A Benchmark Suite for Evaluating Caches' Vulnerability to Timing Attacks

Shuwen Deng, Wenjie Xiong, Jakub Szefer
{shuwen.deng,wenjie.xiong,jakub.szefer}@yale.edu
Yale University

Abstract

Based on improvements to an existing three-step model for cache timing-based attacks, this work presents 88 *Strong* types of theoretical timing-based vulnerabilities in processor caches. It also presents and implements a new benchmark suite that can be used to test if processor cache is vulnerable to one of the attacks. In total, there are 1094 automatically-generated test programs which cover the 88 *Strong* theoretical vulnerabilities. The benchmark suite generates the Cache Timing Vulnerability Score (CTVS) which can be used to evaluate how vulnerable a specific cache implementation is to different attacks. A smaller CTVS means the design is more secure. Evaluation is conducted on commodity Intel and AMD processors and shows how the differences in processor implementations can result in different types of attacks that they are vulnerable to. Further, the benchmarks and the CTVS can be used in simulation to help designers of new secure processors and caches evaluate their designs' susceptibility to cache timing-based attacks.

CCS Concepts • **Security and privacy Embedded systems security**; *Side-channel analysis and countermeasures*.

Keywords security, timing attacks, caches, benchmark

ACM Reference Format:
Shuwen Deng, Wenjie Xiong, Jakub Szefer. 2020. A Benchmark Suite for Evaluating Caches' Vulnerability to Timing Attacks. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3373376.3378510

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378510

1 Introduction

Cache timing channels have long been used to deploy attacks that can reconstruct sensitive data, especially secrets such as cryptographic keys [2, 3, 35, 41]. They usually make use of the timing difference in the memory operations between observing a cache hit and a cache miss to derive the victim's secrets. Since 2018, the cache timing-based attacks have gained new attention due to their use in Spectre [24] and Meltdown [29] attacks.

With the long history of attacks, there are also many defenses proposed or deployed in software, e.g., [7, 25], and in hardware, e.g., [4, 6, 14, 21–23, 28, 30, 31, 37, 40, 43–45, 47, 49, 50, 54, 55]. However, existing defenses can usually only prevent a subset of the attacks. For example, RIC [21] cache has been shown to be able to defend the Prime+Probe type of attacks [35, 36], but is vulnerable to the Flush+Reload [53] type of attacks. More importantly, so far most researchers have focused on coming up with individual attacks (and defenses for them), and there has been limited work on understanding all possible types of attacks.

To address the need to understand and evaluate all the different possible types of attacks, this paper presents both a theoretical model of all possible timing-based attacks in caches, and a benchmark suite that can test for the theoretical vulnerabilities on real processors, or simulations of new designs. A key part of this work is an improved three-step model. Compared to our existing work [11], the new three-step model additionally considers: (1) differences in "local" and "remote" cores for making the timing observations and running the victim or the attacker code, (2) the victim and attacker running in hyper-threading or time-slicing, (3) using both read and write operations as memory accesses in the potential attacks (all but one prior work only considered reads), and (4) two types of cache line invalidation operations, through flush instruction, or using cache coherence by writing on "remote" core to invalidate "local" core's cache lines. The new three-step model shows 32 new types of timing-based vulnerabilities not considered before, which is in addition to the new attacks found in the original three-step model [11].

Based upon the new three-step model, we derive that there are in total 88 *Strong* type of vulnerabilities, including 32 new ones. For these vulnerabilities, we write

scripts to automatically generate a total of 1094 benchmarks, which consider different variants for each vulnerability, such as using reads vs. using writes. The benchmarks are then used to test commodity Intel and AMD machines (workstations and servers in lab and machines in Amazon's EC2 cloud). The benchmarks can also be run on simulators to evaluate new types of secure processor caches. The benchmarks are further used to generate a new Cache Timing Vulnerability Score (CTVS), which can help evaluate how different processors are vulnerable to different attacks – differences in the processor implementations mean that not all processors have the same vulnerabilities. Based on the CTVS, designers can have a better understanding of the vulnerabilities in their designs and can develop new defenses. The defenses can, for example, be customized to vulnerabilities that CTVS detects on the given processor, instead of a one-size-fits-all defense.

While the paper focuses on the 88 *Strong* types of vulnerabilities for the benchmark design, we also generated benchmarks for all 4913 possible three-step combinations in the three-step model. Based on the results, there are no more effective ones that are found apart from the 88 vulnerabilities.

1.1 Contributions

The contributions of this work are as follows:

- Improved modeling of processors' behavior to derive a set of 88 *Strong* timing-based cache vulnerabilities using the three-step model, including 32 new ones compared to our prior work [11].
- Development of the first automatically-generated benchmark suite that is used to evaluate processor caches' vulnerability to all possible timing attacks.
- Evaluation of the benchmarks on Intel and AMD processors in the lab and on Amazon's Elastic Compute Cloud (Amazon EC2) servers.
- Generation of *Cache Timing Vulnerability Score* for different processors to understand which attacks they are vulnerable to, and to be able to evaluate and customize defenses.
- Validation of the three-step model using the benchmarks, demonstrating no benchmarks beyond the ones corresponding to the 88 *Strong* types of vulnerabilities (in addition to *Weak* and repeat types) show vulnerabilities on the tested systems.

The benchmark related code used in this paper will be released under open-source license at https://caslab.csl.yale.edu/code/cache-security-benchmarks/.

2 Background

This section presents background on cache timing-based attacks, the existing three-step model, and existing metrics used to help understand vulnerabilities of caches to timing attacks.

2.1 Timing of Memory Operations and Caches

Processor caches are the key to helping processors maintain high performance when accessing data. However, the caches are of finite size, not all data can fit in them, and timing related to the memory operations involving caches, such as accesses resulting in cache hits and misses, can reveal information about the addresses or even data (for instruction caches it may be possible to reveal information about instructions as well). In general, two types of memory-related operations exhibit timing variations that can be abused for timing-based side or covert channel attacks in processor data caches. First, memory access operations, such as loads and stores can be fast (e.g., a cache hit) or slow (e.g., a cache miss). Second, invalidation-related operations, such as cache flush, can be fast (e.g., there is no dirty data in cache so flush finishes quickly) or slow (e.g., there is dirty data in the cache so it has to be written back, resulting in longer timing).

2.2 Timing-Based Attacks on Processor Caches

Researchers have proposed to use the timing differences in memory-related operations to attack software, e.g., [1–3, 17, 36]. Especially, the timing-based side-channel attacks often focus on cryptographic applications, e.g., attacks on software using AES encryption or decryption with table lookups [8]. Further, there are many timing-based covert-channel attacks, where the sender and receiver cooperate to leak data, e.g., there are cache covert channels focusing on the last-level cache mentioned in study [32] and cross-core cache covert channels [33]. And most recently, timing-based channels are used as a part of Spectre and Meltdown transient-execution attacks, e.g., [24, 27, 29, 39].

2.3 Previous Three-Step Model for Timing-Based Attacks in Caches

Our prior work [11] has presented a systematic approach to find all possible cache timing-based vulnerabilities. The model was established based on two observations: all existing cache timing attacks focusing on the data are within three memory operations, and timing attacks can be analyzed by checking the behavior of one cache block (since all blocks are updated in the same manner by the cache logic).

Table 1. The 17 possible states for a single cache block of three-step model we previously proposed [11].

State	Description
V_u	The cache block contains a specific memory location u brought in by the victim, which is at an address unknown to the attacker but within the set of sensitive memory locations x.
V_u^{inv}	The cache block state can be anything except u in the cache block. The data and its address u are "removed" from the cache block by the victim.
A_a or V_a	The cache block contains a specific memory location a, brought in by a memory access by the attacker, A_a, or the victim, V_a. The address a is within the range of sensitive locations x and known to the attacker.
$A_{a^{alias}}$ or $V_{a^{alias}}$	The cache block contains a memory address a^{alias}, brought in by a memory access by the attacker, $A_{a^{alias}}$, or the victim, $V_{a^{alias}}$. The address a^{alias} is within the range x and not the same as a, but it maps to the same cache block as a, i.e. it "aliases" to a. The address a^{alias} is known to the attacker.
A_d or V_d	The cache block contains a memory address d brought in by a memory access by the attacker, A_d, or the victim, V_d. The address d is not within the range x and known to the attacker.
A^{inv} or V^{inv}	The data and its address are "removed" from the cache block by the attacker, A^{inv}, or the victim, V^{inv}, as a result of cache block being invalidated.
A_a^{inv} or V_a^{inv}	The cache block state can be anything except a. The data and its address a are "removed" from the cache block by the attacker, A_a^{inv}, or the victim, V_a^{inv}.
$A_{a^{alias}}^{inv}$ or $V_{a^{alias}}^{inv}$	The cache block state can be anything except a^{alias}. The data and its address a^{alias} are "removed" from the cache block by the attacker, $A_{a^{alias}}^{inv}$, or the victim, $V_{a^{alias}}^{inv}$.
A_d^{inv} or V_d^{inv}	The cache block state can be anything except d. The data and its address d are "removed" from the cache block by the attacker, A_d^{inv}, or the victim, V_d^{inv}.
$\star$	Any data, or no data, can be in the cache block. The attacker has no knowledge of the memory address in this cache block.

