Analysis of Secure Caches and Timing-Based Side-Channel Attacks

Shuwen Deng, Wenjie Xiong and Jakub Szefer

Yale University, New Haven, CT, USA
{shuwen.deng,wenjie.xiong,jakub.szefer}@yale.edu

Abstract.
Many secure cache designs have been proposed in literature with the aim of mitigating different types of cache timing-based side-channel attacks. However, there has so far been no systematic analysis of how these secure cache designs can, or cannot, protect against different types of the timing-based attacks. To provide a means of analyzing the caches, this paper first presents a novel three-step modeling approach to exhaustively enumerate all the possible cache timing-based side-channel vulnerabilities. The model covers not only attacks that leverage cache accesses or flushes from the local processor core, but also attacks that leverage changes in the cache state due to the cache coherence protocol actions from remote cores. Moreover, both conventional attacks and speculative execution attacks are considered. With the list of all possible cache timing side-channel vulnerabilities derived from the three-step model, this work further analyzes each of the existing secure cache designs to show which types of timing-based side-channel vulnerabilities each secure cache can mitigate. Based on the security analysis of the existing secure cache designs, this paper further summaries different techniques gleaned from the secure cache designs and the technique's ability help mitigate different types of cache timing-based side-channel vulnerabilities.

Keywords: Secure Caches · Side-Channel Attacks · Security Analysis

1 Introduction

Research on timing-based side-channel attacks in computer processor caches has a long history predating the recent Spectre [1] and Meltdown [2] attacks, e.g., [3, 4, 5, 6, 7]. These attacks have shown the possibility to extract sensitive information via the timing-based side channels, and often the focus is on extracting cryptographic keys. In addition, due to the recent Spectre [1] and Meltdown [2] attacks, there is now renewed interested in timing-channels. Especially, the Spectre and Meltdown attacks consist of two parts: first, speculative execution is used to access some sensitive information; second, a timing-based channel is used to actually transfer the information to the attacker. Whether by itself, or combined with speculative execution, the timing-based channels in processors pose a threat to a system's security, and should be mitigated.

To address the threat of these cache timing-based attacks, different secure cache designs have been presented in academic literature to date [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]. The different secure processor caches are designed with different assumptions and often address only specific types of timing-based side-channel or covert-channel attacks. To help analyze the security of these designs, this work uses a three-step modeling approach to reason about all the possible timing-based vulnerabilities.

A three-step model has been recently proposed by us [25] in order to analyze cache timing-based side-channel attacks. The model considers cache timing-based side-channel vulnerabilities as a set of three "steps" or actions performed by either the attacker or the

victim, which can affect the states of the cache. In this work, our methodology from [25] is improved to better represent actions of the attacker and the victim. For each step, all possible states for a cache block are enumerated in terms of whether the operation is driven by the attacker or the victim, what memory range the data being operated on belongs to, and whether the state is changed because of a memory access or data invalidation operation (e.g., due to a cache coherence operation). To understand which possible three-step actions can lead to an attack, we further propose and develop a cache three-step simulator, and apply a set of reduction rules to derive a complete list of vulnerabilities by eliminating three-step combinations that do not map to an attack. Furthermore, we consider both normal and speculative execution for the memory operations and modeling of the cache attacks. Speculative execution has gotten increased attention due to recent Spectre [1] and Meltdown [2] attacks, which depend on timing channels to actually extract information – speculation alone is not enough for the attacks.

In the process of development of the improved three-step model, we have uncovered 43 types of timing-based side-channel vulnerabilities which have not been previously exploited (in addition, there are other 29 types that map to attacks already known in literature). These new types consider data that are operated on by either the victim or the attacker, and are in different data ranges, as well as different types of memory related operations.

Furthermore, this work analyzes whether the existing secure cache designs can prevent all the timing-based side-channel attacks (possibly with assistance of software assumptions that the authors discuss in their papers). We reviewed and analyzed 17 existing secure cache designs [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24] in terms of the security features and implementations. Based on the analysis, we summarize cache features that help improve security. Especially, the caches and the processor architectures should provide new features to let software explicitly label memory loads or stores of sensitive data, and differentiate them from normal loads and stores, so sensitive data can be efficiently identified and protected by the hardware. The caches can use partitioning to isolate the attacker and the victim and prevent the attacker from being able to set the victim's cache blocks into a known state, which is needed by many attacks. To mitigate attacks based on internal interference, the caches can use randomization to de-correlate the data that is accessed and the data that is put to the cache.

1.1 Contributions

The contributions of this work are as follows:

- A new formulation of the three-step model with new cache states and derivation of a new set of types for covering all the cache timing-based side-channel vulnerabilities.
- Inclusion of cache coherence issues into the three-step mode.
- Expansion of the three-step model to consider both normal and speculative execution.
- Design of reduction rules and cache three-step simulator to automatically derive the exhaustive list of all the three steps which map to effective vulnerabilities; and elimination of three-step patterns which do not map to a potential attack.
- Evaluation of 17 secure processor cache designs on how they can help prevent timing-based side-channel attacks and analysis of security features they used.

2 Cache Timing-Based Side-Channel Attacks and the Threat Model

Modern processor caches are known to be vulnerable to timing-based side-channel attacks. The timing of the memory accesses varies due to caches' operation. For example, a cache hit is fast while a cache miss is slow. Cache coherence protocol can also change the cache

states and affect the timing of the memory operations. Cache coherence may invalidate a cache line from a remote core, resulting in a cache miss in the local core, for example. Also, the timing of cache flush operations varies depending on whether the data to be flushed is in the cache or not. Flushing an address using $clflush$ with valid data in the cache is slow, while flushing an address not in the cache is fast. From these timing differences of memory related operations, the attacker can infer if a data at a specific memory address is in the cache or not, and thus potentially learn some information about the victim's secrets.

2.1 Threat Model

An attacker's objective is to retrieve victim's secret information using timing-based side channels in the cache. Specifically, we consider the situation where the victim accesses an address V_u and the address depends on some secret information. V_u is within some fixed range of physical memory locations x, which are known to the attacker. The goal of the attacker is to obtain the address V_u. We assume the attacker knows some of the source code of the victim, so he or she can further infer the secret from knowing the address V_u.

The attacker cannot directly access any data in the state machine of the cache logic, nor directly read the data of the victim, if the two are not sharing the same address space. The attacker can, however, observe its own timing or the timing of the victim process. And the attacker knows that the timing of the memory related operations depends on the cache states.

The attacker is able to force the victim to execute a specific function. For example, attacker can request victim to decrypt a specific piece of data, thus triggering the victim to execute a function that makes use of a secret key he or she wants to learn. The victim in the cache attacks can be user software, code in an enclave, operating system, or another virtual machine.

The processor microarchitecture and the operating system are assumed to be able to differentiate between the victim and the attacker in different processes by assigning different process IDs. If the victim and the attacker are in the same process, e.g., attacker is a malicious library, they will have the same process ID. The system software (e.g., operating system or hypervisor) is responsible for properly setting up virtual memory (page tables) and assigning IDs, which may be used by the hardware to identify different threads, processes or virtual machines. When analyzing secure cache designs, the system software is considered trusted and bug-free. For secure cache designs which add new instructions for security related operations, the victim process or management software is assumed to correctly use these instructions. During speculative execution, the cache state can be modified by the instructions executed speculatively, unless a processor cache architecture explicitly prevents or forbids certain speculative accesses.

2.2 Side Channels Versus Covert Channels

This work focuses on side channel. However, there are also covert channels. Covert channels use the same methods as side channels, but the attacker controls both the sender and the receiver side of the channel. All types of side-channel attacks are equally applicable to covert channels. But for brevity, we just use the term “side channels".

2.3 Hyperthreading Versus Timing-Slice Sharing

When the hyperthreading is supported in a system, the attacker and the victim are able to run on different threads in parallel instead of run once every time slice when no hyperthreading is used. Our model can be applied to both of the scenario since our model abstracts away how the sharing happens.

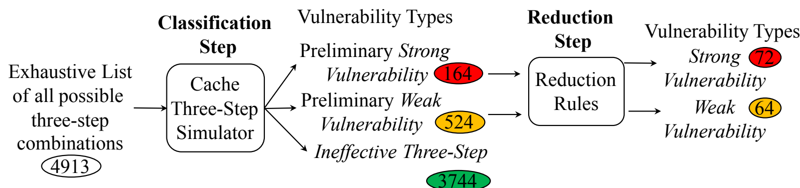

Figure 1: Processing procedures to derive the effective types of three-step timing-based side-channel vulnerabilities. Ovals refer to the number of vulnerabilities in each category.

3 Modeling of the Cache Timing-Based Side-Channel Vulnerabilities

This section explains how a three-step modeling approach is developed and used to model the behavior of the cache logic and how it can be used to enumerate all the possible cache timing-based side-channel vulnerabilities.

3.1 Introduction of the Three-Step Model

We have observed that all of the existing cache timing-based side-channel attacks can be modeled with three steps of memory related operations. Here, memory related operation refers to loads, stores or different flushes that can be done on the same core or different cores (cache coherence will then be triggered). The three-step model has three steps, as the name implies. In *Step* 1, a memory operation is performed, placing the cache in an initial state known to the attacker (e.g., a new piece of data at some address is put into the cache or the cache line is invalidated). Then, in *Step* 2, a second memory operation alters the state of the cache from the initial state. Finally, in *Step* 3, a final memory operation is performed, and the timing of the final operation reveals some information about the relationship among the addresses from *Step* 1, *Step* 2 and *Step* 3. For example, in Flush + Reload [26] attack, in *Step* 1, a cache line is flushed by the attacker. In *Step* 2, security critical data is accessed by, for example, victim's AES encryption operation. In *Step* 3, the same cache line as the one flushed in *Step* 1 will be accessed and the time of the access will be measured by the attacker. If the victim's secret dependent operation in *Step* 2 accesses the cache line, in *Step* 3 there will be a cache hit and fast timing of the memory operation will be observed, and the attacker learns the victim's secret address.

