution time to perform each experiment. Note that each experiment is a one-time effort required once per side channel and architecture.

6.1 Case Study: Eviction

To best illustrate the utility of LTs, we use Plumber to analyze the ARM Cortex-A72 cache eviction behavior. For experiment generation we leverage the parameterized eviction strategy from Rowhammer.js [13] for x86 architectures and adopted in ARMageddon [22] for ARM CPUs. The eviction strategy is shown in Algorithm 1.

Algorithm 1: Parameterized eviction strategy

```
1 for ($s = 0; s \leq S - D; s$ += $L$) do
2     for ($c = 0; c \leq C; c$ += 1) do
3         for ($d = 0; d \leq D; d$ += 1) do
4             $*a[s + d]$;
```

For this experiment, we created the following GTS which corresponds to the parameterized eviction strategy:

$$\left|\mathsf{P}\left(\mathsf{M}_{t_2,s_1}\ \left[\left[\left[\mathsf{M}_{t_1,s_1}\right]\mathsf{M}_t, \mathsf{D}, 1\right]\mathsf{C}\right]\mathsf{M}_t, \mathsf{S}, \mathsf{L}\right)\mathsf{M}_{t_2,s_1}\right|1000$$

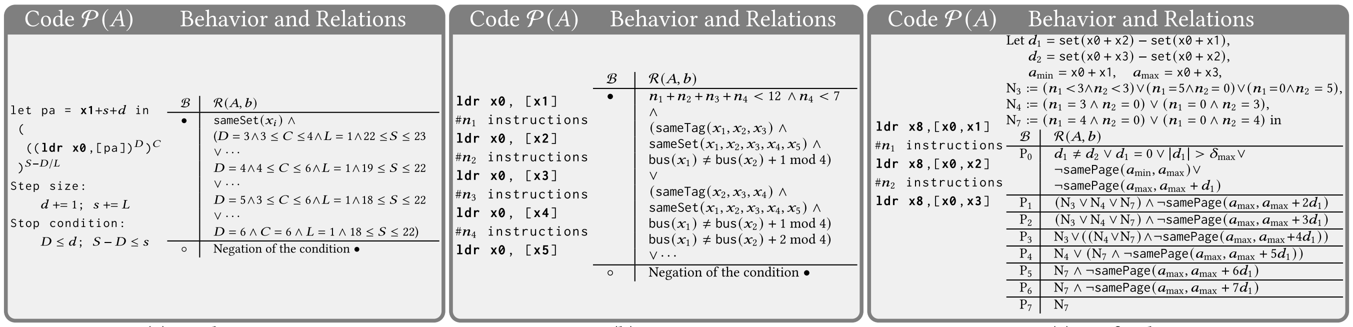
where each experiment is repeated 1000 times. The results were classified based on the existence of the pre-loaded address, indicating whether eviction occurred. Based on the results, the LT in Fig. 6.a was generated. The generated LT has an error tolerance of 5%, i.e., testcases that lead to eviction in more than 95% of the times where classified as triggering (●) behaviors.

The eviction test characterized by the parameterized eviction strategy is not a typical use case for Plumber. However, it is suitable for illustrating LTs. This test also demonstrates how Plumber facilitates reverse engineering tasks as will be shown in § 10.

6.2 Case Study: Previction

We used Scam-V [28] to generate a set of leakage witnesses, i.e., programs that may cause previction on the ARM Cortex-A53 (see § 2.1). All the generated examples were formed of exactly five load instructions. Moreover, looking at the cache content, it was evident that all leakage witnesses loaded three different tags to the same cache set (and optionally a fourth tag to a different set), i.e., two to three loads targeted the same cache line. We exploit this knowledge to construct initial GTSes. We use these GTSes in five experiments (E1–E5) to iteratively refine an LT for previction:

E1: Minimal Code. First, we wanted to check whether the obtained leakage witnesses contain minimal programs that trigger previction. We created a GTS which generates testcases containing all possible subsets (without repetition) of all instructions of each leakage witness. The GTS further mutated the word offsets for each of the loaded addresses. None of the generated testcases triggered previction. Thus, the leakage witnesses contained minimal previction programs.



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$(x_2) + 1 \bmod 4)$ $\vee$ (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$(x_2) + 1 \bmod 4)$ bus$(x_1) \neq$ bus$(x_2) + 2 \bmod 4)$ $\vee \cdots$
○	Negation of the condition ●

(b) Prevction

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,
$\mathrm{N}_3 := (n_1 < 3 \wedge n_2 < 3) \vee (n_1 = 5 \wedge n_2 = 0) \vee (n_1 = 0 \wedge n_2 = 5)$,
$\mathrm{N}_4 := (n_1 = 3 \wedge n_2 = 0) \vee (n_1 = 0 \wedge n_2 = 3)$,
$\mathrm{N}_7 := (n_1 = 4 \wedge n_2 = 0) \vee (n_1 = 0 \wedge n_2 = 4)$ in

$\mathcal{B}$	$\mathcal{R}(A,b)$
P_0	$d_1 \neq d_2 \vee d_1 = 0 \vee \lvert d_1 \rvert > \delta_{\max} \vee$ ¬samePage$(a_{\min}, a_{\max}) \vee$ ¬samePage$(a_{\max}, a_{\max} + d_1)$
P_1	$(\mathrm{N}_3 \vee \mathrm{N}_4 \vee \mathrm{N}_7) \wedge$ ¬samePage$(a_{\max}, a_{\max} + 2d_1)$
P_2	$(\mathrm{N}_3 \vee \mathrm{N}_4 \vee \mathrm{N}_7) \wedge$ ¬samePage$(a_{\max}, a_{\max} + 3d_1)$
P_3	$\mathrm{N}_3 \vee ((\mathrm{N}_4 \vee \mathrm{N}_7) \wedge$ ¬samePage$(a_{\max}, a_{\max} + 4d_1))$
P_4	$\mathrm{N}_4 \vee (\mathrm{N}_7 \wedge$ ¬samePage$(a_{\max}, a_{\max} + 5d_1))$
P_5	$\mathrm{N}_7 \wedge$ ¬samePage$(a_{\max}, a_{\max} + 6d_1)$
P_6	$\mathrm{N}_7 \wedge$ ¬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, $\cdots$, $\underline{2}$-$\underline{3}$-$\underline{1}$-4-5, $\cdots$, $\underline{3}$-4-$\underline{1}$-5-$\underline{2}$, $\cdots$

Performing **E1** requires doing 4 sets of experiments, each consisting of $5!/(5-x)! \times 16^x$ testcases, where $1 \le x \le 4$. Table 2 shows the total execution time to perform this experiment.