Following these observations, our work [11] presented a three-step model focusing on one cache block for evaluating all possible timing-based attacks. Further, a soundness analysis of the three-step model was performed to show that three steps are sufficient to model all the timing-based attacks in caches. In the three-step model, each step represents the state of the cache line after a memory-related operation is performed. First, there is a initial step that sets the cache line into a known state. Second, there is a step that modifies the state of the cache line. Finally, there is the last step, based on the timing of which, the change in the state of the cache line is observed. Each of the steps can be performed by the attacker (A) or the victim (V). The goal of the prior study [11] was to find which three-step combinations can represent timing attacks from which the attacker learns information about the unknown address accessed by the victim. We listed 17 possible states for a cache line, shown in Table 1. Among these states, V represents that the state is a result of the victim's operation, while A represents that the state is a result of the attacker's operation. x denotes the set of virtual memory addresses storing addresses of sensitive data, and u denotes the victim's secret address within x which is unknown to the attacker. a, a^{alias} and d denote known memory addresses that map to the same cache line. d refers to an address outside of x, while the others are the address within x. The attacker's goal is to obtain u, which could be the same as a or a^{alias}, maps to the same set as a, a^{alias} and d, or not.

Given that there are 17 states and 3 steps, there are in total $17^3 = 4913$ possible combinations of three steps that can be derived. The prior analysis demonstrated that 72 of the three-step combinations are *Strong* effective vulnerabilities, where the attacker can obtain the value of the unknown address u unambiguously with fast or slow timing. Meanwhile, 64 patterns were shown to be *Weak* effective vulnerabilities, where there are timing differences according to different values of u, but no single timing corresponds to a unique possible value of u. Based on the analysis, the prior work [11] showed that 29 out of 72 vulnerabilities maped to existing cache attacks. The other 43 types were considered new without corresponding attacks in literature at that time.

Our prior work [11], however, was limited in how it modeled the caches. In particular, it did not consider: read vs. write accesses, invalidation using flush instruction vs. invalidation using cache coherence, multithreading, and multicore system, and the possibility that there are more than just one "fast" and one "slow" timing in memory-related operation. Also, the work did not present any code or benchmarks for realizing and testing for the possible types of vulnerabilities.

Meanwhile, this paper improves our model of caches, presents a set of 88 *Strong* vulnerability types, including 32 types not in prior work [11], implements benchmarks for testing commodity processors, and validates the theoretical analysis by running all possible three-step combinations to show no other types of timing differences (and thus vulnerabilities) exist.

2.4 Metrics for Vulnerabilities in Caches

A few existing papers have explored different types of metrics to try to understand security of caches. One study [56] leveraged mutual information to measure potential cache side-channel leakage. Another work [18] modeled cache interference using probabilistic information flow graph. However, both of the studies [18, 56] only examined limited attacks including Evict + Time attack [35], Cache Collision attack [3], Bernstein's attack [2], Prime + Probe attack [35, 36], and Flush + Reload attack [53]. In a separate work, an analytical model was proposed in study [13] to track the fraction

of the victim's critical items accessible in the cache to determine leakage. In a different work, SVF [10] metric measured information leakage by measuring the signal-to-noise ratio in an attacker's observations. Meanwhile, CSV [57] metric used direct correlation, in place of phase correlation used by SVF, to measure leakage. The analytical model [13], SVF [10], and CSV [57] all only evaluated Prime + Probe attack [35, 36]. Besides, another work [26] quantified the cache side-channel leakage but mainly focused on access-driven attacks such as Evict + Time Attack [35]. CacheD [42] identified cache-based timing channels but mainly targeted access-driven attacks such as Evict + Time Attack [35] and time-driven attacks such as Bernstein's Attack [2]. SCADET [38] provided a side-channel detection tool targeting Prime + Probe attack [35, 36]. Moreover, the timing-channel toolkit Mastik [52] was previously presented to experiment with micro-architectural side-channel attacks, but only consider three specific cache timing-based attack types: Prime+Probe [35, 36], Flush+Reload [53] and Flush+Flush [16].

This work is the first work to actually test for all possible timing-based vulnerabilities in caches, not just verify the cases concerning one or few attacks. It is also the first that presents code which can be run on commodity processors and which can generate the Cache Timing Vulnerability Score (CTVS) to evaluate different processors' caches.

3 Modeling for Cache Timing Attacks

The goal of this work is to present the first set of benchmarks which can be used to evaluate all the vulnerabilities of processor caches to timing-based attacks. Such attacks can be used, for example, by Spectre variants, e.g., [24, 27, 29, 39], to extract sensitive information. For each benchmark, if there is observable timing difference on a particular processor, it means that the processor may be vulnerable to the corresponding attack.

3.1 Assumptions and Threat Model

We assume that there is a victim process running on the CPU core and performing secret-dependent memory accesses. There is also a malicious attacker process on the same or different CPU core, whose aim is to determine a secret memory address or address index used by the victim. Both attacker's and victim's accesses affect a cache block in one of the L1 data caches, through which a possible timing channel exists.

The goal of the benchmarks is to evaluate for which types of accesses by the victim and the attacker there is indeed a timing-based vulnerability in caches. The presented benchmarks are not actual security exploits, rather they implement memory-related operations that correspond to all possible timing-based attacks. Each benchmark outputs whether there is a statistically significant timing difference that the attacker could observe to extract information from the timing channel about the secret and unknown address u of the victim.

The current model focuses on all possible timing-based attacks in the L1 data cache. The model includes uses of any memory-related operations (load, store, flush) and cache coherence protocol. The model assumes a multi-core and possibly hyper-threading processor, with a cache hierarchy of local and remote L1 cache, L2 cache, and a shared L3 cache (which is possibly divided into different cache slices).

Current benchmarks do not consider timing-based attacks of other levels in cache hierarchy besides L1, but it should be straightforward to extend to the other levels. We do not consider directory-related attacks [51] or attacks based on replacement policy [48], but it should be possible to model these by adding more states to the model (and still keep an only total of three steps). This work does not cover TLB attacks [15, 19], but there is already a theoretical model for TLBs [12], and similar benchmarks can be developed for TLB attacks (possibly merge with our benchmarks).

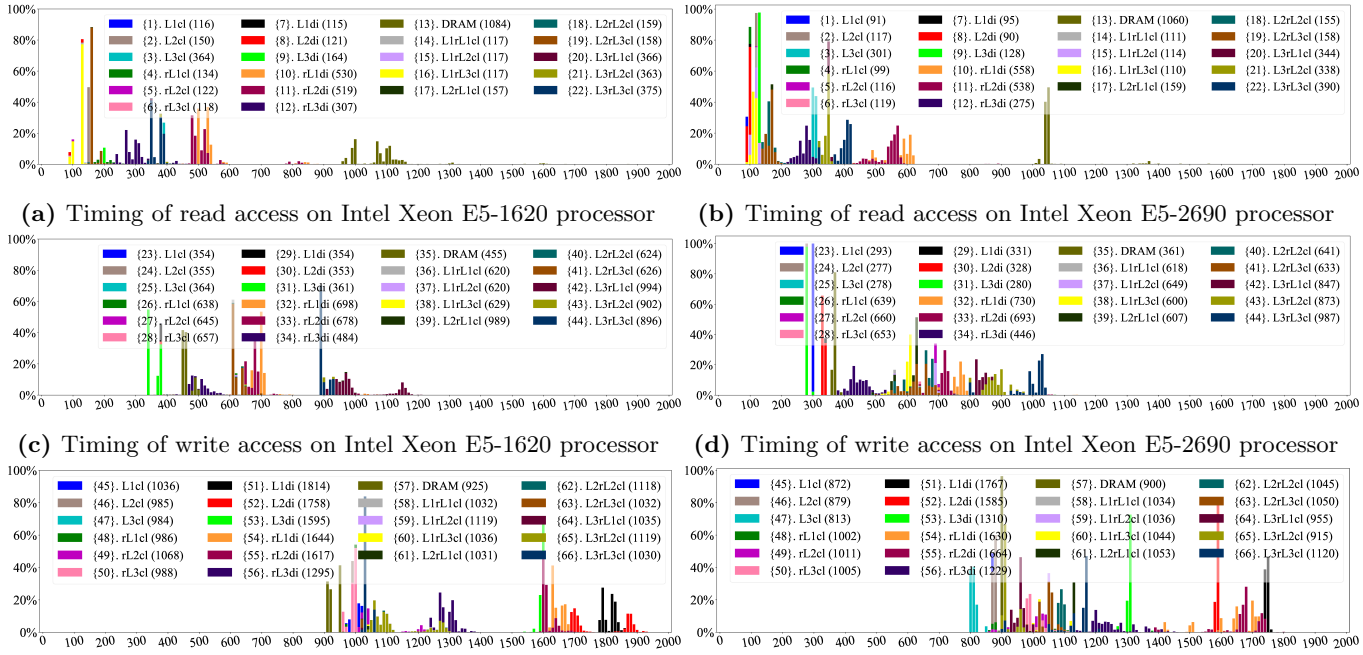
The work considers more than just "fast" and "slow" timings. This means that the influence of structures such as Miss Status Holding Registers (MSHRs), load and store buffers between processor and caches, and line-fill buffers between cache levels are accounted for. However, benchmarks for timing attacks that are just due to these structures could likely be developed. Our analysis is also general for all the cases of the three-step model and we do not differentiate if the access is from the instruction or a prefetcher.