To model all the timing-based attacks, we write the three steps as: *Step* 1 $\rightsquigarrow$ *Step* 2 $\rightsquigarrow$ *Step* 3, which represents a sequence of steps taken by the attacker or the victim. To simplify the model, we focus on memory related operation affecting one single cache block (also called cache slot or cache entry). Cache block is the smallest unit of the cache. Since all the cache blocks are updated following the same cache state machine logic, it is sufficient to consider only one cache block.

When modeling the attacks, we propose that there are 17 possible states for a cache block. Table 1 lists all the 17 possible states of the cache block after each step in our three-step model.

3.2 Derivation of All Cache Timing-Based Vulnerabilities

With the 17 candidate states shown in Table 1 for each step, there are in total $17*17*17 = 4913$ combinations of three-steps. Not all of them can lead to real attacks. We developed a cache three-step simulator and a set of reduction rules to process all the three-step combinations and decide which ones can indicate a real attack. As is shown in Figure 1,

Table 1: The 17 possible states for a single cache block in our three-step vulnerability modeling procedure.

State	Description
V_u	A memory location belonging to the victim is accessed and is placed in the cache block by the victim (V). Attacker does not know u, but u is from a range x of memory locations, range which is known to the attacker. It may have the same index as A_a, V_a, $A_{a^{alias}}$, $V_{a^{alias}}$ and thus conflict with them in the cache block. The goal of the attacker is to learn the index or address of V_u.
A_a or V_a	The cache block contains a specific memory location a. The memory location is placed in the cache block due to a memory access by the attacker, A_a, or the victim, V_a. The attacker knows the address a, independent of whether the access was by the victim or the attacker themselves. The address a is within the range of sensitive locations x. The address a may or may not be the same as the address u.
$A_{a^{alias}}$ or $V_{a^{alias}}$	The cache block contains a memory address a^{alias}. The memory location is placed in the cache block due to 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 has the same address index and maps to the same cache block, i.e. it "aliases" to the same block.
A_d or V_d	The cache block contains a memory address d. The memory address is placed in the cache block due to a memory access by the attacker, A_d, or the victim, V_d. The address d is not within the range x.
A^{inv} or V^{inv}	The cache block previously might have contained some memory address but is now invalid. 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, e.g., this is a cache flush of the whole cache.
A_a^{inv} or V_a^{inv}	The cache block state can be anything except a in this cache block now. The address a is "removed" from the cache block by the attacker A_a^{inv} or the victim V_a^{inv}. E.g., by using a flush instruction such as $clflush$ that can flush specific address, or by causing certain cache coherence protocol events that force a to be removed from the cache block.
$A_{a^{alias}}^{inv}$ or $V_{a^{alias}}^{inv}$	The cache block state can be anything except a^{alias} in this cache block now. The address a^{alias} is "removed" from the cache block by the attacker $A_{a^{alias}}^{inv}$ or the victim $V_{a^{alias}}^{inv}$. E.g., by using a flush instruction such as $clflush$ that can flush specific address, or by causing certain cache coherence protocol events that force a^{alias} to be removed from the cache block.
A_d^{inv} or V_d^{inv}	The cache block state can be anything except d in this cache block now. The address d is "removed" from the cache block by the attacker A_d^{inv} or the victim V_d^{inv}. E.g., by using a flush instruction such as $clflush$ that can flush specific address, or by causing certain cache coherence protocol events that force d to be removed from the cache block.
V_u^{inv}	The cache block state can be anything except u being in the cache block. The data and its address is "removed" from the cache block by the victim V_u^{inv} as a result of cache block being invalidated, e.g., by using a flush instruction such as $clflush$, or by certain cache coherence protocol events that force u to be removed from the cache block. The attacker does not know u. Therefore, the attacker is not able to trigger this invalidation and A_u^{inv} does not exist in the model.
$\star$	Any data, or no data, can be in the cache block. The attacker has no knowledge of the memory address in this cache block.

the exhaustive list of the 4913 combinations will first be input to the cache three-step simulator, where the preliminary classification of vulnerabilities is derived. The effective vulnerabilities will then be sent as the input to the reduction rules to remove the redundant three-steps and obtain final list of vulnerabilities.

3.2.1 Cache Three-Step Simulator

We developed a cache three-step simulator that simulates the state of one cache block and derives the attacker's observations in the last step of the three-step patterns that it analyzes, for different possible u. Since u is in secure range x, the possible candidates of u for a cache block are a, a^{alias} and NIB (Not-In-Block). Here, NIB indicates the case that u does not have same index as a and a^{alias} and thus does not map to this cache block. In this way, the cache three-step simulator can derive the relationship between the victim's

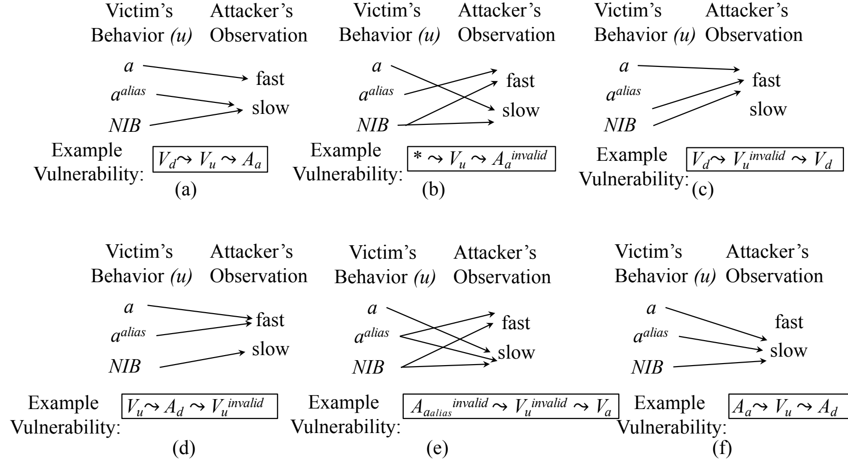

Figure 2: Examples of relations between victim's behavior (u) and attacker's observation for each vulnerability type ((a)(d) *Strong Vulnerability*, (b)(e) *Weak Vulnerability*, (c)(f) *Ineffective Three-Step*).

behavior (regarding the secret) and the attacker's observation (timing in the last step).

The cache three-step simulator judges whether a three-step pattern is a potential vulnerability by analyzing whether the attacker is able to observe different and unambiguous timing for different values of V_u. The simulator categorizes all the three-step patterns in the following way:

1. *Strong Vulnerability:* When a fast or slow timing is observed by the attacker, he or she is able to uniquely distinguish the value of u (either it maps to some known address or has the same index with some known address). a has the same address known to the attacker or not). In this case, the vulnerability has strong information leakage (i.e. attacker can directly obtain the value of u based on the observed timing). We categorize these vulnerabilities to be strong. E.g., for $V_d \rightsquigarrow V_u \rightsquigarrow A_a$ vulnerability shown in Figure 2a, if u maps to a, the attacker will always derive fast timing. If u is a^{alias} or NIB, slow timing will be observed. This indicates that the attacker is able to unambiguously infer the victim's behavior (u) from the timing observation.
2. *Weak Vulnerability:* Fast or slow timing corresponds to more than one possible value of u (e.g., a and a^{alias}). For these vulnerabilities, timing variation can still be observed due to different victim's behavior (V_u). However, the attacker cannot fully extract the address of V_u. E.g., for type $\star \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ shown in Figure 2b, when fast timing is observed, u possibly maps to a^{alias} or NIB (the reason for the possibility of u mapping to NIB is that due to the $\star$ in *Step* 1, the cache could have a hit and then A_a would result in a cache hit). On the other hand, when slow timing is observed, u possibly maps to a or NIB. This pattern leads to uncertain u guess about value of u based on timing observation.
3. *Ineffective Three-Step:* The remaining types are treated to be ineffective. E.g., for type $A_a \rightsquigarrow V_u \rightsquigarrow A_d$ shown in Figure 2f, no matter what the value of u is, attacker's observation is always slow timing.

After figuring out the type of all the three-step patterns, the cache three-step simulator will output effective (*Strong Vulnerability* or *Weak Vulnerability*) three-step patterns. Due to the space limit, we only list and analyze the *Strong* vulnerabilities in this paper. *Weak*

vulnerabilities are left for future work when channels with smaller channel capacities are desired to be analyzed.

3.2.2 Reduction Rules

We also have developed rules that can further reduce the output list of all the effective three-steps from the cache three-step simulator. Reduction's goal is to remove repeated vulnerabilities from the list of effective *Strong* or *Weak Vulnerability.* A script was developed that automatically applies below reduction rules to the output of the simulator to get the final list of vulnerabilities. A three-step combination will be eliminated if it satisfies one of the below rules:

1. Three-step patterns with two adjacent steps which are repeating, or which are both known to the attacker, can be eliminated, e.g., $A_d \rightsquigarrow A_a \rightsquigarrow V_u$ can be reduced to $A_a \rightsquigarrow V_u$, which is equivalent to $\star \rightsquigarrow A_a \rightsquigarrow V_u$. Therefore, $A_d \rightsquigarrow A_a \rightsquigarrow V_u$ is a repeat type of $\star \rightsquigarrow A_a \rightsquigarrow V_u$ and can be eliminated.
2. Step involving a known address a and an alias to that address a^{alias} gives the same information. Thus three step combinations which only differ in use of a or a^{alias} cannot represent different attacks, and only one combination needs to be considered. For example, $V_u \rightsquigarrow A_{a^{alias}} \rightsquigarrow V_u$ is a repeat type of $V_u \rightsquigarrow A_a \rightsquigarrow V_u$, and we will eliminate the first pattern.
3. V_u and V_u^{inv} in adjacent consecutive steps with each other can be eliminated, since the effective information is lost after executing these two steps.
4. V^{inv} or A^{inv} followed by address-specific invalidation, for example, V_u^{inv}, can be eliminated, since the memory location is repeatably flushed by the two steps.