E2: Order of Instructions. Next, we checked whether the order of instructions affects prevction. We created GTSes which generate testcases containing all possible permutations of instructions in distinct leakage witnesses, e.g., $(\mathsf{M}_{t_1,s_1}\ \mathsf{M}_{t_1,s_1}\ \mathsf{M}_{t_1,s_1}\ \mathsf{M}_{t_2,s_1}\ \mathsf{M}_{t_3,s_1})!$. In this GTS, the first three loads (denoted as $\underline{1}$, $\underline{2}$ and $\underline{3}$) target the same cache line, but at different offsets. The outcome of the permutation is shown in Table 3. PR (first row) denotes prevction and nPR (second row) denotes that no prevction occurred. We draw three conclusions from this test:

- *Relation on tags:* The relation between the location of tags affects prevction. Only programs with three consecutive load instructions with the same tag cause prevction.
- *Tag value:* The exact tag value and location of the three consecutive loads does not matter, and can be similarly arbitrary for the non-consecutive load instructions.
- *Word offset:* Prevction behavior differs based on the byte (word) offset of the loaded addresses. In our example, multiple programs with the same order of cache sets and tags are in both PR and nPR, e.g., 2-1-3-5-4 (PR) and 2-3-1-5-4 (nPR). These permutations have instructions 1 and 3 swapped, i.e., two load instructions that only differ in their word offset.

E3: Tags and Sets. We also checked the effect of *exact tag and set values* on prevction. We created GTSes which generate testcases that preserve the relations between tags and/or sets of loaded addresses, while randomly changing these addresses:

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

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 prevction. 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 cache sets) while mutating the word offset. On Cortex-A53, each cache line (64 bytes) is divided into four disjoint "buses" of 16 bytes (i.e. a cache line is loaded in 4 bus rounds). For example, for three tag-identical loads in a sequence of five set-identical loads, prevction occurs if the bus of the first load is not the direct successor to the bus targeted by the second load.

This experiment took approximately 1.5 days to complete. The most time consuming part of this experiment is the case number 12 in Table 4 which consists of 12×16^5 testcases.

E5: Priming the Cache. We further checked if prevction also affects data cached before the execution of a testcase. We created GTSes which generate and execute prevction testcases while 1-4 lines of a cache set are occupied:

(1) $\mathsf{P}(\mathsf{M}_{t_4,s_1})\left[\mathsf{M}_{t_1,s_1}\right]3\ \mathsf{M}_{t_2,s_1}\ \ \mathsf{M}_{t_3,s_1}$
(2) $\mathsf{P}(\mathsf{M}_{t_4,s_1}\ \ \mathsf{M}_{t_5,s_1})\left[\mathsf{M}_{t_1,s_1}\right]3\ \mathsf{M}_{t_2,s_1}\ \ \mathsf{M}_{t_3,s_1}$
(3) $\mathsf{P}(\mathsf{M}_{t_4,s_1}\ \ \mathsf{M}_{t_5,s_1}\ \ \mathsf{M}_{t_6,s_1})\left[\mathsf{M}_{t_1,s_1}\right]3\ \mathsf{M}_{t_2,s_1}\ \ \mathsf{M}_{t_3,s_1}$
(4) $\mathsf{P}(\mathsf{M}_{t_4,s_1}\ \ \mathsf{M}_{t_5,s_1}\ \ \mathsf{M}_{t_6,s_1}\ \ \mathsf{M}_{t_7,s_1})\left[\mathsf{M}_{t_1,s_1}\right]3\ \mathsf{M}_{t_2,s_1}\ \ \mathsf{M}_{t_3,s_1}$

The results show that when the targeted set is half full (the second GTS), one of the two preloaded cache lines is evicted. As we will show, this insight may result in a side-channel attack (see § 8).

Prevction Leakage Template. Our experiments resulted in LTs that allow us to identify prevction side channels in applications. Fig. 6.b illustrates an example LT. Each of the experiments substantially refined this template. **E1** dictated the general structure of five loads; **E2** contributed to the set and tag affinity; **E3** showed that we do not need to constrain certain tag/set values; **E4** revealed the bus relationship; and **E5** gave auxiliary information

Table 4: Example GTSes used as input to Plumber

	Requests	description
(1)	$\langle[\mathtt{M}]5\rangle@$	1 Tag & 1 Set
(2)	$\langle[\mathtt{M}_{t_1,s_1}]4\ \ \mathtt{M}_{t_2,s_1}\rangle@$	2 Tags & 1 Set
(3)	$\langle[\mathtt{M}_{t_1,s_1}]3\ \ [\mathtt{M}_{t_2,s_1}]2\rangle@$	
	. . .	
(12)	$\langle\mathtt{M}_{t_1,s_1}\ \ \mathtt{M}_{t_2,s_1}\ \ \mathtt{M}_{t_3,s_1}\ \ \mathtt{M}_{t_4,s_1}\ \ \mathtt{M}_{t_5,s_2}\rangle@$	5 Tags & 2 Sets

about previction behavior when caches are primed. **E7** explains how we derived the wildcard instructions between the loads.

6.3 Case Study: Prefetching

Based on the ARM Cortex-A53's reference manual, prefetching could leak the (1) number of loads, (2) relation between loaded addresses (a.k.a. *stride*), (3) cache miss occurrences, and (4) end of a page. These characteristics are not sensitive, as they can be extracted from the cache content even in the absence of prefetching. We aim at validating the documented behavior, and also examine whether the undocumented behavior could leak sensitive information. To this end, we again design five experiments (E6–E10).

E6: Prerequisites for Prefetching. First, we wanted to devise necessary conditions for prefetching and determine the number of prefetched cache lines (**Q1**). We created GTSes which generate testcases for all possible programs consisting of (three to five)[1] loads while mutating their set index:

$\langle[\mathtt{M}_{t_1,s_1}]3\rangle\$$, $\langle[\mathtt{M}_{t_1,s_1}]4\rangle\$$, $\langle[\mathtt{M}_{t_1,s_1}]5\rangle\$$.