The flush operation in this work refers to the *clflush* instruction in x86, which causes data to be flushed from all levels of caches (including data in other cores) back to the main memory. The timing are measured from when each of the memory-related operations is issued until the instruction commits in the processor pipeline.

3.2 Improved Modeling of Real Processors

We expand the original model [11] by considering more realistic cases for a processor's memory-related operation. The expanded modeling allows us to cover all possible attacks, and uncover new vulnerabilities. For example, some proposals [50] discuss disabling flush instruction to prevent Flush+Reload [53] based attacks. However, because we consider different flush operations, our benchmarks show that using remote access to invalidate (flush) the cache could also result in a vulnerability.

Timing Observation on Local vs. Remote Core. Our cache attack model assumes a multi-core system and possibly a hyper-threading system as well. We model



(a) Timing of read access on Intel Xeon E5-1620 processor

(b) Timing of read access on Intel Xeon E5-2690 processor

(c) Timing of write access on Intel Xeon E5-1620 processor

(d) Timing of write access on Intel Xeon E5-2690 processor

(e) Timing of flush operation on Intel Xeon E5-1620 processor

(f) Timing of flush operation on Intel Xeon E5-2690 processor

Figure 1. Histograms of read, write, and flush operations' timing (each contains 8 operations for timing measurement) under all possible data movements considered in this work. The timing is for the timing observation step, i.e. *Step* 3, in the tested three-step patterns. Note, different processors have different timing, and not all different types of data movements can be distinguished on different processors. The data is presented for Intel Xeon E5-1620 (a, c, e) and Intel Xeon E5-2690 (b, d, f) processors. Numbers in the "{}" in the legend denote the different data movement types. {1} - {22} correspond to read operation, {23} - {44} correspond to write operation, {45} - {66} correspond to flush operation, to access clean L1 data, clean L2 data, clean L3 data, remote clean L1 data, remote clean L2 data, remote clean L3 data, dirty L1 data, dirty L2 data, dirty L3 data, remote dirty L1 data, remote dirty L2 data, remote dirty L3 data, DRAM data, clean data in both L1 and remote L1, clean data in both L1 and remote L2, clean data in both L1 and remote L3, clean data in both L2 and remote L1, clean data in both L2 and remote L2, clean data in both L2 and remote L3, clean data in both L3 and remote L1, clean data in both L3 and remote L2, clean data in both L3 and remote L3, respectively. Numbers in the "()" in the legend show the average cycles needed for completing that type of memory operation. The x axis shows the access latency in cycles.

such a system using two cores: a "local" and a "remote" core, each with L1, L2, and shared L3 caches. The target cache block is located in the local core. Remote core affects the target cache block on the local core by using cache coherence protocol. E.g., perform write operations on the remote core to invalidate the local core's data using cache coherence protocol. As future work, more detailed modeling of multi-core system can be done.

For each read, write or flush operation, it may target the data that is in the local L1 cache, L2 cache, or L3 cache slice, or that is in the remote L1 cache, L2 cache, or L3 cache slice. The cache block can be either in a clean or dirty state for the above 6 locations ($6 \times 2 = 12$ types). The clean data may also be in both local or remote core, which can be in any cache hierarchy (L1, L2, or L3 cache) for both cores ($3 \times 3 = 9$ types). Otherwise, the data is not in any level of the cache hierarchy, i.e., it is in the DRAM (1 type). We consider all these 66 timings (3 operations $\times\ (12 + 9 + 1) = 66$) to be different from each other and use these 66 types of timings in our three-step cache simulator, discussed in Section 4, to determine if a three-step combination can be used in an attack.

Figure 1 shows the histograms of these 66 types of timing observations for Intel Xeon E5-1620 and E5-2690 processors. Based on the histograms, we found that some operations are differentiable from each other, while some are not. In general, the timing is processor-specific, so we need to consider and examine all various cache timings and cannot just assume "fast" and "slow" timings as was done previously [11].

Hyper-Threading vs. Time-Slicing. We consider that the victim and the attacker on one core can either run in time-slicing setting or run in parallel as two hyper-threads (if there is hyper-threading support in the processor). For the case of accesses on "local" vs.

"remote" cores, the accesses on local and remote cores can be done in parallel.

Read (Load) Access vs. Write (Store) Access. For operations related to memory accesses, our model considers that they can be either read (load) access or write (store) access. The timing of writes is not well explored in attacks, except for one work [5]. For example, for Flush + Reload attack, the previous attack [53] uses the load operation in the final step to reload secret data and observe timing. In our model, we also test store operation in the final step to access secret data and reveal that attacks with write in the final step are also effective, for example.

Flush vs. Write Invalidation. In our model, we consider that a flush operation can be achieved by a $clflush$ type instruction, that flushes data from all caches back to main memory, or that by writing the corresponding line in the remote core it will trigger cache coherence and result in the local cache line being invalidated.

4 Derivation of All Vulnerabilities

In this work, we build a new cache three-step simulator based on the new model discussed in Section 3.2. It considers different memory-related operations and differentiates among the 66 timing variations discussed in Section 3.2 that are related to L1 cache timing-based attack for the final timing observation step. Further, we give categorizations of vulnerabilities to find common features that attacks exploit.

4.1 Judging the Effectiveness of Three-Step Combination

In order for a three-step combination to be effective for an attack, at least the unknown victim's address u should be involved in one of the three steps since u is the unknown secret the attacker tries to learn. In this case, the vulnerability will have V_u or V_u^{inv} as one or more of the three-steps to represent the operations on the secret u.

Based on the 17 states shown in Table 1, for the three-step model, the attacker tries to learn the value of u by guessing if u equals to: a, a^{alias} or NIB. a denotes the address that is within the set of sensitive locations x and maps to the target cache line. a^{alias} denotes any data address that belongs to sensitive locations x and also maps to the cache line but is not a. Apart from all possible sensitive address mapping to the target cache line, u may not map to the target cache line the attacker is measuring. We denote these addresses as NIB (not-in-block). Therefore, u can be either a, a^{alias}, or NIB. If the attacker is able to find access time of one value significantly different from the other two values, he or she is able to learn the value of u and the corresponding three-steps is a *Strong* type vulnerability. Meanwhile, if the attacker is not able to clearly distinguish whether u is a, a^{alias}, or NIB based on the timing, but there are still timing differences observed, then the corresponding attacks belong to *Weak* type of vulnerabilities. Otherwise, if the timing is always the same regardless of different values of u, it will be an *Ineffective* three-step combination.

4.2 New Cache Three-Step Simulator

Figure 3 shows the derivation process of vulnerabilities. We wrote Python scripts to develop the cache three-step simulator. The simulator takes all 4913 three-step combinations and 66 types of timing observations as input, checks and outputs the three-steps that belong to *Strong*, *Weak* vulnerabilities, or *Ineffective* types, respectively. For the step that is u-related, since u is in secure range x, the possible candidates of u for a cache block are a, a^{alias}, and NIB, so the simulator checks the timing when u is a, a^{alias}, and NIB, respectively. The timing variance exists if different possible values of u correspond to different timings of the 66 types. We enumerate all possible operations (read/write for access, remote write/flush for invalidation) for a step and consider different timings for each operation. Therefore, each three-step pattern may have different types of timing observations. The rules from our prior three-step model work [11] are used to remove repeat and redundant three-step patterns.

As shown in Figure 3, based on the much finer-grained categorization of timing differences, we derived in total 88 *Strong* effective vulnerabilities and 80 *Weak* effective vulnerabilities after removing repeat three-step patterns. They are shown in Figure 2, where light-blue colored rows (in total 32 types) are the new vulnerabilities (compared to study [11]) which we found through running of new cache three-step simulator (16 types of the original *Strong* effective vulnerabilities [11] become *Weak* vulnerabilities when considering multi-core systems). We provide new names for the new attacks in *Attack Strategy* in Figure 2 while re-use existing names if the attacks were presented before. As validated in Section 8 through the tested processors, there are no other effective vulnerabilities except the types we derive in Figure 2.

4.3 Categorizations of the Vulnerabilities

We first categorize different vulnerabilities as based on internal (I) or external (E) interference. The types that only involve the victim's behavior, V, in the states of *Step* 2 and *Step* 3 are internal interference vulnerabilities (I). The remaining ones are external interference (E) vulnerabilities.