3.2.3 Categorization of *Strong* Vulnerabilities

As is shown in Figure 1, after applying the rule reduction, the remaining *Strong* vulnerabilities are in total 72 types. Table 2 lists all the vulnerability types of which the last step is a memory access and Table 3 shows all the vulnerability types of which the last step is an invalidation-related operation. To ease the understanding of all the vulnerability types, we group the vulnerabilities based on attack strategies (left most column in Table 2 and Table 3), these strategies correspond to well-known names for the attacks, if such exist, otherwise we provide a new name.

The list of vulnerability types can be further collected into four simple macro types which cover one or more vulnerability types: internal interference miss-based (IM), internal interference hit-based (IH), external interference miss-based (EM), external interference hit-based (EH), as labeled in column 5 of Table 2 and Table 3. All the types of vulnerabilities that only involve the victim's behavior, V, in the states in *Step* 2 and *Step* 3 are called internal interference vulnerabilities (I). The remaining ones are called external interference (E). Some vulnerabilities allow the attacker to learn that the address of the victim accesses map to the set the attacker is attacking by observing *slow* timing due to a cache miss or $fast$ timing due to invalidation of data not in the cache[1]. We call these miss-based vulnerabilities (M). The remaining ones leverage observation of $fast$ timing due to a cache hit or *slow* timing due to an invalidation of an address that is currently valid in the cache, and are called hit-based vulnerabilities (H).

Many vulnerability types have been explored before. E.g., the Cache Collision attack [6] is effectively based on the Internal Collision, and it maps to types labeled (2) in the Attack column in Table 2 and Table 3. The types labeled **new** correspond to new attack not

[1]Invalidation is fast when the corresponding address to be invalidated does not exist in the cache and thus no operation is needed for the invalidation.

Table 2: The table shows all the cache timing-based cache side-channel vulnerabilities where the last step is a memory access related operation. The *Attack Strategy* column gives our common name for each set of one or more specific vulnerabilities that would be exploited in an attack in a similar manner. The *Vulnerability Type* column gives the three steps that define each vulnerability. For *Step* 3, *fast* indicates a cache hit must be observed, while *slow* indicates a cache miss must be observed. The *Macro Type* column proposes the categorization the vulnerability belongs to. “E” is for external interference vulnerabilities. “I” is for internal interference vulnerabilities. “M” is for miss-based vulnerabilities. “H” is for hit-based vulnerabilities. The *Attack* column shows if a type of vulnerability has been previously presented in literature.

Attack Strategy	Vulnerability Type			Macro Type	Attack
	Step 1	*Step* 2	*Step* 3		
Cache Internal Collision	A^{inv}	V_u	V_a (fast)	IH	(2)
	V^{inv}	V_u	V_a (fast)	IH	(2)
	A_d	V_u	V_a (fast)	IH	(2)
	V_d	V_u	V_a (fast)	IH	(2)
	$A_{a^{alias}}$	V_u	V_a (fast)	IH	(2)
	$V_{a^{alias}}$	V_u	V_a (fast)	IH	(2)
	A_a^{inv}	V_u	V_a (fast)	IH	(2)
	V_a^{inv}	V_u	V_a (fast)	IH	(2)
Flush + Reload	A_a^{inv}	V_u	A_a (fast)	EH	(5)
	V_a^{inv}	V_u	A_a (fast)	EH	(5)
	A^{inv}	V_u	A_a (fast)	EH	(5)
	V^{inv}	V_u	A_a (fast)	EH	(5)
	A_d	V_u	A_a (fast)	EH	(5)
	V_d	V_u	A_a (fast)	EH	(5)
	$A_{a^{alias}}$	V_u	A_a (fast)	EH	(5)
	$V_{a^{alias}}$	V_u	A_a (fast)	EH	(5)
Reload + Time	V_u^{inv}	A_a	V_u (fast)	EH	**new**
	V_u^{inv}	V_a	V_u (fast)	IH	**new**
Flush + Probe	A_a	V_u^{inv}	A_a (slow)	EM	(6)
	A_a	V_u^{inv}	V_a (slow)	IM	**new**
	V_a	V_u^{inv}	A_a (slow)	EM	**new**
	V_a	V_u^{inv}	V_a (slow)	IM	**new**
Evict + Time	V_u	A_d	V_u (slow)	EM	(1)
	V_u	A_a	V_u (slow)	EM	(1)
Prime + Probe	A_d	V_u	A_d (slow)	EM	(4)
	A_a	V_u	A_a (slow)	EM	(4)
Bernstein’s Attack	V_u	V_a	V_u (slow)	IM	(3)
	V_u	V_d	V_u (slow)	IM	(3)
	V_d	V_u	V_d (slow)	IM	(3)
	V_a	V_u	V_a (slow)	IM	(3)
Evict + Probe	V_d	V_u	A_d (slow)	EM	**new**
	V_a	V_u	A_a (slow)	EM	**new**
Prime + Time	A_d	V_u	V_d (slow)	IM	**new**
	A_a	V_u	V_a (slow)	IM	**new**
Flush + Time	V_u	A_a^{inv}	V_u (slow)	EM	**new**
	V_u	V_a^{inv}	V_u (slow)	IM	**new**

(1) Evict + Time attack [27].
(2) Cache Internal Collision attack [6].
(3) Bernstein’s attack [5].
(4) Prime + Probe attack [27, 4], Alias-driven attack [28].
(5) Flush + Reload attack [26, 29], Evict + Reload attack [30].
(6) SpectrePrime, MeltdownPrime attack [31].

previously discussed in literature. We believe these 43 are new attacks not previously analyzed nor known.

3.3 Soundness of the Three-Step Model

In this section we analyze the soundness of the three-step model to demonstrate that the three-step model can cover all possible cache timing-based side-channel vulnerabilities. If there is a vulnerability, it can always be reduced to a model that requires only three steps.

Let β denote the number of memory related operations in a vulnerability.

When $\beta = 1$, there is only one memory related operation, and it is not possible to create

Table 3: The table shows the second part of the timing-based cache side-channel vulnerabilities where the last step is an invalidation related operation. For *Step* 3, *fast* indicates no corresponding address of the data is invalidated, while *slow* indicates invalidation operation makes some data invalid, causing longer processing time.

Attack Strategy	Vulnerability Type			Macro Type	Attack
	Step 1	*Step* 2	*Step* 3		
Cache Internal Collision Invalidation	A^{inv}	V_u	V_a^{inv} (slow)	IH	**new**
	V^{inv}	V_u	V_a^{inv} (slow)	IH	**new**
	A_d	V_u	V_a^{inv} (slow)	IH	**new**
	V_d	V_u	V_a^{inv} (slow)	IH	**new**
	$A_{a_{alias}}$	V_u	V_a^{inv} (slow)	IH	**new**
	$V_{a_{alias}}$	V_u	V_a^{inv} (slow)	IH	**new**
Flush + Flush	A_a^{inv}	V_u	V_a^{inv} (slow)	IH	(1)
	V_a^{inv}	V_u	V_a^{inv} (slow)	IH	(1)
	A_a^{inv}	V_u	A_a^{inv} (slow)	EH	(1)
	V_a^{inv}	V_u	A_a^{inv} (slow)	EH	(1)
Flush + Reload Invalidation	A^{inv}	V_u	A_a^{inv} (slow)	EH	**new**
	V^{inv}	V_u	A_a^{inv} (slow)	EH	**new**
	A_d	V_u	A_a^{inv} (slow)	EH	**new**
	V_d	V_u	A_a^{inv} (slow)	EH	**new**
	$A_{a_{alias}}$	V_u	A_a^{inv} (slow)	EH	**new**
	$V_{a_{alias}}$	V_u	A_a^{inv} (slow)	EH	**new**
Reload + Time Invalidation	V_u^{inv}	A_a	V_u^{inv} (slow)	EH	**new**
	V_u^{inv}	V_a	V_u^{inv} (slow)	IH	**new**
Flush + Probe Invalidation	A_a	V_u^{inv}	A_a^{inv} (fast)	EM	**new**
	A_a	V_u^{inv}	V_a^{inv} (fast)	IM	**new**
	V_a	V_u^{inv}	A_a^{inv} (fast)	EM	**new**
	V_a	V_u^{inv}	V_a^{inv} (fast)	IM	**new**
Evict + Time Invalidation	V_u	A_d	V_u^{inv} (fast)	EM	**new**
	V_u	A_a	V_u^{inv} (fast)	EM	**new**
Prime + Probe Invalidation	A_d	V_u	A_d^{inv} (fast)	EM	**new**
	A_a	V_u	A_a^{inv} (fast)	EM	**new**
Bernstein's Invalidation Attack	V_u	V_a	V_u^{inv} (fast)	IM	**new**
	V_u	V_d	V_u^{inv} (fast)	IM	**new**
	V_d	V_u	V_d^{inv} (fast)	IM	**new**
	V_a	V_u	V_a^{inv} (fast)	IM	**new**
Evict + Probe Invalidation	V_d	V_u	A_d^{inv} (fast)	EM	**new**
	V_a	V_u	A_a^{inv} (fast)	EM	**new**
Prime + Time Invalidation	A_d	V_u	V_d^{inv} (fast)	IM	**new**
	A_a	V_u	V_a^{inv} (fast)	IM	**new**
Flush + Time Invalidation	V_u	A_a^{inv}	V_u^{inv} (fast)	EM	**new**
	V_u	V_a^{inv}	V_u^{inv} (fast)	IM	**new**

(1) Flush + Flush attack [32].

interference between memory related operations since two memory related operations are the minimum requirement for an interference. Furthermore, $\beta = 1$ corresponds to the three-step pattern with both *Step* 0 and *Step* 1 to be $\star$ since the cache state $\star$ gives no information. These types are not listed in Table 2 and Table 3, which show all the effective vulnerabilities. Therefore, attack cannot happen when $\beta = 1$.