The main outcome of this test is an LT describing the relations between loaded cache lines and the number of prefetched addresses. Consider the GTS $\mathtt{M}_{t_1,s_1}\ \ \mathtt{M}_{t_1,s_2}\ \ \mathtt{M}_{t_1,s_3}$. Prefetching occurs when $s_3 - s_2 = s_2 - s_1 \leq \delta_{max}$, where $\delta_{max} = 4$ denotes the maximum stride. Moreover, programs with 3–4 consecutive loads trigger the prefetching of 3 additional cache lines, while streams with 5 loads lead to 4 prefetched addresses.

For prefetching, **E6** is the most time consuming case, as it consists of 3 sets of experiment each containing $2^{(7\times x)}$ testcases of x-loads programs, where $x \in \{3, 4, 5\}$. For the $x = 3$ case we have done the full experiment but for the two other cases we have fixed a few bits (1 and 2 bits resp.) of the set indices to make the experiment manageable. Overall this experiment took 4 days to complete: 0.2d for $x = 3$, 1d for $x = 4$ and 2.8d for $x = 5$.

E7: Intermediate Instructions. Next, we wanted to check the effect of intervening instructions on prefetching (**Q2**). For this, we have created GTSes which generate testcases containing programs with a fixed stride and a varying number of intermediate arithmetic instructions. For $0 < n \leq 100$ and $0 < m \leq 30$ we created:

(1) $\mathtt{M}_{t_1,s_1}\ \ [\mathtt{A}]\mathtt{n}\ \ \mathtt{M}_{t_1,s_1+1}\ \ \mathtt{M}_{t_2,s_1+2}\ \ \mathtt{M}_{t_1,s_1+3}$
(2) $\mathtt{M}_{t_1,s_1}\ \ \mathtt{M}_{t_1,s_1+1}\ \ [\mathtt{A}]\mathtt{n}\ \ \mathtt{M}_{t_2,s_1+2}\ \ \mathtt{M}_{t_1,s_1+3}$
(3) $\mathtt{M}_{t_1,s_1}\ \ \mathtt{M}_{t_1,s_1+1}\ \ \mathtt{M}_{t_2,s_1+2}\ \ [\mathtt{A}]\mathtt{n}\ \ \mathtt{M}_{t_1,s_1+3}$
(4) $\mathtt{M}_{t_1,s_1}\ \ [\mathtt{A}]\mathtt{m}\ \ \mathtt{M}_{t_1,s_1+1}\ \ [\mathtt{A}]\mathtt{m}\ \ \mathtt{M}_{t_2,s_1+2}\ \ [\mathtt{A}]\mathtt{m}\ \ \mathtt{M}_{t_1,s_1+3}$

We created similar GTSes for 3- and 6-load streams.

Our results showed that adding instructions between consecutive loads could alter the number of prefetched addresses, e.g., adding 3 arithmetic instructions between two consecutive loads can increase the number of prefetched addresses from 3 to 4. Similarly, adding 4 arithmetic instructions increases this number to 7 and adding 5 instructions reduces it again to 3. Thus, the prefetcher may leak the control flow at the granularity of one instruction, a new insight which may lead to potential side channels (see § 8).

E8: Respecting Page Boundary. We also checked whether prefetching respects page boundaries (as stated in the manual), i.e., if the processor prefetches addresses past the end of a page (**Q3**). To this end, we created GTSes to generate testcases containing programs with fixed strides while gradually shifting the loaded addresses toward the next page, i.e., up to one page ($64 * 64 = 4096$). For $0 < n \leq 5$ we created:

(1) $(\mathtt{M}_{t_1,s_1}\ \ \mathtt{M}_{t_1,s_1+n}\ \ \mathtt{M}_{t_1,s_1+2n})64$
(2) $(\mathtt{M}_{t_1,s_1}\ \ \mathtt{M}_{t_1,s_1+n}\ \ \mathtt{M}_{t_1,s_1+2n}\ \ \mathtt{M}_{t_1,s_1+3n})64$
(3) $(\mathtt{M}_{t_1,s_1}\ \ \mathtt{M}_{t_1,s_1+n}\ \ \mathtt{M}_{t_1,s_1+2n}\ \ \mathtt{M}_{t_1,s_1+3n}\ \ \mathtt{M}_{t_1,s_1+4n})64$

The results show that testcases at the end of the page had fewer prefetched cache lines as not to cross page boundary. Testcases with loads spread across different pages did not cause prefetching.

E9: Multiple Prefetching Sequences. We now explore how the prefetcher handles multiple, possibly interleaving sequences. To this end, we specify a GTS that merges three 3-load sequences with distinct tags and sets, i.e., from different memory pages.

$$\left(\begin{array}{c} (\mathtt{M}_{t_1,s_1}\ \mathtt{M}_{t_1,s_1+1}\ \mathtt{M}_{t_1,s_1+2} : \mathtt{M}_{t_2,s_2}\ \ \mathtt{M}_{t_2,s_2+1}\ \ \mathtt{M}_{t_2,s_2+2})+ \\ : \mathtt{M}_{t_3,s_3}\ \ \mathtt{M}_{t_3,s_3+1}\ \ \mathtt{M}_{t_3,s_3+2} \end{array}\right)+$$

Our results show that the prefetcher becomes active for only the first two sequences; any additional sequence will *not* be prefetched. To decide which one is first, the prefetcher picks the first sequences of three consecutive loads (two strides). This means that multiple independent sequences can cause interference, even if they are on different pages (**Q3**). Again, this novel observation can lead to potential side channels to leak information (§ 8).

E10: Cache Hits. Finally, we tested the influence of cache hits on prefetching. We created GTSes, which generated and executed prefetching testcases while one of the loaded addresses is cached:

(1) $\mathtt{P}(\mathtt{M}_{t_1,s_1})\langle\mathtt{M}_{t_1,s_1}\ \ \mathtt{M}_{t_1,s_1+1}\ \ \mathtt{M}_{t_1,s_1+2}\rangle\$$
(2) $\mathtt{P}(\mathtt{M}_{t_2,s_2})\mathtt{M}_{t_1,s_1}\ \ \mathtt{M}_{t_2,s_2}\ \ \mathtt{M}_{t_1,s_1+1}\ \ \mathtt{M}_{t_1,s_1+2}$

The GTS (1) did not trigger prefetching for all generated testcases, while the GTS (2) induced a behavior similar to that of $\mathtt{M}_{t_1,s_1}$ $\mathtt{M}_{t_1,s_1+1}\ \ \mathtt{M}_{t_1,s_1+2}$. Thus, the prefetcher only monitors cache misses, i.e., preloaded data may destroy sequences that would have otherwise been prefetched. This is problematic if prefetched sequences from different pages (and contexts) interfere with each other.