In prior work [11, 56], cache vulnerabilities are categorized as hit-based and miss-based vulnerabilities, based

No.	Vulnerability Type			Type	Attack	Attack Strategy
	$S1$	$S2$	$S3$			
1	A_a^{inv}	V_u	V_a	I-A	[2]	Cache Collision
2	V^{inv}	V_u	V_a	I-A	[2]	
3	A_a^{inv}	V_u	V_a	I-A	[2]	
4	V_a^{inv}	V_u	V_a	I-A	[2]	
5	A_a^{inv}	V_u	A_a	E-A	[5, 9, 11]	Flush + Reload
6	V_a^{inv}	V_u	A_a	E-A	[5, 9, 11]	
7	A^{inv}	V_u	A_a	E-A	[5, 9, 11]	
8	V^{inv}	V_u	A_a	E-A	[5, 9, 11]	
9	V_u^{inv}	A_a	V_u	E-A	new in [3]	Reload + Time
10	V_u^{inv}	V_a	V_u	I-A	new in [3]	
11	A_a	V_u^{inv}	A_a	E-A	[10]	Flush + Probe
12	A_a	V_u^{inv}	V_a	I-A	new in [3]	
13	V_a	V_u^{inv}	A_a	E-A	new in [3]	
14	V_a	V_u^{inv}	V_a	I-A	new in [3]	
15	V_u	A_a^{inv}	V_u	E-A	new in [3]	Flush + Time
16	V_u	V_a^{inv}	V_u	I-A	new in [3]	
17	A^{inv}	V_u^{inv}	A_a	E-A	**new**	Cache Coherence Flush + Reload
18	A^{inv}	V_u^{inv}	V_a	I-A	**new**	
19	V^{inv}	V_u^{inv}	A_a	E-A	**new**	
20	V^{inv}	V_u^{inv}	V_a	I-A	**new**	
21	A_a^{inv}	V_u^{inv}	A_a	E-SA	**new**	Cache Coherence Prime + Probe
22	A_a^{inv}	V_u^{inv}	V_a	I-SA	**new**	
23	V_a^{inv}	V_u^{inv}	A_a	E-SA	**new**	
24	V_a^{inv}	V_u^{inv}	V_a	I-SA	**new**	
25	A_d^{inv}	V_u^{inv}	A_d	E-S	**new**	
26	A_d^{inv}	V_u^{inv}	V_d	I-S	**new**	
27	V_d^{inv}	V_u^{inv}	A_d	E-S	**new**	
28	V_d^{inv}	V_u^{inv}	V_d	I-S	**new**	
29	V_u^{inv}	A_a^{inv}	V_u	E-SA	**new**	Cache Coherence Evict + Time
30	V_u^{inv}	V_a^{inv}	V_u	I-SA	**new**	
31	V_u^{inv}	A_d^{inv}	V_u	E-S	**new**	
32	V_u^{inv}	V_d^{inv}	V_u	I-S	**new**	
33	V_u	V_a	V_u	I-SA	[1]	Bernstein's Attack
34	V_u	V_d	V_u	I-S	[1]	
35	V_d	V_u	V_d	I-S	[1]	
36	V_a	V_u	V_a	I-SA	[1]	
37	V_d	V_u	A_d	E-S	new in [3]	Evict + Probe
38	V_a	V_u	A_a	E-SA	new in [3]	
39	A_d	V_u	V_d	I-S	new in [3]	Prime + Time
40	A_a	V_u	V_a	I-SA	new in [3]	
41	V_u	A_d	V_u	E-S	[7]	Evict + Time
42	V_u	A_a	V_u	E-SA	[7]	
43	A_d	V_u	A_d	E-S	[6–8]	Prime + Probe
44	A_a	V_u	A_a	E-SA	[6–8]	

(a) Timing vulnerabilities with *Step*3 as memory access operation.

No.	Vulnerability Type			Type	Attack	Attack Strategy
	$S1$	$S2$	$S3$			
45	A^{inv}	V_u	V_a^{inv}	I-A	new in [3]	Cache Collision Inv.
46	V^{inv}	V_u	V_a^{inv}	I-A	new in [3]	
47	A_a^{inv}	V_u	V_a^{inv}	I-A	[4]	Flush + Flush
48	V_a^{inv}	V_u	V_a^{inv}	I-A	[4]	
49	A_a^{inv}	V_u	A_a^{inv}	E-A	[4]	
50	V_a^{inv}	V_u	A_a^{inv}	E-A	[4]	
51	A^{inv}	V_u	A_a^{inv}	E-A	new in [3]	Flush + Reload Inv.
52	V^{inv}	V_u	A_a^{inv}	E-A	new in [3]	
53	V_u^{inv}	A_a	V_u^{inv}	E-A	new in [3]	Reload + Time Inv.
54	V_u^{inv}	V_a	V_u^{inv}	I-A	new in [3]	
55	A_a	V_u^{inv}	A_a^{inv}	E-A	new in [3]	Flush + Probe Inv.
56	A_a	V_u^{inv}	V_a^{inv}	I-A	new in [3]	
57	V_a	V_u^{inv}	A_a^{inv}	E-A	new in [3]	
58	V_a	V_u^{inv}	V_a^{inv}	I-A	new in [3]	
59	V_u	A_a^{inv}	V_u^{inv}	E-A	new in [3]	Flush + Time Inv.
60	V_u	V_a^{inv}	V_u^{inv}	I-A	new in [3]	
61	A^{inv}	V_u^{inv}	A_a^{inv}	E-A	**new**	Cache Coherence Flush + Reload Inv.
62	A^{inv}	V_u^{inv}	V_a^{inv}	I-A	**new**	
63	V^{inv}	V_u^{inv}	A_a^{inv}	E-A	**new**	
64	V^{inv}	V_u^{inv}	V_a^{inv}	I-A	**new**	
65	A_a^{inv}	V_u^{inv}	A_a^{inv}	E-SA	**new**	Cache Coherence Prime + Probe Inv.
66	A_a^{inv}	V_u^{inv}	V_a^{inv}	I-SA	**new**	
67	V_a^{inv}	V_u^{inv}	A_a^{inv}	E-SA	**new**	
68	V_a^{inv}	V_u^{inv}	V_a^{inv}	I-SA	**new**	
69	A_d^{inv}	V_u^{inv}	A_d^{inv}	E-S	**new**	
70	A_d^{inv}	V_u^{inv}	V_d^{inv}	I-S	**new**	
71	V_d^{inv}	V_u^{inv}	A_d^{inv}	E-S	**new**	
72	V_d^{inv}	V_u^{inv}	V_d^{inv}	I-S	**new**	
73	V_u^{inv}	A_a^{inv}	V_u^{inv}	E-SA	**new**	Cache Coherence Evict + Time Inv.
74	V_u^{inv}	V_a^{inv}	V_u^{inv}	I-SA	**new**	
75	V_u^{inv}	A_d^{inv}	V_u^{inv}	E-S	**new**	
76	V_u^{inv}	V_d^{inv}	V_u^{inv}	I-S	**new**	
77	V_u	V_a	V_u^{inv}	I-SA	new in [3]	Bernstein's Inv. Attack
78	V_u	V_d	V_u^{inv}	I-S	new in [3]	
79	V_d	V_u	V_d^{inv}	I-S	new in [3]	
80	V_a	V_u	V_a^{inv}	I-SA	new in [3]	
81	V_d	V_u	A_d^{inv}	E-S	new in [3]	Evict + Probe Inv.
82	V_a	V_u	A_a^{inv}	E-SA	new in [3]	
83	A_d	V_u	V_d^{inv}	I-S	new in [3]	Prime + Time Inv.
84	A_a	V_u	V_a^{inv}	I-SA	new in [3]	
85	V_u	A_d	V_u^{inv}	E-S	new in [3]	Evict + Time Inv.
86	V_u	A_a	V_u^{inv}	E-SA	new in [3]	
87	A_d	V_u	A_d^{inv}	E-S	new in [3]	Prime + Probe Inv.
88	A_a	V_u	A_a^{inv}	E-SA	new in [3]	

(b) Timing vulnerabilities with *Step*3 as invalidation operation.

Figure 2. The table shows all the L1 cache timing-based vulnerabilities. The *No.* column assigns each type of vulnerability a number. The *Vulnerability Type* column shows the three steps that define each vulnerability. The *Type* column proposes the categorization the vulnerability belongs to. "E" and "I" are for internal and external interference types, respectively. "S", "A" and "SA" are set-based, address-based types and the types that are both set-based and address-based, respectively. The *Attack* column shows if a vulnerability has been previously presented in the literature. The *Attack Strategy* column gives a common name for each set of vulnerabilities that would be exploited in an attack in a similar manner. *Inv.* means invalidation. Light-blue colored rows are the vulnerabilities which are first presented in this work.

on the cache behaviors the attackers want to observe (cache misses or hits). This definition does not fit our model since there are different types of timings for L1 data hits and misses in the real machines. For example, attacks can derive information using timing difference from two types of cache misses.

Therefore, we further categorize the vulnerabilities as address-based (A) if they are able to derive the cache line address of u by observing cache hit of u and obtaining

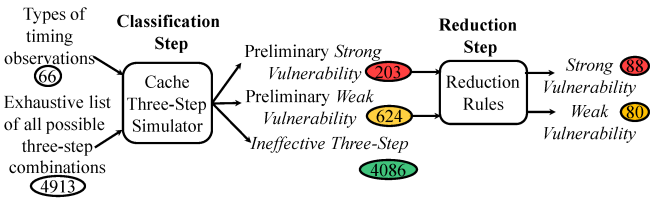


Figure 3. The derivation process of all the *Strong* and *Weak* types of L1 cache timing-based vulnerabilities.

different timing compared with other candidate data. Set-based (S) vulnerabilities are the ones that can know the mapped set of u by conflicting and generating eviction between u and candidate data addresses. The third type are the ones that potentially derive information from set or address (SA) depending on timing differences derived for all the candidates of u. For example, SA type #33 vulnerability $V_u \rightsquigarrow V_a \rightsquigarrow V_u$ can be set-based if a and u are not the same but map to the same cache set, which differs in timing between {2}{8}{24}{30}, a local L2 hit, and {1}{7}{23}{29}, a local L1 hit. Or it can be address-based if a maps to u and *Step* 1 (V_u) and *Step* 2 (V_a) are accessed by different operations (read or write), which have different timing between reads of L1 clean data and dirty data, {1} and {7}, or writes of L1 clean data and dirty data, {23} and {29}.