When $\beta = 2$, this case corresponds to the three-step cases where *Step* 0 is $\star$. And there are no vulnerabilities listed in Table 2 and Table 3 with *Step* 0 to be $\star$. So $\beta! = 2$.

When $\beta = 3$, we exhaustively list all possible three-step memory related operations in Section 3.2 and we conclude that there are in total 72 types of *Strong* attacks, of which 46 are new compared to what is known in literature.

When $\beta > 3$, the pattern of memory related operations for a vulnerability can be broken down using the following rules:

- *Rule 1*: If there is a sub-pattern such as $\{\ ... \rightsquigarrow \star \rightsquigarrow ...\}$, the longer pattern can be divided into two separate patterns, where $\star$ is assigned as *Step* 0 of the second pattern. This is because $\star$ gives no timing information, and the attacker loses track of the cache state after $\star$. This rule should be recursively checked until there are no sub-patterns with a $\star$ in the middle.
- *Rule 2*: If the remaining memory related operations have a sub-pattern that has two

adjacent states both related to known addresses or both related to unknown address, the two adjacent states can be reduced to only one.

- For two unknown adjacent memory related operations (containing u), although u is unknown, both of the accesses target on the same u so can be reduced. E.g., $\{V_u \rightsquigarrow V_u\}$ can be reduced to $\{V_u\}$.
- For two known adjacent memory related operations, there are two cases.
  * In one case, two operations result in a deterministic state of the cache block, so this two steps can be reduced to only one step. E.g., $\{A_d \rightsquigarrow V_a\}$ can be reduced to $\{V_a\}$.
  * In another case, two operations can be separated in use for two access patterns. E.g., for $\{V_u \rightsquigarrow A^{inv}_{a^{alias}} \rightsquigarrow V^{inv}_a \rightsquigarrow V_u\}$ that contains adjacent known operations $\{A^{inv}_{a^{alias}} \rightsquigarrow V^{inv}_a\}$, it can be split into two patterns: $\{V_u \rightsquigarrow A^{inv}_{a^{alias}} \rightsquigarrow V_u\}$ and $\{V_u \rightsquigarrow V^{inv}_a \rightsquigarrow V_u\}$. This transformation not only allows the attacker to trigger less steps for a vulnerability, but also provides clearer information leakage for the attacker. For $\{V_u \rightsquigarrow A^{inv}_{a^{alias}} \rightsquigarrow V^{inv}_a \rightsquigarrow V_u\}$, a corresponding slow timing indicates u can either map to a or a^{alias}. On the other hand, slow timing observation of $\{V_u \rightsquigarrow A^{inv}_{a^{alias}} \rightsquigarrow V_u\}$ or $\{V_u \rightsquigarrow V^{inv}_a \rightsquigarrow V_u\}$ only indicates one possible mapping (a^{alias} or a, respectively) of u.

The *Rule 2* should be recursively checked until there are no two adjacent states both related to known addresses or both related to unknown address.

- *Rule 3*: After recursive reductions of *Rule 1* and *Rule 2*, either $\beta \leq 3$ holds, or one of the following two sub-pattern partitions still exists:

  - *known_memory_operation* $\rightsquigarrow$ *u_operation* $\rightsquigarrow$ *known_memory_operation*
  - *u_operation* $\rightsquigarrow$ *known_memory_operation* $\rightsquigarrow$ *u_operation*

  Here *u_operation* refers to V_u or V^{inv}_u, *known_memory_operation* refers to all the possible states shown in Table 1 except $\star$, V_u and V^{inv}_u. Effective ones of these two sub-patterns map to known vulnerabilities listed in Table 2 and Table 3. *Rule 3* will check every adjacent three-step partition of the patterns that are input to *Rule 3*. If a pattern contains a partition that maps to one of the effective patterns, the corresponding pattern already represents a vulnerability, which proves the effectiveness of the original long pattern. *Rule 3* will output corresponding effective partition(s) if there are some. Otherwise, the original long pattern is ineffective.

We make use of the three *Rule*s in the way shown in Algorithm 1 to reduce β-step ($\beta > 3$) pattern to be within three steps and demonstrate that the β-step pattern can be mapped to three-step vulnerabilities if it is effective.

In conclusion, the three-step model can model all possible timing-based cache side-channel vulnerability with any β steps. Attacks which are represented by more than three steps can be always reduced to one (or more) vulnerabilities from our three-step model; and thus, using more than three step is not necessary.

4 Secure Caches

Having explained the three-step model, we now explore the various secure caches which have been presented in literature to date [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]. Latter, in Section 5 we will apply the three-step model to check if the secure caches can defend some or all of the vulnerabilities in our model.

Algorithm 1 β-Step ($\beta > 3$) Pattern Reduction

Input: β: number of steps of the pattern
step_list: a two-dimensional dynamic-size array. *step_list*[0] contains the states of each step of the original pattern in order. *step_list*[1], *step_list*[2], ... is empty initially.
Output: *reduce_list*: reduced effective vulnerability pattern(s) array. It will be an empty list if the original pattern does not correspond to an effective vulnerability.

```
1: while step_list.contain(⋆) and ⋆.index not 0 do
2:    Rule_1 (step_list)
3: end while
4: while step_list.contain(adjacent known_memory_operation or u_operation) do
5:    Rule_2 (step_list)
6: end while
7: reduce_list = Rule_3 (step_list)
8: return reduce_list
```

This section gives brief overview of the 17 secure cache designs that have been presented in academic literature in the last 14 years. To the best of our knowledge, these cover all the secure cache designs proposed to date. Most of the designs have been realized in functional simulation, e.g., [14, 19]. Some have been realized in FPGA, e.g., [23], and a few have been realized in real ASIC hardware, e.g., [33]. No specific secure caches have been implemented in commercial processors to the best of our knowledge, however, CATalyst [18] leverages Intel's CAT (Cache Allocation Technology) technology available today in Intel Xeon E5 2618L v3 processors, and could be deployed today.

When the secure cache description in the cited papers did not mention the issue of using flush or cache coherence, we assume the victim or the attacker cannot invalidate each other's cache blocks by using *clflush* instructions or through cache coherence protocol operations; but they can flush or use cache coherence to invalidate their own cache lines. The victim and the attacker also cannot invalidate protected or locked data. Further, if the authors specified any specific assumptions (mainly about the software), we list the assumption as part of the description of the cache. What's more, when the level of cache hierarchy was unspecified, we assume the secure caches' features can be applied to all the microarchitecture in the cache hierarchy, including L1 cache, L2 cache and Last Level Cache (LLC). If the inclusivity of the caches was not specified, we assume they target inclusive caches. Following the below descriptions of each secure cache design, the analysis of the secure caches is given in Section 5.

SP* cache [15, 34][2] uses partitioning techniques to statically partition the cache ways into *High* and *Low* partition for the victim and the attacker according to their different process IDs. The victim typically belongs to *High* security and attacker belongs to *Low* security. Victim's memory accesses cannot modify *Low* partition (assigned to processes such as the attacker), while the attacker's memory accesses cannot modify *High* partition (assigned to the victim). However, the memory accesses of both the victim and the attacker can result in a hit in either *Low* or *High* partition if the data is in the cache.

SecVerilog cache [9, 8] statically partitions cache blocks between security levels L (*Low*) and H (*High*). Each instruction in the source code for programs using SecVerilog cache needs to include a *timing label* which effectively represents whether the data being accessed by that instruction is *Low* or *High* based on the code and this *timing label* can be similar to a process ID that differentiates attacker's (*Low*) instructions from victim's (*High*) instructions. The cache is designed such that operations in the *High* partition cannot affect timing of operations in the *Low* partition. For a cache miss due to *Low* instructions, when

[2] Two existing papers give slightly different definitions for an "SP" cache, thus we selected to define a new cache, the SP* cache, that combines secure cache features of the Secret-Protecting cache from [15] with secure cache features of the Static-Partitioned cache from [34].

the data is in the *High* partition, it will behave as a cache miss, and the data will be moved from the *High* to the *Low* partition to preserve consistency. However, *High* instructions are able to result in a cache hit in both *High* and *Low* partitions, if the data is already in the cache.

SecDCP cache [14] builds on the SecVerilog cache and uses partitioning idea from the original SecVerilog cache, but the partitioning is dynamic. It can support at least two security classes H (*High*) and L (*Low*), and configurations with more security classes are possible. They use the percentage of cache misses for L instructions that was reduced (increased) when L's partition size was increased (reduced) by one cache way to adjust the number of ways of the cache assigned to the *Low* partition. When adjusting number of ways in the cache dedicated to each partition, if L's partition size decreases, the process ID is checked and L blocks are flushed before the way is reallocated to H. On the other hand, if L's partition size increases, H blocks in the adjusted cache way remain unmodified so as to not add more performance overhead, and they will eventually be evicted by L's memory accesses. However, the feature of not flushing *High* partition data during way adjustment may leak timing information to the attacker.