Prefetching Leakage Template. Fig. 6.c illustrates an LT for a 3-load stream, allowing to identify prefetching side channels in applications. To construct this LT: **E6** identified constraints on the cache sets; **E7** derived bounds on intermediate instructions and their effect on the prefetching behavior; **E8** refined constraints on cache sets based on page boundaries; **E9** revealed the interference between interleaving prefetching sequences; and **E10** gave auxiliary information about the effect of cache hits on prefetching.

7 MATCHING PREFETCHING LT IN BINARIES

We now demonstrate how an LT can be used to identify an instance of the side channel it describes in a target binary.

Target Binary and Side Channel. As a proof of concept, we re-identify a known prefetching-based side channel in OpenSSL 1.1.0g. It was first found and exploited on an Intel CPU by Shin

[1]We chose this range as 1–2 loads do not trigger prefetching, and more than 5 loads would create too many testcases.

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.

Performing Static Analysis. We use *asmregex* [5] to statically scan the target binary for the code pattern $\mathcal{P}(A)$ of the prefetching LT. Asmregex searches binaries for code sections that match a specified pattern. We extended the tool by approx. 200 LoC (code available at [36]) to support a subset of the ARM instruction set and added support for backreferences to the pattern language. Backreferences allow to express simple relations between instructions. For instance, two subsequent load instructions can be required to use the same base address register. To identify code sections matching $\mathcal{P}(A)$ in OpenSSL, we convert $\mathcal{P}(A)$ into the asmregex pattern shown in Appendix A. This pattern matches 429 3-load sequences across 18 OpenSSL modules. By briefly inspecting the matching candidate sections, we identify accesses to lookup tables in 11 of these modules. The remaining matches are predominantly caused by operations on complex data structures. Most importantly, we identify the code section exploited in [40] among the candidates in the module `crypto/bn/bn_gf2m.o`. Further investigation of other matches is considered out of scope.

Performing Dynamic Analysis. We proceed with the dynamic analysis step to check a candidate code section for input-dependent behavior using the relations $\mathcal{R}(A, b)$ from the LT. We create a simple wrapper program that calls the matching library function with varying input values. This program can be used to log all (input-dependent) loads from the relevant lookup table `SQR_tb`, which spans across three cache lines in memory. We record two different traces for each input value. First, we use Valgrind [29] and GDB [10] to record an *access trace*, a list of all loads from `SQR_tb` during program execution. This trace can be used to determine the *expected* 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.

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

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 range we tested when the LT was created. We denote these cases as *undecidable* cases. We note that no misclassifications occurred.

Conclusion. We successfully demonstrated that the prefetcher of the Cortex-A53 CPU shows input-dependent behavior for the library function under investigation. This is the base requirement for the differential attack in [40]. The LT helped us to re-identify this vulnerability known from the Intel architecture in ARM binary code. In contrast to prior work, our starting point was the whole OpenSSL code base. For code sections that closely match the LT (i.e., they closely correspond to code and relations that Plumber encountered during creation of the LT), the behavior classification based on the relations is accurate. When unknown relations occur, undecidable cases are more likely to appear. In our example, undecidable cases occur due to higher values of n_1 and n_2 than we used when creating the LT (to keep the number of test cases within a reasonable range). However, these cases can be detected and the analyst may use them to design further experiments in order to refine the LT in a targeted manner. This highlights again that a LT, which can hardly be ever complete, can be developed in an iterative fashion.

8 NOVEL LEAKAGE PRIMITIVES

Our experiments in § 6 also helped us to identify five novel prefetching based leakage primitives. For four of these primitives, we present a minimal code example and evaluation results of its leakage speed

[2]As we found in § 6.3-**E10** that the prefetcher only operates on cache misses, the load instructions relevant to the prefetcher are not necessarily the first three load instructions in the matching code section. Therefore, we perform our analysis based on the first three loads in each access trace that target different cache lines.

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 %

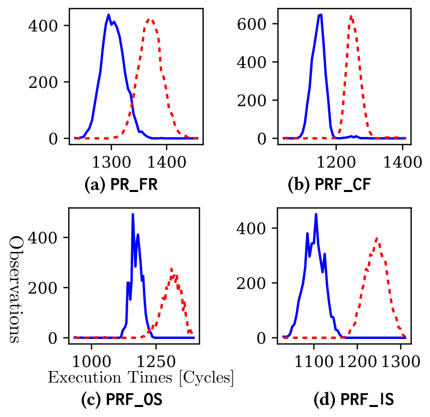


Figure 7: Histograms showing the execution time of the channel w.r.t. behavior (solid blue line) or not (dashed red line).

and error rate (see Table 6 and Fig. 7). We omit the evaluation of PR_PP as it is not applicable in our covert channel setup.

8.1 PrevICTION w/ Shared Memory (PR_FR)

Our previction-based Flush+Reload primitive PR_FR is based on our insights from **E4** in § 6.2. Unlike traditional Flush+Reload primitives, PR_FR allows leaking information based on a *bus* rather than cache lines. The idea is to make the (observable) occurrence of previction dependent on a secret bit (the leak target) by changing bus relations.

The primitive PR_FR in Fig. 8 leverages the strong bus dependency between the consecutive load instructions in a valid previction sequence. Let the lines 8–12 be a valid previction sequence following our LT in Fig. 6.b, i.e., x1–x3 being consecutive loads with a "valid" bus relation, and x4/x5 arbitrary other loads from the same set. The idea is to use a secret-dependent conditional change to the byte offset of the first load (x1) to destroy previction (lines 1 through 6). That is, the word offset of the first address loaded (in x1) depends on the value of the least significant bit of the data stored in x20. According to the LT, the relation between the offsets in the addresses in x1 and x2 determines if x1 will be previcted from the cache. Thus, by measuring the time required to load from the address stored in x1, the receiver can learn whether this address resides in the cache and consequently leak the secret bit.

8.2 Previction w/o Shared Memory (PR_PP)

Based on experiment **E5** in § 6.2, previction may target preloaded memory addresses and leak information in the absence of shared memory, e.g., through Prime+Probe. The sender code of our previction-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 leaking accesses to non-shared memory through shared pages.