5 Benchmark Implementation

For each vulnerability, there are three steps, where each can be: read or write access for a memory access operation, or flush or write in the remote core for an invalidation-related operation. Thus, there are in total of $2^3 = 8$ cases considering different types of operations. Further, if the vulnerabilities have both the victim and the attacker running in one core, these two parties can run either time-slicing or multi-threading. Based on that, one case may be doubled for running in two settings. So for one vulnerability type, there are corresponding 8 - 16 cases depending on the specific vulnerability. In total, there are 1094 benchmarks for all 88 *Strong* type vulnerabilities. We wrote C programs to automatically generate the binaries for each of the 1094 benchmarks.

5.1 Judging A Three-Step Combination

For a specific benchmark that implements one case of the three-step combinations, following the idea of the cache three-step simulator in Section 4, if the step is V_u, the benchmarks separately test the timing when V_u is V_a, $V_{a^{alias}}$, or V_{NIB}. If the step is V_u^{inv}, the benchmarks separately test the timing when V_u^{inv} is V_a^{inv}, $V_{a^{alias}}^{inv}$, or V_{NIB}^{inv}. The timing of the last step in the three-step pattern is measured. For each of the cases, there is RUN_NUM number of trials, and Welch's t-test [46] is used to distinguish the distributions of the measured timings. We consider two distributions to be significantly different from each other if the probability of observing the data given that they come from the same distribution is less than 0.05%.

For an effective vulnerability, one of the three candidates of V_u (or V_u^{inv}) should generate timing distribution that is statistically different from the other two candidates, which we use to extract information from the runs. This is for the *Strong* vulnerability types which are 88 types in total. The 80 *Weak* vulnerability types are not currently considered in the benchmarks but can be straightforward to add if needed. At end of each benchmark run, the benchmark outputs if there was significant timing difference – "vulnerability is found", or not – "vulnerability not found".

5.2 Timing Measurement and Noise Minimization

We use *rdtsc* instruction in our benchmarks to do timing measurements, which is the most effective method compared with hardware performance counters, which may be limited [16] or lacking-determinism [9], or using a "counting" thread. AMD's *rdtsc* instruction is not as accurate as Intel machine's, but there are many works [19, 20] showing that it is also able to be used for cache timing-based attacks.

Noise and variation in the timing measurements could further result in false negatives (if the time measurement was not accurate enough to distinguish different timings of accesses) or false positives (if timing changes resulted in timing measurement differences even though there is no timing difference). We isolate cores to reduce the software noise to minimize the false positives. To reduce the false negatives from the noise, instead of measuring just one cache block, we arbitrarily chose 8 cache blocks from different cache sets to do operation on. Further, the measurements are all repeated RUN_NUM times and collect statistical data. The *fence* instructions are added between each memory-related instruction to enforce an ordering constraint for the attacks.

To reduce the variation of the timing among different cache sets and further minimize the false negatives, the timing measurement of the last step is repeated for each test if the last step is u-related step. Specifically, right after the third step's timing measurement, we trigger and measure the timing of this step again, which is guaranteed to result in an L1 cache hit timing or timing to invalidate the data that is not in the caches, depending on the concrete memory operations. We then compare the timing of the third step with the repeated third step. This eliminates any variations in timing among different cache sets.

```
1.  #define LIM 0.0005
2.  //mutex to sequence three-step operations
3.  mutex = mmap(mutex_size, PROT_READ|PROT_WRITE, ...)
4.  init_array(arr); //initialize data array and load it into L1, L2, and L3
5.  mutex[0] = NOONE_RUN;
6.  if ((pid_l1=fork()) < 0) { exit(1); //fail to fork process}       //local attacker
7.  else if (pid_l1==0){
8.     CPU_SET(att_num, &mycpuset);
9.     sched_setaffinity(getpid(), sizeof(cpu_set_t), &mycpuset);
10.    for (int m=0; m<RUN_NUM; m++){
11.       for (int j=0; j<4; j++){
12.             // before the attack, initialize mutex
13.             if(mutex[0]==NOONE_RUN){ mutex[0]=STEP1_RUN;}
14.             // step 2 Aa
15.             while(mutex[0]!=STEP2_RUN) sched_yield;
16.             attacker_write_8_access(a);
17.             mutex[0]=STEP3_RUN;
18.    }} exit(0);}
19. if ((pid_l2=fork()) < 0) { exit(1); //fail to fork process}       //local victim
20. else if (pid_l2==0){
21.    CPU_SET(vic_num, &mycpuset);
22.    sched_setaffinity(getpid(), sizeof(cpu_set_t), &mycpuset);
23.    for (int m=0; m<RUN_NUM; m++){
24.       for (int j=0; j<4; j++){
25.             // step 1 Vu
26.             while(mutex[0]!=STEP1_RUN) sched_yield;
27.             if(j==0) victim_read_8_access(a);
28.             else if (j==1) victim_read_8_access(a_alias);
29.             else if (j==2) victim_read_8_access(NIB);
30.             else if (j==3) dummy_operation;
31.             mutex[0]=STEP2_RUN;
32.             // step 3 Vu and measure time
33.             while(mutex[0]!=STEP3_RUN) sched_yield;
34.             if(j==0) {victim_write_8_access_time(a,t);
35.                       victim_write_8_access_time(a, t_r);}
36.             else if (j==1) {victim_write_8_access_time(a_alias, t);
37.                             victim_write_8_access_time(a_alias, t_r);}
38.             else if (j==2) {victim_write_8_access_time(NIB, t);
39.                             victim_write_8_access_time(NIB, t_r);}
40.             else if (j==3)  dummy_operation;
41.             // timing store
42.             store_third_step_timing(j,t);
43.             store_repeat_access_timing(j,t_r);
44.             mutex[0]=STEP1_RUN;
45.       }}
46.    //timing analysis
47.    if((p_value(a, a_alias)<LIM && p_value(a, NIB)<LIM)||(
48.      p_value(a, NIB)<LIM&& p_value(a_alias, NIB)<LIM)||(
49.      p_value(a, a_alias)<LIM&& p_value(a_alias, NIB)<LIM)
50.    &&((!u_last_step) ||
51.   ((pvalue(a_dif, a_alias_dif)<LIM&&pvalue(a_dif, NIB_dif)<LIM)||(
52.     pvalue(a_dif, NIB_dif)<LIM&& pvalue(a_alias_dif, NIB_dif)<LIM)||(
53.     pvalue(a_dif, a_alias_dif)<LIM&& pvalue(a_alias_dif, NIB_dif)<LIM))))
54.          printf("Vulnerability is found");
55.    else printf("Vulnerability not found");
56.    exit(0);}
```

Figure 4. The pesudo code of #42 vulnerability $V_u \rightsquigarrow A_a \rightsquigarrow V_u$ for read (V_u), write (A_a), and write (V_u) case running hyper-threading.

5.3 Benchmark Code Example

Figure 4 shows an example pseudo code of #42 vulnerability $V_u \rightsquigarrow A_a \rightsquigarrow V_u$'s benchmark for read ($V_u$), write ($A_a$), and write ($V_u$) access of the three steps and running in hyper-threading setting.

First, we define probability bound of Welch's t-test (line 1) and initialize a shared array (line 2-3) used by mutexes to control the sequence of the three-step accesses. Then, the data (stored in the array) that will be accessed by the victim and the attacker is loaded into the L1 cache (line 4), and consequently possibly brought into L2 and L3 caches. We use *fork()* (line 6 and line 19) to create sub-process, one for the victim and one for the attacker in this example. Each remote and local victim and attacker will have one sub-process throughout the whole test. Each sub-process is assigned to a hardware thread (line 8-9, line 21-22). When running hyper-threading, two local or two remote sub-processes are run in different hardware threads of one CPU, if applicable. If running time-slicing, sub-processes are assigned to one hardware thread. Within each sub-process, the test will be run for a certain predefined RUN_NUM (line 10 and line 23) times so the timing statistics can be done based on a large number of runs. We set RUN_NUM at 600 to minimize noise and maintain a suitable test set number for Welch's t-test to measure distributions.

As discussed in Section 4.1, for all the effective vulnerabilities, there will be at least one V_u step (or V_u^{inv}). Within each test, the three candidate values (i.e., V_a, $V_{a^{alias}}$, or V_{NIB}) will be tested for the V_u or V_u^{inv} (line 11 and line 24). The "dummy operation" branch is used to avoid making the third branch to be the last branch, which we found experimentally has an abnormal stable longer timing measurement result.

Figure 4 shows a test performing *Step* 1 (V_u, line 25-31), *Step* 2 (A_a, line 14-17) and *Step* 3 (V_u, line 32-44). Last step *Step* 3 is performed twice and results are stored (line 41-43). The first access of *Step* 3 is done to measure if the attacker can observe timing differences when running different values of u. The second access of the third step will always be a hit (fast timing, and is used to obtain baseline "fast" timing for that cache set, as we observed different cache sets can have different timing). In this case, we can collect results of difference between the first access timing and the second access timing for each candidate of u to limit the possibilities that timing difference is due to different cache sets but not different values of u.

In the end, Welch's t-test is first applied to each statistical distribution of candidate values for V_u (or V_u^{inv}) to see whether the attacker can observe different timing when V_u refers to different addresses (line 47-49). If the three-step patterns have u-related step as the last step (implemented by *u_last_step* in line 50), to remove the noise in the timing among different cache sets, the second access timing is considered. Welch's t-test is applied to test the difference of the first and the second access of the last step *Step* 3. Only if one candidate's distribution has significant timing difference compared with the other

Table 2. Configurations of the experimental machines, which all have 64B L1 cache line size. (1) denotes the number of hardware threads sharing one L1 cache; (2) denotes the number of hardware threads per socket; (3) denotes the number of sockets.