NoMo cache [17] dynamically partitions the cache ways among the currently "active" simultaneous multithreading (SMT) threads. Each thread is exclusively reserved Y blocks in each cache set, where Y is within the range of [0, $\lfloor \frac{N}{M} \rfloor$], where N is the number of ways and M is the number of SMT threads. NoMo-0 equals to traditional set associative cache while NoMo-$\lfloor \frac{N}{M} \rfloor$ partitions cache evenly for the different threads and there are no non-reserved ways. The number of Y assigned to each thread is adjusted based on its activeness. When adjusting number of blocks assigned to a thread, Y blocks are invalidated for cache sets to protect timing leakage. Eviction is not allowed within each thread's own reserved ways while it is possible for the shared ways. Therefore, to avoid eviction caused by the unreserved ways, we assume NoMo-$\lfloor \frac{N}{M} \rfloor$ is used to fully partition the cache. When the attacker and the victim share the same library, there will be a cache hit if accessing the shared data, and the normal cache hit policy holds to guarantee the cache coherence.

SHARP cache [16] uses both partitioning and randomization techniques to prevent victim's data from being evicted or flushed by other malicious processes and it targets on the inclusive caches. Each cache block is augmented with the core valid bits (CVB) to indicate which private cache (process) it belongs to (similar to the Process ID), where CVB stores a bitmap and i-th bit in the bitmap is set if the line is present in i-th core's private cache. Cache hit is allowed among different processes' data. When there is cache miss and data needs to be evicted, data not belonging to any current processes will be evicted first. If there is no such data, data belonging to the same process will be evicted. If there is no existing data in the cache that is in the same process, a random data in the cache set will be evicted. This random eviction will generate an interrupt to the OS to notify it of a suspicious activity. For pages that are read-only or executable, SHARP cache disallows flushing using $clflush$ in user mode. However, invalidating victim's blocks by using cache coherence protocol is still possible.

Sanctum cache [13] focuses on isolation of enclaves (equivalent to Trusted Software Module in other designs) from each other and the operating system (OS). In terms of caches, they implements security features for L1 cache, TLB and LLC. Cache isolation of LLC is achieved by assigning each enclave or OS to different DRAM address regions. It uses page-coloring-based cache partitioning scheme [35, 36] and a software security monitor that ensures per-core isolation between OS and enclaves. For L1 cache and TLB, when there is a transition between enclave and non-enclave mode, the security monitor will flush the core-private caches to achieve isolation. Normal flushes triggered by the enclave or the OS can only be done within enclave or not within enclave code. Also, timing-based side-channel attacks exploiting cache coherence are explicitly not prevented, thus behavior on cache coherence operations is not defined. This cache listed extra software assumptions

as follows:

Assumption 1. Software security monitor guarantees that victim and attacker process cannot share the same cache blocks. It uses page coloring [35, 36] to ensure that victim and attacker's memory is never mapped to the same cache blocks for the LLC.

Assumption 2. The software runs on a system with a single processor core where victim and attacker alternate execution, but can never run truly in parallel. Moreover, security critical data is always flushed by the security monitor when program execution switches away from the victim program for the L1 cache and the TLB.

MI6 cache [23] is part of the memory hierarchy of the MI6 processor, which combines Sanctum [13] cache's security feature with disabling speculation during the speculative execution of memory related operations. During normal processor execution, for L1 caches and TLB, the corresponding states will be flushed across context switches between software threads. For the LLC, set partitioning is used to divide DRAM into contiguous regions. And cache sets are guaranteed to be strictly partitioned (two DRAM regions cannot map to the same cache set). Each enclave is only able to access its own partition. Speculation is simply disabled when enclave interacts with the outside world because of small performance influence based on the rare cases of speculation. This cache listed extra software assumptions as follows:

Assumption 1. Software security monitor guarantees that the victim and the attacker process cannot share the same cache blocks. It uses page coloring [35, 36] to ensure that victim's and attacker's memory are never mapped to the same cache blocks for the LLC.

Assumption 2. The software runs on a system with a single processor core where victim and attacker alternate execution, but can never run truly in parallel. Moreover, security critical data is always flushed by the security monitor when program execution switches away from the victim program for the L1 cache and the TLB.

Assumption 3. When an enclave is interacting with the outside environment, the corresponding speculation is disabled by the software.

InvisiSpec cache [22] is able to make speculation invisible in the data cache hierarchy. Before a *visibility point* shows up, when all of its prior control flow instructions resolve, unsafe speculative loads (USL) will be put into a speculative buffer (SB) without modifying any cache states. When reaching the visibility point, there are two cases. In one case, the USL and successive instructions will be possibly squashed because of mismatch of data in the SB and the up-to-date values in the cache. In another case, the core receives possible invalidation from the OS before checking of memory consistency model and no comparison is needed. When speculative execution happens, the hardware puts the data into SB, as to identify visibility point for dealing with final state transition of the speculative execution. InvisiSpec cache targets on Spectre-like attacks and futuristic attacks. However, InvisiSpec cache is vulnerable to all non-speculative side channels.

CATalyst cache [18] uses partitioning, especially Cache Allocation Technology (CAT) [37] available in the LLC of some Intel processors. CAT allocates up to 4 different Classes of Services (CoS) for separate cache ways so that replacement of cache blocks is only allowed within a certain CoS. CATalyst first uses CAT mechanism to partition caches into secure and non-secure parts (non-secure parts may map to 3 CoS while secure parts map to 1 CoS). Secure pages are assigned to virtual machines (VMs) at a granularity of a page, and not shared by more than one VM. Here, attacker and victim reside in different VMs. Combined with CAT technology and pseudo-locking mechanism which pins certain page frames managed by software, CATalyst guarantees that malicious code cannot evict secure pages. CATalyst implicitly performs preloading by remapping security-critical code or data to secure pages. Flushes can only be done within each VM. And cache coherence is achieved by assigning secure pages to only one processor and not sharing pages among VMs. This cache listed extra software assumptions as follows:

Assumption 1. Security critical data is always preloaded into the cache at the beginning

of the whole program execution.

Assumption 2. Security critical data is always able to fit within the secure partition of the cache. I.e. all data in the range x can fit in the secure partition.

Assumption 3. The victim process and the attacker process cannot share the same memory space.

Assumption 4. Use pseudo-locking mechanism by software to make sure that victim and attacker process cannot share the same cache blocks.

Assumption 5. Secure pages are reloaded immediately after the flush, which is done by the virtual machine monitor (VMM) to make sure all the secure pages are still pinned in the secure partition.

DAWG cache [21] (Dynamically Allocated Way Guard) partitions the cache by cache ways and provides full isolation for hits, misses and metadata updates across different protection domains (between the attacker and the victim). DAWG cache is partitioned for the attacker and the victim and each of them keep their own different *domain_id* (which is similar to process ID used in general caches). Each *domain_id* has its own bit fields, one is called *policy_fillmap*, for masking fills and selecting the victim to replace, another is called *policy_hitmap*, for masking hit ways. Only both the tag and the *domain_id* are the same will a cache hit happen. Therefore, DAWG allows read-only cache lines to be replicated across ways for different protection domain. For a cache miss, the victim can only be chosen within the ways belonging to the same *domain_id*, recorded by the *policy_fillmap*. Consistently, the replacement policy is updated with the victim selection and the metadata derived from the *policy_fillmap* for different domains is updated as well. The paper also proposes the idea to dynamically partitions the cache ways following the system's workload changes but does not actually implement it.

RIC cache [20] (Relaxed Inclusion Caches) proposes a low-complexity cache to defend against eviction-based timing-based side-channel attacks on the LLC. Normally for an inclusive cache, if the data R is in the LLC, it is also in the higher level cache, and eviction of the R in the LLC will cause the same data in the higher level cache, e.g., L1 cache to be invalidated, making eviction-based attacks in the higher level cache possible (e.g., attacker is able to evict victim's security critical cache line). For RIC, each cache line is extended with a single bit to set the relaxed inclusion. Once the relaxed inclusion is set for that cache line, the corresponding LLC line eviction will not cause the same line in the higher-level cache to be invalidated. Two kinds of data will be set relaxed inclusion bit: *read only data* and *thread private data* when they are loaded into the cache. These two kinds of data are claimed by the paper to cover all the critical data for ciphers. Therefore, RIC will not prevent writable in-private critical data, which is currently not found in any ciphers. Apart from that, RIC requires flushing for the corresponding cache lines in the cases that the RIC bits are modified or for thread migration events to avoid the timing leakage during transition time.

PL cache [10] provides isolation by partitioning cache based on cache blocks. It extends each cache block with a process ID and a lock status bit. The process ID and the lock status bits are controlled by the extended load and store instructions (*ld.lock*/*ld.unlock* and *st.lock*/*st.unlock*) which allow the programmer and compiler to set or reset the lock bit through use of the right load or store instruction. In terms of cache replacement policy, for a cache hit, PL cache will perform the normal cache hit handling procedure and the instructions with locking or unlocking capability can update the process ID and the lock status bits while the hit is processed. When there is a cache miss, locked data cannot be evicted by data that is not locked and locked data among different processes cannot be evicted by each other. In this case, the new data will be either loaded or stored without caching. In other cases, data eviction is possible. This cache listed extra software assumption as follows:

Assumption 1. Security critical data is always preloaded into the cache at the beginning

of the whole program execution.

RP cache [10] uses randomization to de-correlate the memory address accessing and timing of the cache. For each block of RP cache, there is a process ID and one protection bit P set to indicate if this cache block needs to be protected or not. A permutation table (PT) stores each cache set's pre-computed permuted set number and the number of tables depends on number of protected processes. For memory access operations, cache hit needs both process ID and address to be the same. When a cache miss happens to data D of a cache set S, if the to-be-evicted data and to-be-brought-in data belong to the same process but have different protection bit, an arbitrary data of a random cache set S' will be evicted and D will be accessed without caching. If they belong to different processes, D will be stored in an evicted cache block of S' and mapping of S and S' will be swapped as well. Otherwise, the normal replacement policy is executed.