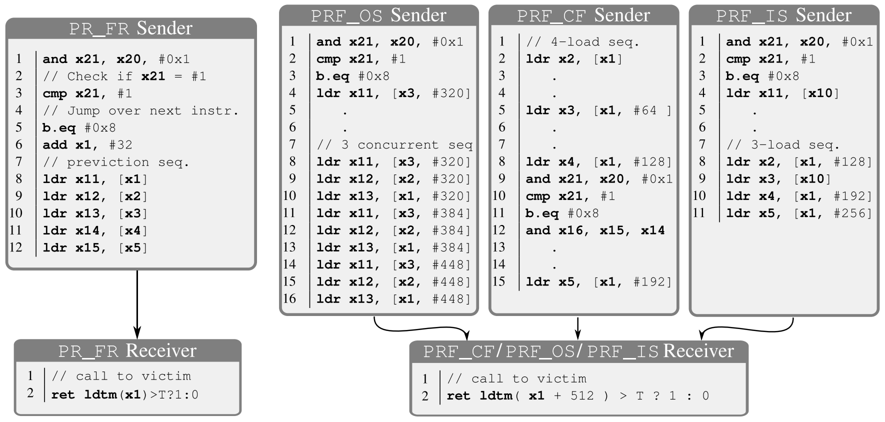
As shown in Fig. 8, the sender code contains a 3-load prefetching sequence with a fixed stride and an interleaving load from a different page, i.e., [x10] (lines 8 through 11). Since the processor ignores cache hits, the number of prefetched lines will depend on whether x10 is cached. Consequently, by measuring the time required to load from x1+512, the receiver can determine whether x10 has been cached and consequently learn the secret bit.

In contrast to the prefetching experiment in § 6.3, PRF_IS checks how the prefetcher's behavior changes when it *observes* interleaved loads from different pages (i.e., across page boundaries). Note that in PRF_IS all *predicted* addresses are within bounds of the same page (in accordance with **E7**).

8.5 Prefetching and Outstanding Seq. (PRF_OS)

PRF_OS exploits competing prefetching sequences to leak accesses to non-shared memory through shared pages. In other words, it allows leaking secrets through Flush+Reload even when secret-dependent memory accesses are from non-shared memory. PRF_OS is based on the outcome of experiments **E9** and **E10** in § 6.3.

As shown in Fig. 8, the sender has three interleaving 3-load streams with fixed strides (lines 8, 11, 14; lines 9, 12, 15; and lines 10, 13, 16). These streams are preceded by a load from the address x3, whose execution depends on one bit of a secret stored in x20 (lines 1 through 4). According to **E9**, memory addresses are prefetched for sequences whose strides are detected first. Additionally, according to **E9**, the processor ignores cache hits when detecting prefetching sequences. Consequently, depending on whether x3 is cached,



PR_FR Sender

```
and x21, x20, #0x1
// Check if x21 = #1
cmp x21, #1
// Jump over next instr.
b.eq #0x8
add x1, #32
// previction seq.
ldr x11, [x1]
ldr x12, [x2]
ldr x13, [x3]
ldr x14, [x4]
ldr x15, [x5]
```

PRF_OS Sender

```
and x21, x20, #0x1
cmp x21, #1
b.eq #0x8
ldr x11, [x3, #320]
    .
    .
// 3 concurrent seq
ldr x11, [x3, #320]
ldr x12, [x2, #320]
ldr x13, [x1, #320]
ldr x11, [x3, #384]
ldr x12, [x2, #384]
ldr x13, [x1, #384]
ldr x11, [x3, #448]
ldr x12, [x2, #448]
ldr x13, [x1, #448]
```

PRF_CF Sender

```
// 4-load seq.
ldr x2, [x1]
    .
    .
ldr x3, [x1, #64 ]
    .
    .
ldr x4, [x1, #128]
and x21, x20, #0x1
cmp x21, #1
b.eq #0x8
and x16, x15, x14
    .
    .
ldr x5, [x1, #192]
```

PRF_IS Sender

```
and x21, x20, #0x1
cmp x21, #1
b.eq #0x8
ldr x11, [x10]
    .
    .
// 3-load seq.
ldr x2, [x1, #128]
ldr x3, [x10]
ldr x4, [x1, #192]
ldr x5, [x1, #256]
```

PR_FR Receiver

```
// call to victim
ret ldtm(x1)>T?1:0
```

PRF_CF/PRF_OS/PRF_IS Receiver

```
// call to victim
ret ldtm( x1 + 512 ) > T ? 1 : 0
```

Figure 8: Examples for leakage primitives. `ldtm` measures the ldr execution time for the given address, T is the threshold.

prefetching would be triggered for either the first or the second two prefetching sequences. By measuring the time required to load from x1+512, the receiver can determine whether prefetching occurred for the third stream and consequently learn the secret bit.

9 RELATED WORK

Reverse-engineering cache behavior. Prior work on reverse-engineering the cache behavior mainly focused on replacement policies. In particular, existing approaches aim at reverse-engineering known permutation-based replacement policies [1, 2], e.g., FIFO and PLRU, as well as new adaptive policies [7, 17]. Some approaches pursued ad-hoc means [1, 2], others relied on a novel register automata learning technique [7] or compared hardware output against software-simulated caches [41, 50]. However, these methods are either not practical [7], or not general and do not guarantee correctness [50]. We rather design a framework to better understand undocumented behavior of hardware features such as previction [28], eviction policies and the cache prefetcher and specify their leakage templates. Plumber can also be used to facilitate reverse engineering of microarchitectural components like the branch predictor.

Side-channel attacks. Over the past decades, researchers have devised various means to exfiltrate secrets from computing devices based on electromagnetic [9], power-based [20], and timing-based [14, 34, 45, 53] side channels. Timing-based side channels can be exploited purely in software, thus also remotely. Most timing attacks exploit timing differences introduced by processor caches. Cache-based attacks proposed in the literature include but are not limited to Prime+Probe [34, 44], Flush+Reload [53], Evict+Time [32], and their variants [14, 15]. Such attacks target implementations of cryptographic algorithms [14, 19, 21, 32], and more generic attack vectors such as keylogging [15]. Closely related to previction are CacheOut [48] and RIDL [47] which use cache eviction to leak secret information. Similarly, CacheBleed [54] exploits cache bank conflicts to break, resp., RSA and AES.