Model Name	L1-D Cache	L1-I Cache	L2 Cache	L3 Cache	(1)	(2)	(3)
Intel Xeon E5-1620	32KB, 8-way	32KB, 8-way	256KB, 8-way	10MB, 20-way	2	8	1
Intel Xeon E5-2667	32KB, 8-way	32KB, 8-way	256KB, 8-way	15MB, 20-way	2	12	2
Intel Xeon E5-2690	32KB, 8-way	32KB, 8-way	256KB, 8-way	20MB, 20-way	2	16	1
Intel Core i5-4570	32KB, 8-way	32KB, 8-way	256KB, 8-way	6MB, 12-way	1	4	1
Intel Xeon E5-2686	32KB, 8-way	32KB, 8-way	1MB, 16-way	33MB, 11-way	1	4	1
Intel Xeon P-8175	32KB, 8-way	32KB, 8-way	256KB, 8-way	45MB, 20-way	2	8	1
AMD FX-8150	16KB, 4-way	64KB, 2-way	2MB, 16-way	8MB, 64-way	1	8	1
AMD EPYC 7571	32KB, 8-way	64KB, 4-way	512KB	8MB	2	4	1

two, the cache sets' noise is shown to be not the reason of timing difference and the corresponding vulnerability is judged to be effective (line 51-53).

6 Evaluation and Security Discussion

The experimental results reported for Intel processors were performed on Intel Core i5-4570, Xeon E5-2690, E5-2667, E5-1620, P-8175 and E5-2686 CPUs. The AMD tests were on AMD EPYC 7571 and AMD FX-8150. P-8175, E5-2686 and AMD EPYC 7571 instance are from Amazon EC2. Table 2 shows the processor configurations.

6.1 Vulnerability Evaluation on Commodity CPUs

We evaluated 88 *Strong* effective vulnerabilities shown in Figure 2. Figure 5 lists the experimental results when testing the 9 types of processor configurations upon 88 effective vulnerabilities. For each type of processors, a dot showing up in the figure means that the machine is vulnerable to this vulnerability. Apart from the 9 types of tested processors, Figure 5 has a row showing if the vulnerability is found in at least one tested processor, i.e., *or* result, and another row showing if the vulnerability is found in all tested processors, i.e., *and* result.

Figure 5 shows that 88 effective vulnerabilities are mostly found in all the tested CPUs. Since our new cache three-step simulator considers the ideal case where 66 types of timing observations all have unique results, it outputs all the possible vulnerability types. For commodity processors, a subset of them is shown to be effective. This is due to the actual cache implementation and timing measurement methods, making some of the timing of the 66 types not differentiable, as is shown in histograms of Figure 1. Figure 5 also demonstrates that different machines are vulnerable to different types of attacks. The *and* result of 9 types of processor configuration experiments have relatively small percentage of vulnerabilities to which machines are all vulnerable. We further list the statistical results as CTVS for each machine in Section 7.

6.2 Analysis of Vulnerabilities Found

Figure 6 shows the results of benchmarks for all the cases of the 88 vulnerability types. Machines not supporting hyper-threading have much fewer effective cases. Similar to Figure 5, the dot means the related processor is vulnerable to the specific case. The gray vertical lines are used to group all the cases per vulnerability (there are thus 88 vertical bars and groupings). We further collect the data in Figure 6 and group them with different *Step* 3 types as the timing observation steps in Table 3 to compare effects of different operations on processor cache timing attacks.

Local read and local write of timing observation step. Previous attacks normally used *read* access to implement the side-channel attacks, as analyzed in Section 3.2. However, write access is shown in Figure 6 and Table 3 to be an effective method to implement attacks as well. It has generally smaller rate compared with read access to trigger effective vulnerabilities of different cases, especially for tested machine Intel Xeon E5-1620 and E5-2690. For the 44 types of vulnerabilities (#1 - #44) that have access operation as timing observation step, Figure 6 demonstrates that there are 38 out of 44 vulnerabilities to which at least one machine is vulnerable when using read as the timing observation step. While using write access as the timing observation step, 34 out of 44 vulnerabilities are vulnerable to at least one machine.

Invalidation using cache coherence or flush for timing observation step. According to Table 3, the percentage of vulnerabilities to which the machine is vulnerable mainly depends on processor types when comparing different invalidation-related operation as the timing observation step. Among the tested processors, Intel Xeon E5-2667 running inter-chip, AMD FX-8150 and AMD EPYC are more vulnerable to remote write

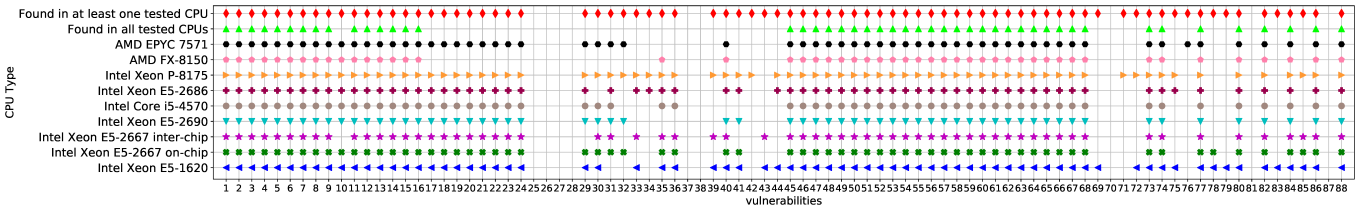
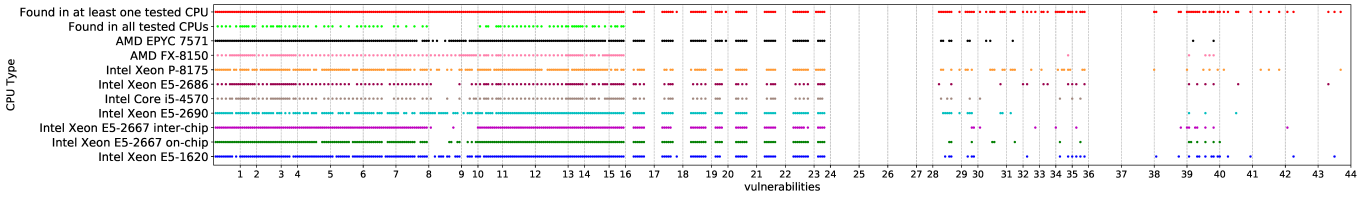
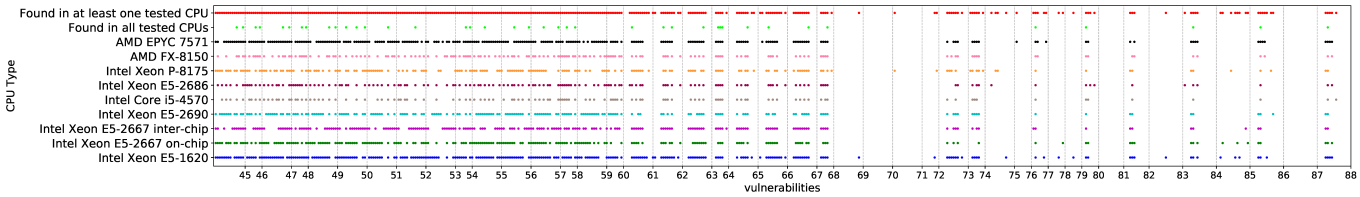


Figure 5. Evaluation of 88 *Strong* types of vulnerabilities on different machines. A dot means the corresponding processor is vulnerable to the vulnerability type. Intel Xeon E5-2667 in our lab has two sockets. Therefore, the local and remote core can be both in one socket, i.e., run on-chip; or local and remote core can be in different sockets, i.e., run inter-chip.



(a) #1 - #44 vulnerability testing results on different machines.



(b) #45 - #88 vulnerability testing results on different machines.

Figure 6. Evaluation of 88 *Strong* types of vulnerabilities for all the cases. A dot means the corresponding processor is vulnerable to the vulnerability case. For each vulnerability, a fixed number of cases (see Section 5) are tested according to the vulnerability type. And there are in total 1094 cases for 88 *Strong* types of vulnerabilities.

as the timing observation step. Intel Xeon E5-1620, E5-2667 running on-chip, E5-2690, E5-2686, P-8175, and Core i5-4570 are more vulnerable to flush observation step. Overall, for the 44 types of vulnerabilities (#45 - #88) that have invalidation for timing observation, remote write and flush operations both have 38 out of 44 vulnerabilities to which at least one machine is vulnerable.

Running time-slicing or hyper-threading. Besides different kinds of operations, we also collect results in Table 4 for running time-slicing and hyper-threading when the victim and the attacker run on the same core (either local or remote core). There are also vulnerabilities for which the victim and the attacker run on different cores, or vulnerabilities only having victim steps. Based on the results, running time-slicing is more vulnerable compared with running hyper-threading for Intel processors. While AMD processor EPYC 7571 shows that running hyper-threading is more vulnerable. Furthermore, hyper-threading provides more choices for the attacker to exploit the corresponding vulnerability in different ways.