Newcache cache [11, 33] dynamically randomizes memory-to-cache mapping. It introduced a ReMapping Table (RMT), and the mapping between memory addresses and this RMT is as in a direct mapped cache, while the mapping between the RMT and actual cache is fully associative. The index bits of memory address are used to look up entries in the RMT to find the cache block that should be accessed. It stores the most useful cache lines rather than hold a fixed set of cache lines. This index stored in RMT combined with the process ID is used to look up the actual cache where each cache line is associated with its real index and process ID. Each cache block is also associated with a protection bit (P) to indicate if it is security critical. For cache replacement policy, it is very similar to RP cache. Cache hit needs both process ID and address to be the same. When cache miss happens to data D, an arbitrary data will be evicted and D will be accessed without caching if they belong to the same process but either one of their protection bit is set. If the evicted data and brought-in data have different process IDs, D will randomly replace a cache line since it is fully associative in the actual cache. Otherwise, the normal replacement policy for direct mapped cache is executed.

Random Fill cache [12] de-correlates cache fills with the memory access using random filling technique. New instructions used by applications in Random Fill cache can control if the requested data belongs to a normal request or a random fill request. Cache hits are processed as in normal cache. For the security critical data accesses of the victim, a $Nofill\ request$ is executed and the requested data access will be performed without caching. Meanwhile, on a *Random Fill request*, arbitrary data, from the range of addresses, will be brought into the cache. In the paper [12], the authors show that random fill of spatially near data does not hurt performance. For other processes' memory accesses and normal victim's memory accesses, $Normal\ request$ will be used to achieve normal replacement policy. Victim and attacker are able to remove victim's own security critical data including using $clflush$ instructions or cache coherence protocol since the flush will not influence timing-based side-channel attack prevention (the random filling technique is used for this).

CEASER cache [24] is able to mitigate conflict-based LLC timing-based side-channel attacks using address encryption and dynamic remapping. CEASER cache does not differentiate whom the address belongs to and whether the address is security critical. When memory access tries to modify the cache state, the address will first be encrypted using Low-Latency BlockCipher (LLBC) [38], which not only randomizes the cache set it maps, but also scatters the original, possibly ordered and location-intensive addresses to different cache sets, decreasing the probability of conflict misses. The encryption and decryption can be done within two cycles using LLBC. Furthermore, the encryption key will be periodically changed to avoid key reconstruction. The periodic re-keying will cause the address remapping to dynamically change.

Non Deterministic cache [19] uses cache access delay to randomize the relation between cache block access and cache access timing. There is no differentiation of data caching between different process ID or whether the data is secure or not. A per-cache-block

counter records the interval of its data activeness, and is increased on each global counter clock tick when the data is untouched. When the counter reaches a predefined value, the corresponding cache line will be invalidated. Non Deterministic Cache randomly sets the local counters' initial value to be a number less than the maximum value of the global counter to change the cache delay. Cache delay interval controlled by this non-deterministic execution can lead to different cache hit and miss statistics because the invalidation is determined by the randomized counter of each cache line, and therefore de-correlates any cache access time from the address being accessed. However, the performance degradation is tremendous.

5 Analysis of the Secure Caches

In this section, we evaluate the effectiveness of the 17 secure caches [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]. We analyze how well the different caches can protect against the 72 types of vulnerabilities defined in Table 2 and Table 3, which cover all the possible *Strong* (according to the definition in Seciton 3) cache timing-based side-channel vulnerabilities. Following the analysis, we conclude on what types of secure caches and features are best suited for defending different types of timing-based attacks.

5.1 Effectiveness of the Secure Caches Against Timing-Based Attacks

Table 4 and Table 5 list the result of our analysis of which caches can prevent which types of attacks. Some caches are able to prevent certain vulnerabilities, denoted by a checkmark, ✓, and green color in the table. For example, SP* cache can defend against $V_u \rightsquigarrow A_d \rightsquigarrow V_u$ (slow) (one type of Evict + Time [27]) vulnerability. For some other caches and vulnerabilities, the cache is not able to prevent the vulnerabilities and it is indicated by $\times$ and red color. For example, SecDCP cache cannot defend against $V_u \rightsquigarrow V_a \rightsquigarrow V_u$ (slow) (one type of Bernstein's Attack [5]) vulnerability.

Each cache is analyzed for each type of vulnerability listed in Table 2 and Table 3. A cache is judged to be able to prevent a type of cache timing-based side-channel vulnerability in three cases:

1. The timing of the last step in a vulnerability is always constant and the attacker can never observe fast and slow timing difference for the given set of three steps. For instance, on a regular set-associative cache, the $V_d \rightsquigarrow V_u \rightsquigarrow A_a$ (fast) (one type of Flush + Reload [26]) vulnerability will allow the attacker to know that address a maps to secret u when the attacker observes fast timing, compared with observing slow timing in the other cases. However, RP cache will make the timing of the last step to be always slow because RP cache does not allow data of different processes to derive cache hit between each other.
2. The timing of last step is randomized and cannot have original corresponding relation between victim's behavior and attacker's observation. For instance, $A_d \rightsquigarrow V_u \rightsquigarrow A_d^{inv}$ (fast) (one type of Prime + Probe Invalidation) vulnerability when executed on a normal set-associative cache will allow the attacker to know that the address d has the same index with secret u when observing fast timing, compared with slow timing in the other cases. However, when executing this attacks on the Random Fill cache, a slow timing will not determine that u and d have the same index as the secret, since in Random Fill cache u would be accessed without caching and another random data u would be cached in *Step* 2 instead.

Table 4: Analysis of secure caches' ability to protect against different types of timing-based attacks. For each of the 72 types of attacks, the table shows if the cache can defend the attack or not. Single ✓ in a green cell means this cache can prevent the corresponding vulnerability. A × in a red cell means this cache cannot prevent this vulnerability. Furthermore, for each cache, we analyze normal execution (left column under the cache name) and speculative execution (right column under the cache name).

Type	Vulnerability	Set Associative Cache		SP* Cache		Sec-Verilog Cache		SecDCP Cache		NoMo Cache		SHARP Cache[6]		Sanctum Cache[2]		MI6 Cache		InvisiSpec Cache[9]		CATalyst Cache[2,5]		DAWG Cache		RIC[8]		PL Cache		RP Cache		Newcache		Random Fill Cache		CEASER Cache		Non Deterministic Cache		Cache Disabled	
Cache Internal Collision	$A^{inv} \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10]	✓[10]	✓[10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	×	×	×	×	✓	✓	×	×	✓	✓	✓	✓
	$V^{inv} \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	×	×	×	×	✓	✓	×	×	✓	✓	✓	✓
	$A_a^{inv} \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10]	✓[10]	✓[10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	×	×	×	×	✓	✓	×	×	✓	✓	✓	✓
	$V_a^{inv} \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	×	×	×	×	✓	✓	×	×	✓	✓	✓	✓
	$A_d \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10]	✓[10]	✓[10]	✓	×	✓	✓[8]	✓[8]	✓	✓	×	×	✓	✓	×	×	×	×	✓	✓	×	×	✓	✓	✓	✓
	$V_d \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓[8]	✓[8]	×	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$A_{a^{alias}} \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10]	✓[10]	✓[10]	✓	×	✓	✓[10]	✓[10]	✓	✓	×	×	✓	✓	×	×	×	×	✓	✓	×	×	✓	✓	✓	✓
	$V_{a^{alias}} \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	×	×	×	×	✓	✓	✓	✓	×	×	✓	✓	✓	✓
Flush + Reload	$A^{inv} \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$V^{inv} \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	×	×	×	×	✓	✓	✓	✓	×	×	×	×	✓[10]	✓[4,10]	✓[10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$A_a^{inv} \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$V_a^{inv} \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	×	×	×	×	✓	✓	✓	✓	×	×	×	×	✓[10]	✓[4,10]	✓[10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$A_d \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓[8]	✓[8]	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$V_d \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	×	×	×	×	✓	✓	✓	✓	×	×	×	×	✓[10]	✓[4,10]	✓[10]	✓	×	✓	✓[8]	✓[8]	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$A_{a^{alias}} \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓[10]	✓[10]	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$V_{a^{alias}} \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	×	×	×	×	✓	✓	✓	✓	×	×	×	×	✓[10]	✓[4,10]	✓[10]	✓	×	✓	✓[10]	✓[10]	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
Reload + Time	$V_u^{inv} \rightsquigarrow A_a \rightsquigarrow V_u$ (fast)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$V_u^{inv} \rightsquigarrow V_a \rightsquigarrow V_u$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
Flush + Probe	$A_a \rightsquigarrow V_u^{inv} \rightsquigarrow A_a$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$A_a \rightsquigarrow V_u^{inv} \rightsquigarrow V_a$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$V_a \rightsquigarrow V_u^{inv} \rightsquigarrow A_a$ (slow)	×	×	×	×	✓	✓	×	×	×	×	×	×	✓[4]	✓[4]	✓[4]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$V_a \rightsquigarrow V_u^{inv} \rightsquigarrow V_a$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
Evict + Time	$V_u \rightsquigarrow A_d \rightsquigarrow V_u$ (slow)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓[8]	✓[8]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓
	$V_u \rightsquigarrow A_a \rightsquigarrow V_u$ (slow)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓[8]	✓[8]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Prime + Probe	$A_d \rightsquigarrow V_u \rightsquigarrow A_d$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓[8]	✓[8]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	$A_a \rightsquigarrow V_u \rightsquigarrow A_a$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓[10]	✓[10]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Bernstein's Attack	$V_u \rightsquigarrow V_a \rightsquigarrow V_u$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	$V_u \rightsquigarrow V_d \rightsquigarrow V_u$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓[8]	✓[8]	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓
	$V_d \rightsquigarrow V_u \rightsquigarrow V_d$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓[8]	✓[8]	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	$V_a \rightsquigarrow V_u \rightsquigarrow V_a$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Evict + Probe	$V_d \rightsquigarrow V_u \rightsquigarrow A_d$ (slow)	×	×	×	×	✓	✓	×	×	×	×	×	×	✓[10]	✓[4,10]	✓[10]	✓	×	✓	✓[8]	✓[8]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	$V_a \rightsquigarrow V_u \rightsquigarrow A_a$ (slow)	×	×	×	×	✓	✓	×	×	×	×	×	×	✓[10]	✓[4,10]	✓[10]	✓	×	✓	✓[10]	✓[10]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Prime + Time	$A_d \rightsquigarrow V_u \rightsquigarrow V_d$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10]	✓[10]	✓[10]	✓	×	✓	✓[8]	✓[8]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	$A_a \rightsquigarrow V_u \rightsquigarrow V_a$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10]	✓[10]	✓[10]	✓	×	✓	✓[10]	✓[10]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Flush + Time	$V_u \rightsquigarrow A_a^{inv} \rightsquigarrow V_u$ (slow)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓	✓	✓[10]	✓[4,10]	✓[10]	✓	×	✓	✓[8]	✓[8]	✓	✓	×	×	✓	✓	✓	✓	✓	✓	×	×	×	×	✓	✓	✓	✓
	$V_u \rightsquigarrow V_a^{inv} \rightsquigarrow V_u$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓[8]	✓[8]	×	×	×	×	✓	✓	✓	✓	✓	✓	×	×	×	×	✓	✓	✓	✓