Plumber vs. Scam-V Plumber's goal is to facilitate understanding microarchitectural behavior, e.g., triggers and effects, via easily constructed queries. The approach of Scam-V [28] (and similar tools, e.g., [11, 51]) is complementary to that of Plumber. This relation is better expressed as a two-step approach: first, Scam-V or similar tools are used to detect a possible channel by specifying the monitored component, e.g., the cache state, which often is dictated by the vulnerability discovery tool (e.g., port contention in [11], or execution time in [51]). Whenever the tool discovers yet-unsupported types of side channels on monitored components, Plumber can be used to learn the correlation between (attacker-controlled) inputs and the channel (see Fig. 3). The challenging aspect of such an integration is the generalization of concrete code examples, which can be provided by the analyst.

Scam-V finds a channel by executing randomly generated program-input pairs. Using an SMT solver to generate inputs requires careful engineering of the queries sent to the solver, otherwise generated inputs and thus counterexamples will be too similar which makes them not suitable for statistical analysis. Conversely, Plumber makes input generation more efficient by using GTS and allows learning the correlation between inputs and the channel.

10 DISCUSSION

We proceed to discuss the utility of Plumber in other use cases and its limitations. In particular, we show that Plumber can facilitate reverse engineering of microarchitectural features. We show this by applying Plumber to reverse engineer the structure of the Cortex-A53's branch predictor. In Appendix B we discuss the possibility of applying Plumber to reverse engineer cache slice mapping.

Reverse Engineering with Plumber: Branch Predictor. The only available information on the Cortex-A53 branch predictor unit is that it is a *global type* that uses branch history registers (BHRs) and a 3072-entry pattern history table (PHT) [8]. Each entry of the PHT contains a 2-bit saturating counter to predicate a branch outcome. The PHT is accessed via a BHR that stores the history of the recently executed branches (see Fig. 9.a). The goal is to find the size and the number of BHRs, the structure of the PHT, and the mechanism used to map branches to this table. Our experiment is inspired by the work of Uzelac et al. [46] to reverse engineer the branch predictor of Intel's Pentium M CPU.

Since the size of a pattern history table should be a power of 2 [33], we assumed that Cortex-A53's branch predictor should be structured as three tables with 1024 entries, each connected to a unique BHR of size 10 bits. Therefore, to ensure that every branch

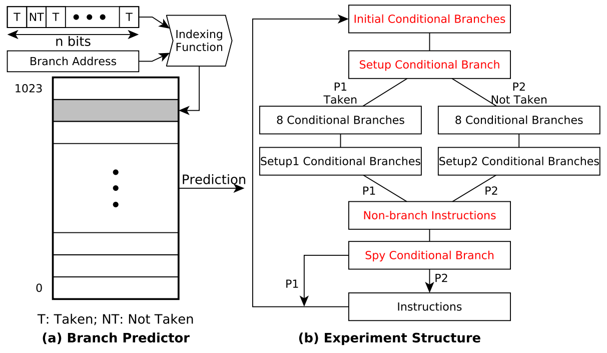


Figure 9: Branch predictor experiment.

instruction is mapped to the same BHR and PHT, we additionally assumed that the branch address modulo 3 is used to compute the index to access BHR and PHT. Fig. 9.b shows the general structure of the experiment (adapted from [46]) we have conducted to validate our conjectures. In this experiment the *initial conditional branches* are used to force the branch predictor to mispredict the *setup conditional branch. Non-conditional instructions* are dummy instructions (e.g. nop) used to control the distance between the setup 1/2 conditional branches and our *spy conditional branch.* To ensure that all branches will be mapped to the same BHR and PHT we pad the distances between all effective branch instructions with nops, i.e., making their address congruent modulo 3.

To determine the size of the BHR and PHT we use a spy branch with controllable outcome pattern. The branch is always taken if reached through path $P1$ and not taken when reached through $P2$. Thus, when the number of BHR entries for P1 and P2 are equal, the spy branch will be mispredicted. Misprediction also happens if the spy branch's history is not present in the PHT. Since the spy branch is not the only mispredicted branch, we observe (utilizing the *performance monitor unit* (PMU)) ≈100% misprediction rate when the spy branch is mispredicted and ≈60% otherwise.

Uzelac et al. [46] proposed an experiment to find the branch address bits which affect the BHR (when PHT is not full). In their experiment the observed misprediction rate is also either 60% or 100%. Based on this experiment, we utilize Plumber to discover the PHT size for Cortex-A53. To do so, we generate a GTS to fuzz the number of initial conditional branches and the distance between the spy branch and the setup 1 and 2 branches. The GTS is formed of the following blocks (see Fig. 9.b):

$\left[\mathsf{B}_{c_1,T,12}\ \ \mathsf{S}_{c_2,T}\ \ \mathsf{N}\right]X$ (Initial conditional branches)
$\mathsf{S}_{c_1,bool}\ \ \mathsf{B}_{c_1,F,53}$ (Setup conditional branch)
$\left[\mathsf{B}_{c_2,T,12}\ \ \mathsf{S}_{c_2,T}\ \ \mathsf{N}\right]8\ \ \mathsf{B}_{c_2,T,12}\ \ \mathsf{S}_{c_2,T}$ (Setup 1)
$\left[\mathsf{B}_{c_2,F,12}\ \ \mathsf{S}_{c_2,T}\ \ \mathsf{N}\right]8\ \ \mathsf{B}_{c_2,F,12}\ \ \mathsf{S}_{c_2,T}$ (Setup 2)
$[\mathsf{N}]Y$ (Non-branch instructions)
$\mathsf{B}_{c_1,T,12}\ \ \mathsf{S}_{c_2,T}$ (Spy conditional branch)

For each tested combination of X and Y, the concatenation of the above blocks is executed 10,240 times, while alternating the value of bool every 16 executions. This is done by using the power macro as follows: $[[\mathsf{bool} = \mathsf{T}]16\ \ [\mathsf{bool} = \mathsf{F}]16]40$.

For any number X of initial conditional branches less than 1024 we got (almost) the same result as Uzelac (i.e. a misprediction rate between 60% and 100%). However, when the PHT is full, the misprediction rate is always 100%. The results of these experiments support our conjecture on the size of PHT and BHR.