7 Take-Aways and Need for Cache Timing Vulnerability Benchmarks

Table 5 shows the Cache Timing Vulnerability Score (CTVS) which represents the percentage of the vulnerabilities that are effective for the machine. The number on the right of "/" is the total cases of vulnerabilities for the corresponding categorization; the number on the left of "/" is the number of types to which the corresponding processor is vulnerable. For CTVS number, smaller is better. For all the 88 *Strong* type vulnerabilities, AMD FX-8150 has relatively better CTVS compared with Intel machines. Xeon E5-1620 and P-8175 are the most vulnerable ones among Intel processors. Otherwise, the Xeon family and AMD EPYC 7571 are generally similar.

CTVS numbers vary by different machines and type of vulnerabilities. In Table 5, CTVS numbers

Table 3. Percentage of vulnerability cases that are effective for different types of timing observation steps for different machine configurations. The number on the right of "/" is the total cases of vulnerabilities for the corresponding categorization; the number on the left of "/" is the number of cases to which the corresponding processor is vulnerable. Machines labeled * do not support hyper-threading in hardware.

Model Name	Local Read	Local Write	Remote Write to Inv.	Flush to Inv.
Intel Xeon E5-1620	137/277	118/277	127/277	129/263
Intel Xeon E5-2667 on-chip	121/277	117/277	80/277	119/263
Intel Xeon E5-2667 inter-chip	127/277	111/277	124/277	72/263
Intel Xeon E5-2690	128/277	101/277	77/277	107/263
Intel Core i5-4570*	82/277	66/277	57/277	63/263
Intel Xeon E5-2686*	87/277	74/277	69/277	80/263
Intel Xeon P-8175	124/277	120/277	75/277	105/263
AMD FX-8150*	68/277	65/277	89/277	65/263
AMD EPYC 7571	125/277	125/277	124/277	114/263
in all CPUs	49/277	34/277	10/277	30/263
at least one CPU	175/277	150/277	162/277	162/263

for *in all CPUs* are small, demonstrating that only a few attacks can be effective in all the processors. These numbers are expected to be even smaller if more processors are tested. CTVS numbers for *at lest one CPU* are large, confirming that nearly all of the vulnerabilities derived by the new three-step model are found in real processors.

A type vulnerabilities generally have higher effective rates than *S* type vulnerabilities. This is because that *S* type vulnerabilities normally differentiate timing between L1 cache and L2 cache accesses, i.e., accessing or invalidating L1 or L2 data, which are shown in the histograms in Figure 1 to be much smaller compared with the difference between L1 cache hit and DRAM hit, for example. Especially, the timing difference between remote write to invalidate dirty L1 data and L2 data is almost non-differentiable, resulting in that related vulnerabilities (especially #25 - #28) are found to be not effective in all tested processors (shown in Figure 5). *A* type vulnerabilities generally rely on timing differences between L1 cache hit and DRAM hit, or L1 cache hit and remote L1 cache hit; histograms in Figure 1 demonstrate that these access types have large timing differences, making these vulnerabilities much more effective. *SA* type vulnerabilities generally leverage the timing differences between clean L1 data invalidation and dirty L1 data invalidation or between local access of remote clean L1 data and remote dirty L1 data; histograms in Figure 1 again show large timing differences for these, and related vulnerabilities are found to be very effective by CTVS. Meanwhile, for *I* and *E* type vulnerabilities, they do not have an explicit distinction of CTVS numbers for the tested processors.

Table 4. Percentage of vulnerability cases that are effective for the victim (Vic.) and the attacker (Att.) running the same core (time-slicing or hyper-threading), running different cores or within the victim for different machine configurations. The number on the right of "/" is the total cases of vulnerabilities for the corresponding categorization; the number on the left of "/" is the number of cases to which the corresponding processor is vulnerable. Machines labeled * do not support hyper-threading.

Model Name	Vic., Att. Same Core		Vic., Att. on Different Cores	Within Victim
	Time-Slicing	Hyper-Threading		
Intel Xeon E5-1620	181/390	174/390	51/90	105/224
Intel Xeon E5-2667 on-chip	156/390	146/390	52/90	83/224
Intel Xeon E5-2667 inter-chip	151/390	146/390	49/90	88/224
Intel Xeon E5-2690	144/390	138/390	46/90	85/224
Intel Core i5-4570*	143/390	0/390	46/90	79/224
Intel Xeon E5-2686*	166/390	0/390	50/90	94/224
Intel Xeon P-8175	148/390	143/390	38/90	95/224
AMD FX-8150*	155/390	0/390	43/90	89/224
AMD EPYC 7571	159/390	171/390	55/90	103/224
in all CPUs	67/390	0/390	18/90	38/224
at least one CPU	223/390	217/390	61/90	148/224

Use CTVS to build custom defenses. CTVS has shown that different processors are vulnerable to different attacks. Consequently, customized software or hardware defenses can be deployed for each processor based on the CTVS score, rather than defending vulnerabilities not present in the specific processor's caches. For software defenses, the access patterns from the benchmarks could be used as a reference for scanning software to find if it has similar patterns, e.g., to find malicious software that has such attack patterns.

Understand limits of existing defenses using three-step model. Further, CTVS and our three-step model have shown new attack types which are unknown before, and thus, not considered by defenses based on monitoring performance counters, e.g., study [34]. This

Table 5. Cache Timing Vulnerability Score (CTVS) for each of the tested processors. The number on the right of "/" is the total cases of vulnerabilities for the corresponding categorization; the number on the left of "/" is the number of types to which the corresponding processor is vulnerable. Smaller is better. "I" and "E" are internal and external interference vulnerabilities, respectively. "S" and "A" are set-based and address-based vulnerabilities, respectively. "SA" are the ones that are both set-based and address-based.

Model Name	CTVS Score	*I-A* Vul.	*I-S* Vul.	*I-SA* Vul.	*E-A* Vul.	*E-S* Vul.	*E-SA* Vul.
Intel Xeon E5-1620	73/88	20/20	6/12	12/12	20/20	5/12	10/12
Intel Xeon E5-2667 on-chip	66/88	20/20	3/12	11/12	20/20	3/12	9/12
Intel Xeon E5-2667 inter-chip	64/88	19/20	2/12	12/12	20/20	3/12	8/12
Intel Xeon E5-2690	62/88	20/20	1/12	10/12	20/20	2/12	9/12
Intel Core i5-4570	61/88	20/20	1/12	10/12	20/20	1/12	9/12
Intel Xeon E5-2686	66/88	20/20	2/12	11/12	20/20	3/12	10/12
Intel Xeon P-8175	73/88	20/20	5/12	12/12	20/20	5/12	11/12
AMD FX-8150	50/88	18/20	1/12	7/12	18/20	0/12	6/12
AMD EPYC 7571	62/88	20/20	2/12	10/12	20/20	1/12	9/12
in all CPUs	47/88	17/20	0/12	6/12	18/20	0/12	6/12
at least one CPU	79/88	20/20	9/12	12/12	20/20	7/12	11/12

points to the requirement of using new or different performance counter types in works that use active monitoring, for example.

Understand micro-architecture using CTVS. The vulnerability score can also be used to help understand the implementation of different processors especially the micro-architectures. For example, according to Figure 5, vulnerability #78 $V_u \rightsquigarrow V_d \rightsquigarrow V_u^{inv}$ and #79 $V_d \rightsquigarrow V_u \rightsquigarrow V_d^{inv}$ fully show up on Intel E5-1620 while do not show up on Intel E5-2690. As shown in Figure 1, flushing clean L1 data (L1cl) to DRAM and flushing clean L2 data (L2cl) to DRAM have large timing differences for Intel E5-1620 (1036 vs. 985 average cycles shown in Figure 1(e)), but are non-differentiable for Intel E5-2690 (872 vs. 879 average cycles shown in Figure 1(f)). With the smaller difference, it is not possible to distinguish the timing with high confidence and corresponding vulnerabilities are highly likely unexploitable on this processor.

Diving deeper, the reason for the timing variation may due to the different clock speed of Intel E5-1620 and Intel E5-2690 (3.6GHz vs. 2.9GHz), where faster clock speed will make long memory-related operations more differentiable, even if the absolute timing differences are the same. Besides that, Intel E5-1620 does not support Flex Memory Access, which improves memory access efficiency. Intel E5-2690 supports it, making two operations less differentiable on timing.

8 Validation of the Three-Step Model

To validate if there are any other vulnerabilities that are left out apart from all the effective vulnerabilities we derived from our cache three-step simulator, we empirically ran benchmarks for all the $17^3 = 4913$ three-step combinations for 9 processor configurations.

We discovered a number of three-steps, besides the *Strong*, *Weak* and repeat types, returned by the benchmarks to have timing variations but consider all of them as false positives. The false positives that show up in every processor we tested all have the second or the third step to be A^{inv}, V^{inv} or $\star$. The corresponding types cannot be any effective vulnerabilities because these three types of states will make the attacker lose track of useful information due to whole cache flush (A^{inv}, V^{inv}) or zero-knowledge state inference ($\star$) if they are in *Step* 2 or *Step* 3. Reason of three-steps with the second or the third step as A^{inv}, V^{inv} to seem to be effective in the result of running the benchmarks is that whole cache flush currently cannot be implemented under user-level privilege. We use approximate method to implement these states in the benchmark by invalidating every address that is related to the attacks. An approximate method is also used for $\star$ to simulate the zero-knowledge state. Therefore, the timings of A^{inv}, V^{inv} and $\star$ have extra noise leading to the false positives.