[1] [1] Dynamic adjustment of ways for different threads is assumed to be properly used according to the running program's cache usage. [2] Some software assumptions listed in the entries in this column have been implemented by the cache's related software. [3] Flush is disabled, but cache coherence might be used to do the data removal. [4] For L1 cache and TLB, flushing is done during context switch. [5] The techniques are implemented in L1 cache, TLB and last-level cache which consist of the whole cache hierarchy, where L1 cache and TLB require software flush protection and the last-level cache can be achieved by simple hardware partitioning. To protect all levels of caches, the software assumptions need to be added. [6] The technique is now only implemented in last-level cache. [7] The technique now only targets shared cache. [8] The technique only targets on inclusion last-level cache. [9] The technique targets on data cache hierarchy. [10] For last-level cache, cache is partitioned between the victim and the attacker.

Table 5: Existing secure caches' protection against all possible timing-based side-channel vulnerabilities with last step to be invalidation related operations. Single ✓ in a green cell means this cache can prevent the corresponding vulnerability. A × in a red cell means this cache cannot prevent this vulnerability. Left and right column of the cache represent normal and speculative execution, respectively.

Type	Vulnerability	Set Associative Cache		SP* Cache		Sec-Verilog Cache		SecDCP Cache		NoMo Cache		SHARP Cache[6]		Sanctum Cache[2]		MI6 Cache		InvisiSpec Cache[9]		CATalyst Cache[2,5]		DAWG Cache		RIC[8]		PL Cache		RP Cache		Newcache		Random Fill Cache		CEASER Cache		Non Deterministic Cache		Cache Disabled	
Cache Internal Collision Invalidation	$A^{inv} \rightsquigarrow V_u \rightsquigarrow V_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10]	✓[10]	✓[10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$V^{inv} \rightsquigarrow V_u \rightsquigarrow V_a^{inv}$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$A_d \rightsquigarrow V_u \rightsquigarrow V_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[10]	✓[10]	✓[10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$V_d \rightsquigarrow V_u \rightsquigarrow V_a^{inv}$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$A_{a^{alias}} \rightsquigarrow V_u \rightsquigarrow V_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[10]	✓[10]	✓[10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$V_{a^{alias}} \rightsquigarrow V_u \rightsquigarrow V_a^{inv}$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
Flush + Flush	$A_a^{inv} \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$V_a^{inv} \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$A_a^{inv} \rightsquigarrow V_u \rightsquigarrow V_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10]	✓[10]	✓[10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$V_a^{inv} \rightsquigarrow V_u \rightsquigarrow V_a^{inv}$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
Flush + Reload Invalidation	$A^{inv} \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$V^{inv} \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$A_d \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$V_d \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$A_{a^{alias}} \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$V_{a^{alias}} \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
Reload + Time Invalidation	$V_u^{inv} \rightsquigarrow A_a \rightsquigarrow V_u^{inv}$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$V_u^{inv} \rightsquigarrow V_a \rightsquigarrow V_u^{inv}$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
Flush + Probe Invalidation	$A_a \rightsquigarrow V_u^{inv} \rightsquigarrow A_a^{inv}$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$A_a \rightsquigarrow V_u^{inv} \rightsquigarrow V_a^{inv}$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$V_a \rightsquigarrow V_u^{inv} \rightsquigarrow A_a^{inv}$ (fast)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓	✓	✓[4]	✓[4]	✓[4]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
	$V_a \rightsquigarrow V_u^{inv} \rightsquigarrow V_a^{inv}$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓
Evict + Time Invalidation	$V_u \rightsquigarrow A_d \rightsquigarrow V_u^{inv}$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓
	$V_u \rightsquigarrow A_a \rightsquigarrow V_u^{inv}$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Prime + Probe Invalidation	$A_d \rightsquigarrow V_u \rightsquigarrow A_d^{inv}$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓[8]	✓[8]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	$A_a \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Bernstein's Invalidation Attack	$V_u \rightsquigarrow V_a \rightsquigarrow V_u^{inv}$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	$V_u \rightsquigarrow V_d \rightsquigarrow V_u^{inv}$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓
	$V_d \rightsquigarrow V_u \rightsquigarrow V_d^{inv}$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓[8]	✓[8]	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	$V_a \rightsquigarrow V_u \rightsquigarrow V_a^{inv}$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Evict + Probe Invalidation	$V_d \rightsquigarrow V_u \rightsquigarrow A_d^{inv}$ (fast)	×	×	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓[10]	✓[4,10]	✓[10]	✓	×	✓	✓[8]	✓[8]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	$V_a \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (fast)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Prime + Time Invalidation	$A_d \rightsquigarrow V_u \rightsquigarrow V_d^{inv}$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10]	✓[10]	✓[10]	✓	×	✓	✓[8]	✓[8]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	$A_a \rightsquigarrow V_u \rightsquigarrow V_a^{inv}$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Flush + Time Invalidation	$V_u \rightsquigarrow A_a^{inv} \rightsquigarrow V_u^{inv}$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[4,10]	✓[4,10]	✓[4,10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	×	×	×	×	✓	✓	✓	✓
	$V_u \rightsquigarrow V_a^{inv} \rightsquigarrow V_u^{inv}$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	×	×	×	×	✓	✓	✓	✓

1 [1] Dynamic adjustment of ways for different threads is assumed to be properly used according to the running program's cache usage. [2] Some software assumptions listed in the entries in this column have been implemented by the cache's related software. [3] Flush is disabled, but cache coherence might be used to do the data removal. [4] For L1 cache and TLB, flushing is done during context switch. [5] The techniques are implemented in L1 cache, TLB and last-level cache which consist of the whole cache hierarchy, where L1 cache and TLB require software flush protection and the last-level cache can be achieved by simple hardware partitioning. To protect all levels of caches, the software assumptions need to be added. [6] The technique is now only implemented in last-level cache. [7] The technique now only targets shared cache. [8] The technique only targets on inclusion last-level cache. [9] The technique targets on data cache hierarchy. [10] For last-level cache, cache is partitioned between the victim and the attacker.

3. A secure cache disallows certain step from the three-step model to be executed, thus prevents the corresponding vulnerability. For instance, when PL cache preloads and locks the security critical data in the cache, vulnerabilities such as $A_d \rightsquigarrow V_u \rightsquigarrow V_d^{inv}$ (slow) (one type of Prime + Time Invalidation) will not be possible since a preloaded locked security critical data will not allow A_d in *Step* 1 to replace it. In this case, A_d cannot be in the cache, so this vulnerability cannot be triggered in PL cache.

From the security prospective, the entries of the secure cache in Table 4 and Table 5 should have as many green colored cells as possible. If a cache design has any red cells, then it cannot defend against that type of vulnerability – attacker using the timing-based side-channel vulnerability that corresponds to the red cell can attack the system.

The third column in Table 4 and Table 5 shows a normal set associative cache, which cannot defend against any type of timing-based side-channel vulnerabilities. Meanwhile, the last column of Table 4 and Table 5 shows the situation where cache is fully disabled. As is expected, the timing-based side-channel vulnerabilities are eliminated and timing-based attacks will not succeed. Disabling caches, however, has tremendous performance penalty. Similarly, second-to-last column shows Nondeterministic Cache, which totally randomizes cache access time. It can defend all the attacks, but again will have at tremendous cost to security when it the application is complex.

In each of the entry that shows the effectiveness of a secure cache against a vulnerability, we listed two results. Left one is for normal execution, and the right one is for speculative execution. Some secure caches such as InvisiSpec cache targets timing-based side channels in speculative execution. For most of the caches that do not differentiate speculative executions and normal executions, two sub-columns for each cache are the same.

6 Secure Cache Techniques

Among the secure cache designs presented in the prior section, there are three main techniques that the caches utilize to prevent timing-based side-channel vulnerabilities: differentiating sensitive data, partitioning and randomization.

Differentiating sensitive data (columns for CATalyst cache to columns for Random Fill cache in Table 4 and Table 5) allows the victim or attacker software or management software to explicitly label a certain range of the data of victim which they think are sensitive. The victim process or management software is able to use cache-specific instructions to protect the data and limit internal interference between victim's own data. E.g., it is possible to disable victim's own flushing of victim's labeled data, and therefore prevent vulnerabilities that leverage flushing. This technique allows the designer to have stronger control over security critical data, rather than forcing the system to assume all of victim's data is sensitive. However, how to identify sensitive data and whether this identification process is reliable are open research questions for caches that support differentiation of sensitive data.