Plumber's limitations. Given the complexity of the microarchitectural components and the number of side-channel attacks, the current implementation of Plumber mainly targets cache-based side channels and the implementation of some components is limited to ARM (ARM-v7 and -v8) and RISC-V architectures. However, Plumber's design is generic and not constrained to cache-related channels and its implementation can be ported to other architectures such as x86. The main challenge of such an adaptation is porting Plumber's inspection module, which is currently deployed in ARM TrustZone. Moving to a new platform, e.g. Intel or AMD, would require replacing the inspection module by an alternative probing mechanism such as Flush+Reload [53] or Flush+Flush [14].

11 CONCLUDING REMARKS

We introduced the concept of Leakage Templates to abstractly describe specific side channels, and determine relations between input parameters that when satisfied can trigger specific microarchitectural behavior. LTs allow to automate identifying code snippets that are vulnerable to side-channel attacks in application binaries for certain target architectures. As such, they enable attackers and analysts to identify potential side channels in applications.

As details on microarchitectural aspects such as cache eviction policy, prefetching and previction are scarce, derivation of LTs is challenging. Expressive specifications for testcases are required, a large set of inputs has to be explored, and low noise measurement setups are essential. Also, techniques to automate discovery of relations between code, data, and leakage behavior are needed.

To address those challenges, we proposed Plumber, which leverages *instruction fuzzing*, *instructions' operand mutation* and *statistical analysis* to explore underspecified behavior of microarchitectural optimisations. Plumber's high-level goal is to *facilitate* the understanding of microarchitectural behavior. Therefore, we expect that the user has some prior knowledge regarding the existence of undocumented behavior or potential information leakage.

We showed the utility of templates produced by Plumber by identifying five novel side-channel attack primitives (for an ARM Cortex-A53 core) and used Plumber to reverse engineer the A53's branch predictor structure. Also, we showed how LTs can be used to identify an instance of the specified side channel in a target binary. We plan to extend Plumber further, e.g., to reverse engineer port contention behavior by setting the measured state to be the execution time of various types of instructions. A generated request would contain a number of instructions of the same type. Then, the analysis would only compare execution times and return the relation between executed and measured instructions. One can also use execution time measurements to extract the operand/runtime correlation of a multi-clock-cycle multiplier.

ACKNOWLEDGEMENT

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762).

REFERENCES

[1] Andreas Abel and Jan Reineke. Measurement-based modeling of the cache replacement policy. RTAS '13, page 65–74, USA, 2013. IEEE Computer Society.

[2] Andreas Abel and Jan Reineke. Reverse engineering of cache replacement policies in intel microprocessors and their evaluation. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 141–142, 2014.

[3] Onur Acıiçmez and Çetin Kaya Koç. Trace-driven Cache Attacks on AES (Short Paper). In *Proceedings of the 8th International Conference on Information and Communications Security*, ICICS, pages 112–121. Springer-Verlag, 2006.

[4] Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in intel processors. *IACR Cryptol. ePrint Arch.*, page 690, 2015.

[5] asmregex. https://github.com/Usibre/asmregex/.

[6] Sarani Bhattacharya, Chester Rebeiro, and Debdeep Mukhopadhyay. Hardware prefetchers leak: A revisit of SVF for cache-timing attacks. In *45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2012, Workshops Proceedings, Vancouver, BC, Canada, December 1-5, 2012*, pages 17–23, 2012.

[7] Guillem Rueda Cebollero. Learning cache replacement policies using register automata. 2013.

[8] ARM Cortex-A53 mpcore processor, technical reference manual. https://developer.arm.com/documentation/ddi0500/j/.

[9] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. CHES '01, page 251–261, Berlin, Heidelberg, 2001. Springer-Verlag.

[10] GDB developers. GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb.

[11] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. Absynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, 2020.

[12] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 368–379, New York, NY, USA, 2016. Association for Computing Machinery.

[13] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in javascript. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings*, pages 300–321, 2016.

[14] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912, Washington, D.C., August 2015. USENIX Association.

[16] R. Guanciale, H. Nemati, C. Baumann, and M. Dam. Cache storage channels: Alias-driven attacks and verified countermeasures. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 38–55, 2016.

[17] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). 38(3):60–71, June 2010.

[18] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[19] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, page 104–113, Berlin, Heidelberg, 1996. Springer-Verlag.

[20] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, page 388–397, Berlin, Heidelberg, 1999. Springer-Verlag.

[21] Nate Lawson. Side-channel attacks on cryptographic software. *IEEE Security & Privacy*, 7(6):65–68, 2009.

[22] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 549–564, 2016.

[23] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[24] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*, 2015.

[25] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-Cores Cache Covert Channel. In *DIMVA*, 2015.

[26] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS*, 2017.

[27] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural data leakage via automated attack synthesis. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1427–1444. USENIX Association, 2020.

[28] Hamed Nemati, Pablo Buiras, Andreas Lindner, Roberto Guanciale, and Swen Jacobs. Validation of abstract side-channel models for computer architectures. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, pages 225–248, Cham, 2020. Springer International Publishing.

[29] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, Jun 2007.

[30] Michael Neve and Jean-Pierre Seifert. Advances on access-driven cache attacks on AES. In *Proceedings of the 13th International Conference on Selected Areas in Cryptography*, SAC'06, pages 147–162. Springer-Verlag, 2007.

[31] Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. Diffuzz: Differential fuzzing for side-channel analysis. In *Software Engineering 2020, Fachtagung des GI-Fachbereichs Softwaretechnik, 24.-28. Februar 2020, Innsbruck, Austria*, pages 125–126, 2020.

[32] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, page 1–20, Berlin, Heidelberg, 2006. Springer-Verlag.

[33] Emre Özer, Alastair Reid, and Stuart Biles. Low-cost techniques for reducing branch context pollution in a soft realtime embedded multithreaded processor. In *19th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2007), 24-27 October 2007, Gramado, RS, Brazil*, pages 37–44, 2007.

[34] Colin Percival. Cache missing for fun and profit. In *In Proc. of BSDCan 2005*, 2005.

[35] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium*, 2016.

[36] Plumber. https://github.com/scy-phy/plumber/.

[37] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CrossTalk: Speculative data leaks across cores are real. In *S&P*, May 2021. Intel Bounty Reward.

[38] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *FC*, 2017.