Overall, we found that there are no effective vulnerabilities that are not covered by the vulnerabilities we derived. Detailed analysis is not shown due to space limits of the paper.

9 Conclusion

This work presented a new three-step model and the first benchmark suite for evaluating all 88 possible *Strong* cache timing-based attack types in processors. The model allowed us to find 32 new timing attack types. Further, we implemented scripts to auto-generate the 1094 benchmark tests from our three-step model's 88 theoretical

attack types for testing different combinations and types of instructions that can lead to attacks on real processors. The benchmarks were run on a number of commodity processors to gave each machine the Cache Timing Vulnerability Score (CTVS) to measure the degree of the machine's robustness against cache timing-based vulnerabilities. The three-step model, benchmarks, and the CTVS can be used to measure existing systems and help design future secure caches and other defense mechanisms.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported by NSF grants 1651945 and 1813797, and through SRC award number 2844.001.

References

[1] Onur Acıiçmez and Çetin Kaya Koç. 2006. Trace-Driven Cache Attacks on AES (short paper). In *International Conference on Information and Communications Security*. 112–121.

[2] Daniel J Bernstein. 2005. Cache-Timing Attacks on AES. (2005).

[3] Joseph Bonneau and Ilya Mironov. 2006. Cache-Collision Timing Attacks against AES. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. 201–215.

[4] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Srinivas Devadas, et al. 2019. MI6: Secure enclaves in a speculative out-of-order processor. In *International Symposium on Microarchitecture (MICRO)*. 42–56.

[5] Common Vulnerabilities and Exposures 2018. Speculative Store Bypass Bug CVE, 2018. CVE 2018-3639. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3639

[6] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security Symposium (USENIX)*. 857–874.

[7] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. 2015. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In *Network and Distributed System Security Symposium (NDSS)*. 8–11.

[8] Joan Daemen and Vincent Rijmen. 1999. AES Proposal: Rijndael. (1999).

[9] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. 2019. SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security. In *Symposium on Security and Privacy (S&P)*.

[10] John Demme, Robert Martin, Adam Waksman, and Simha Sethumadhavan. 2012. Side-Channel Vulnerability Factor: A Metric for Measuring Information Leakage. In *International Symposium on Computer Architecture (ISCA)*. 106–117.

[11] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. 2019. Analysis of Secure Caches Using a Three-Step Model for Timing-Based Attacks. *Journal of Hardware and Systems Security* 3, 4 (December 2019), 397–425.

[12] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. 2019. Secure TLBs. In *International Symposium on Computer Architecture (ISCA)*.

[13] Leonid Domnitser, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2010. A Predictive Model for Cache-Based Side Channels in Multicore and Multithreaded Microprocessors. In *International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*. 70–85.

[14] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012. Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side Channel Attacks. *Transactions on Architecture and Code Optimization (TACO)* 8, 4 (2012), 35.

[15] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium (USENIX)*. 955–972.

[16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+ Flush: a Fast and Stealthy Cache Attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 279–299.

[17] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games–Bringing Access-Based Cache Attacks on AES to Practice. In *Symposium on Security and Privacy (S&P)*. 490–505.

[18] Zecheng He and Ruby B Lee. 2017. How Secure is Your Cache against Side-Channel Attacks?. In *International Symposium on Microarchitecture (MICRO)*. 341–353.

[19] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical Timing Side Channel Attacks Against Kernel Space ASLR. In *Symposium on Security and Privacy (S&P)*. 191–205.

[20] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. Cross Processor Cache Attacks. In *Asia Conference on Computer and Communications Security (AsiaCCS)*. 353–364.

[21] Mehmet Kayaalp, Khaled N Khasawneh, Hodjat Asghari Esfeden, Jesse Elwell, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. 2017. RIC: Relaxed Inclusion Caches for Mitigating LLC Side-Channel attacks. In *Design Automation Conference (DAC)*. 1–6.

[22] Georgios Keramidas, Alexandros Antonopoulos, Dimitrios N Serpanos, and Stefanos Kaxiras. 2008. Non Deterministic Caches: A Simple and Effective Defense against Side Channel Attacks. *Design Automation for Embedded Systems* 12, 3 (2008), 221–230.

[23] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *International Symposium on Microarchitecture (MICRO)*. 974–987.

[24] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre attacks: Exploiting Speculative Execution. In *Symposium on Security and Privacy (S&P)*. 1–19.

[25] Jingfei Kong, Onur Aciiçmez, Jean-Pierre Seifert, and Huiyang Zhou. 2009. Hardware-Software Integrated Approaches to Defend against Software Cache-Based Side Channel Attacks. In *International Symposium on High Performance Computer Architecture (HPCA)*. 393–404.

[26] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. 2012. Automatic Quantification of Cache Side-Channels. In *International Conference on Computer Aided Verification*. 564–580.

[27] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *USENIX Workshop on Offensive Technologies (WOOT)*.

[28] Ruby B Lee, Peter Kwan, John P McGregor, Jeffrey Dwoskin, and Zhenghong Wang. 2005. Architecture for Protecting Critical Secrets in Microprocessors. In *ACM SIGARCH Computer Architecture News*, Vol. 33. IEEE Computer Society, 2–13.

[29] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium (USENIX)*.

[30] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. 2016. CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing. In *International Symposium on High Performance Computer Architecture (HPCA)*. 406–418.

[31] Fangfei Liu and Ruby B Lee. 2014. Random Fill Cache Architecture. In *International Symposium on Microarchitecture (MICRO)*. 203–215.

[32] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *Symposium on Security and Privacy (S&P)*. IEEE, 605–622.

[33] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. C5: Cross-Cores Cache Covert Channel. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 46–64.

[34] Junai Nomani and Jakub Szefer. 2015. Predicting Program Phases and Defending Against Side-Channel Attacks using Hardware Performance Counters. In *International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*.

[35] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: the Case of AES. In *Cryptographers' Track at the RSA Conference*. 1–20.

[36] Colin Percival. 2005. Cache Missing for Fun and Profit.

[37] Moinuddin K Qureshi. 2018. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *International Symposium on Microarchitecture (MICRO)*. 775–787.

[38] Majid Sabbagh, Yunsi Fei, Thomas Wahl, and A Adam Ding. 2018. SCADET: A Side-Channel Attack Detection Tool for Tracking Prime-Probe. In *International Conference on Computer-Aided Design (ICCAD)*. 1–8.

[39] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. NetSpectre: Read Arbitrary Memory over Network. In *European Symposium on Research in Computer Security (ESORICS)*. 279–299.

[40] Jakub Szefer. 2013. *Architectures for secure cloud computing servers*. Ph.D. Dissertation.

[41] Jakub Szefer. 2018. Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses. *Journal of Hardware and Systems Security* (13 September 2018).

[42] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. 2017. CacheD: Identifying Cache-Based Timing Channels in Production Software. In *USENIX Security Symposium (USENIX)*. 235–252.

[43] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C Myers, and G Edward Suh. 2016. SecDCP: Secure Dynamic Cache Partitioning for Efficient Timing Channel Protection. In *Design Automation Conference (DAC)*. 1–6.

[44] Zhenghong Wang and Ruby B Lee. 2007. New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks. In *ACM SIGARCH Computer Architecture News*, Vol. 35. ACM, 494–505.

[45] Zhenghong Wang and Ruby B Lee. 2008. A Novel Cache Architecture with Enhanced Performance and Security. In *International Symposium on Microarchitecture (MICRO)*. 83–93.

[46] Bernard L Welch. 1947. The Generalization of Student's Problem When Several Different Population Variances are Involved. *Biometrika* 34, 1/2 (1947), 28–35.

[47] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2019. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *USENIX Security Symposium (USENIX)*.

[48] Wenjie Xiong and Jakub Szefer. 2020. Leaking Information Through Cache LRU States. In *International Symposium on High-Performance Computer Architecture (HPCA)*.

[49] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *International Symposium on Microarchitecture (MICRO)*. 428–441.

[50] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. 2017. Secure Hierarchy-Aware Cache Replacement Policy (SHARP): Defending Against Cache-Based Side Channel Attacks. In *International Symposium on Computer Architecture (ISCA)*. 347–360.

[51] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. 2019. Attack Directories, Not Caches: Side Channel Attacks in a Non-inclusive World. In *USENIX Security Symposium (USENIX)*. 0.

[52] Yuval Yarom. 2016. Mastik: A micro-architectural side-channel toolkit. 16 (2016). https://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf

[53] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack.. In *USENIX Security Symposium (USENIX)*. 719–732.

[54] Danfeng Zhang, Aslan Askarov, and Andrew C Myers. 2012. Language-Based Control and Mitigation of Timing Channels. *ACM SIGPLAN Notices* 47, 6 (2012), 99–110.

[55] Danfeng Zhang, Yao Wang, G Edward Suh, and Andrew C Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 503–516.

[56] Tianwei Zhang and Ruby B Lee. 2014. New Models of Cache Architectures Characterizing Information Leakage from Cache Side Channels. In *Annual Computer Security Applications Conference (ACSAC)*. 96–105.

[57] Tianwei Zhang, Fangfei Liu, Si Chen, and Ruby B Lee. 2013. Side Channel Vulnerability Metrics: the Promise and the Pitfalls. In *International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. ACM, 2.