[39] Benjamin Semal, Konstantinos Markantonakis, Keith Mayes, and Jan Kalbantner. One covert channel to rule them all: A practical approach to data exfiltration in the cloud. 2020.

[40] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. Unveiling hardware-based data prefetcher, a hidden source of information leakage. CCS '18, page 131–145, New York, NY, USA, 2018. Association for Computing Machinery.

[41] Armando Solar-Lezama. The sketching approach to program synthesis. In Zhenjiang Hu, editor, *Programming Languages and Systems*, pages 4–13, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[42] M. Caner Tol, Koray Yurtseven, Berk Gülmezoglu, and Berk Sunar. Fastspec: Scalable generation and detection of spectre gadgets using neural embeddings. *CoRR*, abs/2006.14147, 2020.

[43] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. Checkmate: Automated synthesis of hardware exploits and security litmus tests. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, page 947–960. IEEE Press, 2018.

[44] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *J. Cryptol.*, 23(2):37–71, January 2010.

[45] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, and Maki Shigeri. Cryptanalysis of DES implemented on computers with cache. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems*, CHES'03, LNCS, pages 62–76. Springer, 2003.

[46] Vladimir Uzelac and Aleksandar Milenkovic. Experiment flows and microbenchmarks for reverse engineering of branch predictor structures. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2009, April 26-28, 2009, Boston, Massachusetts, USA, Proceedings*, pages 207–217, 2009.

[47] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *S&P*, May 2019.

[48] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. Cacheout: Leaking data on intel cpus via cache evictions, 2020.

[49] Jose Rodrigo Sanchez Vicarte, Pradyumna Shome, Nandeeka Nayak, Caroline Trippel, Adam Morrison, David Kohlbrenner, and Christopher W Fletcher. Opening pandora's box: A systematic study of new ways microarchitecture can leak private data. 2021.

[50] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. Cachequery: Learning replacement policies from hardware caches. PLDI 2020, page 519–532, New York, NY, USA, 2020. Association for Computing Machinery.

```
1   [prefetch]
2   <ARMLD,AG,QR,RO,{1:1:*base_reg}>
3   <any,>{0,5}
4   <ARMLD,AG,QR,RO,{1:1:=base_reg}>
5   <any,>{0,5}
6   <ARMLD,AG,QR,RO,{1:1:=base_reg}>
```

Figure 10: An *asmregex* pattern to identify instances of the prefetching LT

[51] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. Osiris: Automated discovery of microarchitectural side channels. *CoRR*, abs/2106.03470, 2021.

[52] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *USENIX Security Symposium*, 2012.

[53] Yuval Yarom and Katrina Falkner. Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, pages 719–732, 2014.

[54] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time RSA. *J. Cryptogr. Eng.*, 7(2):99–112, 2017.

[55] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the Conference on Computer and Communications Security*, CCS, pages 305–316. ACM, 2012.

A ASMREGEX PATTERN FOR THE PREFECH LEAKAGE TEMPLATE

Fig. 10 shows the asmregex pattern that is used to identify instances of the prefetching LT in binaries. It specifies a sequence of three load instructions with a maximum distance of 5 other instructions between them.

Each of the load instructions (ARMLD) is further required to fulfill the following properties:

- Operand 0 (AG) is a general purpose register (e.g., `x0`).
- Operand 1 (QR) is a square bracket, followed by a general purpose register (e.g., `[x1`)
- Operand 2 (RO) is a general purpose register, optionally followed by a closing square bracket (e.g. `x2` or `x2]`)

Therefore, this pattern line matches both of the following instructions:

- `ldr x0, [x1, x2]`
- `ldr x0, [x1, x2, lsl#3]`

In addition, all three load instruction pattern lines contain a back-reference to constrain operand 1. In particular, the pattern requires that operand 1 is the same for all three load instructions. The back-reference is initialized in line 2 and is used as a constraint in lines 4 and 6. This constraint is added as a heuristic to ensure that all three loads load from addresses that are close to each other, which is a mandatory requirement to trigger prefetching.

B REVERSE ENGINEERING CACHE SLICE SELECTION

To improve the performance of the last-level cache, modern processors divide it into multiple slices that can be accessed in parallel by different cores. Intel CPUs use a deterministic hash function to select cache slices. A reverse engineering approach for cache slice selection of Intel CPUs based on Prime+Probe is suggested in [4].

The task is done in three steps: (1) m data blocks residing in the same slice are identified; (2) equations for slice mapping are generated; (3) the used hash function is recovered. In what follows we show how these steps can be done using Plumber. Since this task is similar to the eviction example from § 6.1, the classifier classifies the output based on the existence of data block(s) in memory. This can be achieved through probing and timing, or by simply inspecting the content of the cache.

Step 1. An address is pre-loaded followed by b loads to the same cache set causing eviction of this line. Then one of the loads is removed and the execution is repeated. If eviction no longer occurs, it indicates that the removed load address maps to the same slice as the pre-loaded address. This step is repeated until m data blocks are identified. This experiment can be done by running the GTS:

$$\left| \mathsf{P}\left(\mathsf{M}_{t_1,s_1}\ \left(\mathsf{M}_{t_x,s_1}\ \cdots\ \mathsf{M}_{t_{x+b},s_1}\right)\right) \mathsf{M}_{t_1,s_1} \right| 1000$$

where each experiment is repeated 1000 times.

Step 2. In this step more data blocks that reside in the same slice are generated. This is done similar to step one. The only difference is that the m generated data blocks are used to prime the cache (instead of one block). This step is repeated until a large number of memory blocks that map to the same slice is identified. This experiment is represented by the following GTS:

$$\left| \mathsf{P}\left(\mathsf{M}_{t_1,s_1}\ \cdots\ \mathsf{M}_{t_m,s_1}\ \left(\mathsf{M}_{t_x,s_1}\ \cdots\ \mathsf{M}_{t_{x+b},s_1}\right)\right) \mathsf{M}_{t_1,s_1}\ \cdots\ \mathsf{M}_{t_m,s_1} \right| 1000$$

where each experiment is repeated 1000 times. The found blocks are used to generate a system of equations/matrix. This can be implemented within Plumber's analyzer function.

Step 3. Based on the generated system of equations, the hash function for the slice mapping can be recovered. This is done by modelling this function as a concatenation of binary linear functions. These functions are then determined based on the matrix representing the system equations. This step can also be implemented in Plumber's analyzer function, given the individual bits of each physical address provided by the Classifier.