This technique is independent of whether a cache uses partitioning or randomization techniques to eliminate side channels between the attacker and the victim. Caches that are able to label and identify sensitive data have advantage in preventing internal interference since they are able to differentiate sensitive data from the normal data and can make use of special instructions to give more privileges to sensitive data. However, it requires careful use when identifying the actual sensitive data and implementing corresponding security features on the cache.

Comparing PL cache with SP* cache, although both of them use partitioning, flush is able to be implemented to be disabled for victim's sensitive data in PL cache, where $V_u \rightsquigarrow V_a^{inv} \rightsquigarrow V_u$ (slow) (one type of Flush + Time) is prevented. Newcache is able to prevent $V_u \rightsquigarrow V_a \rightsquigarrow V_u$ (slow) (one type of Bernstein's Attack [5]) while most of the caches

without ability to differentiate sensitive data cannot because Newcache disallows replacing data as long as either data to be evicted or data to be cached is identified to be sensitive. However, permitting differentiation of sensitive data can potentially backfire on the cache itself. For example, Random Fill cache cannot prevent $V_u \rightsquigarrow A_d \rightsquigarrow V_u$ (slow) (one type of Evict + Time [27]) which most of the other caches can prevent or avoid, because the random fill technique loses its intended random behavior when the security critical data is initially loaded into the cache in *Step* 1.

Partitioning-based caches usually limit the victim and the attacker to be able to only access a limited set of cache block (columns for SP* cache to column for PL cache in Table 4 and Table 5). E.g. either there is static or dynamic partitioning of caches which allocates some blocks to *High* victim and *Low* attacker. The partitioning can be based not just on whether the memory access is victim's or attacker's, but also on where the access is to (e.g. *High* partition is determined by the data address) For speculative execution, attacker's code can be the part of speculation or out-of-order load or store, which is able to be partitioned (e.g., using speculative load buffer) from other normal operations. The partitioning granularity can be cache sets, cache lines or cache ways. Partitioning-based secure caches are usually able to prevent external interference by partitioning but are weak at preventing internal interference. When partitioning is used, interference between the attacker and the victim, or data belonging to different security levels, should not be possible and attacks based on external interference between the victim and the attacker will fail. However, the internal interference of victim's own data is hard to be prevented by the partitioning based caches. What's more, partitioning is recognized to be wasteful in terms of cache space and inherently degrades system performance [10]. Dynamic partitioning can help limit the negative performance and space impacts, but it could be at a cost of revealing some side-channel information when adjusting the partitioning size for each part. It also does not help with internal interference prevention.

In terms of the three-step model, the partitioning-based caches excel at making use of partitioning techniques to disallow the attacker to set initial states (*Step* 0) of victim partition by use of flushing or eviction, and therefore bring uncertainty to the final timing observation made by the attacker.

SP* cache can prevent external miss-based interference, but it still allows the victim and the attacker to get cache hits due to each other's data, which makes hit-based vulnerabilities happen, e.g., $V_d \rightsquigarrow V_u \rightsquigarrow V_a$ (fast) (one type of Cache Internal Collision [6]) vulnerability is one of the examples that SP* cache cannot prevent. SecVerilog cache is similar to SP* cache but prevents the attacker from directly getting cache hit due to victim's data for confidentiality and therefore prevents vulnerabilities such as $A_a^{inv} \rightsquigarrow V_u \rightsquigarrow A_a$ (fast) (one type of Flush + Reload [26]). SHARP cache mainly uses partitioning combined with random eviction to minimize the probability of evicting victim's data and prevent external miss-based vulnerabilities. It is vulnerable to hit-based or internal interference vulnerabilities such as $V_u \rightsquigarrow V_a \rightsquigarrow V_u$ (slow) (one type of Bernstein's Attack [5]) vulnerability. DAWG cache will only allow the data to get a cache hit if both its address and the process ID are the same. Therefore, compared with normal partitioning cache such as SP* cache, DAWG cache is able to prevent vulnerabilities such as $V_d \rightsquigarrow V_u \rightsquigarrow A_d^{inv}$ (fast) (one type of Prime + Flush).

SecDCP and NoMo cache both leverage dynamic partitioning to improve performance. Compared to SecVerilog cache, SecDCP cache introduces certain side channels which manifest themselves when the number of ways assigned to the victim and attacker changes, e.g., $V_u \rightsquigarrow A_a^{inv} \rightsquigarrow V_u$ (slow) (one type of Flush + Time) vulnerability. NoMo cache behaves more carefully when changing the number of ways during dynamic partitioning, however, it requires victim's sensitive data to fit into the assigned partitions, otherwise it will be put into the unreserved way and allow eviction by the attacker. SecDCP does not have unreserved way. All the spaces in the cache will either belong to *High* partition or

Low partition.

Sanctum cache and CATalyst cache are both controlled by a powerful software monitor and they disallow secure page sharing between victim and attacker to prevent vulnerabilities such as $A_d \rightsquigarrow V_u \rightsquigarrow A_a$ (fast) (one type of Flush + Reload [26]). Sanctum cache does not consider internal interference while CATalyst cache is more carefully designed to prevent different vulnerabilities with the implemented software system, so far supporting preventing all of the vulnerabilities, but only works for LLC and with high software implementation complexity and some assumptions that might be hard to achieve in other scenarios, e.g., assuming the secure partition is big enough to fit all the secure data. MI6 cache is the combination of Sanctum and disabling speculation when interacting with the outside world. Therefore, in normal execution, it behaves the same as Sanctum. For speculative execution, because it will simply disable all the speculation when involving the outside world, the external interference vulnerability such as $V_d \rightsquigarrow V_u \rightsquigarrow A_d$ (slow) (one type of Evict + Probe) vulnerability will be prevented.

InvisiSpec cache does not modify the original cache state but places the data in a speculative buffer partition during the speculation or out-of-order load or store. Since during speculation cache state is not actually updated, the speculative execution cannot trigger any of the steps in the three-step model. RIC cache and CEASER cache focus on eviction based attack and therefore are good at preventing even some internal miss-based vulnerability such as $V_u \rightsquigarrow V_a \rightsquigarrow V_u$ (slow) (one type of Bernstein's Attack [5]) but are bad at all hit-based vulnerabilities. PL cache is line-partitioned and uses locking techniques for victim's security critical data. It can prevent many vulnerabilities because preloading and locking secure data disallow the attacker or non-secure victim data to set initial states (*Step* 0) for victim partition, and therefore brings uncertainty to the final observation by the attacker, e.g., $A_d \rightsquigarrow V_u \rightsquigarrow V_a$ (fast) (one type of Cache Internal Collision [6]) vulnerability is prevented.

Randomization-based caches (columns for SHARP cache, and columns for RP cache to columns for Non Deterministic cache in Table 4 and Table 5) inherently de-correlate the relationship between information of victim's security critical data's address and observed timing from cache hit or miss, or between the address and observed timing of flush or cache coherence operations. For speculative execution, they also de-correlate the relationship between the address of the data being accessed during speculative execution or out-of-order load or store and the observed timing from a cache hit or miss. Randomization can be used when bringing data into the cache, evicting data, or both. Some designs randomize the address to cache set mapping. As a result of the randomization, the mutual information from the observed timing, due to having or not having data in the cache, could be reduced to 0, if randomization is done on every memory access. Some secure caches use randomization to avoid many of the miss-based internal interference vulnerabilities. However, they may still suffer from hit-based vulnerabilities, especially when the vulnerabilities are related to internal interference. However, randomization is also likewise recognized to increase performance overheads [19]. It also requires a fast and secure random number generator. Most of the randomization is cache-line-based and can be combined with differentiation of sensitive data to be more efficient.

RP cache allows eviction between different sensitive data, which leaves vulnerabilities such as $V_u \rightsquigarrow V_a \rightsquigarrow V_u$ (slow) (one type of Bernstein's Attack [5]) still possible, while Newcache prevents this. Both of the RP cache and Newcache are not able to prevent hit-based internal-interference vulnerabilities such as $A_a^{inv} \rightsquigarrow V_u \rightsquigarrow V_a$ (fast) (one type of Cache Internal Collision [6]). Random Fill cache is able to use total de-correlation of memory access and cache access of victim's security critical data to prevent most of the internal and external interference. However, when security critical data is initially directly loaded into the cache block for *Step* 1, Random Fill cache will not randomly load security critical data and allows vulnerabilities such as $V_u \rightsquigarrow V_a^{inv} \rightsquigarrow V_u$ (slow) (one type of

Table 6: Existing secure caches' implementation method, performance, power and area summary.

Metric		Set Associative Cache	SP* Cache [15, 34]	SecVerilog Cache [9, 8]	SecDCP Cache [14]	NoMo Cache [17]	SHARP Cache [16]	Sanctum Cache [13]	MI6 Cache[23]	InvisiSpec Cache [22]	CATalyst Cache [18]	DAWG Cache [21]	RIC [20]	PL Cache [10]	RP Cache [10]	Newcache [11, 33]	Random Fill Cache [12]	CEASER Cache [24]	Non Deterministic Cache [19]	Cache Disabled
Cache Configuration	L1 Cache	–	–	4-way 32kB	private 2-way 32kB D/I	8-way 32KB D/I	private 4-way 32KB D/I	8-way 32KB D/I	8-way 32KB D/I	private 8-way 64KB D, 4-way 32KB I	–	private 2× 8-way 32 KB	4-way 32 KB D/I	direct-mapped, 2-way and 4-way 4KB to 32KB	2-way 4-way 16KB 32KB	2-way, 4-way or 8-way	4-way 32 KB	private 8-way 32KB	2-way 2 KB D/I	–
	L2 Cache	–	–	–	shared 8/16-way 1/2MB	unified 8-way 256KB	private 8-way 256KB	256KB 8-way L2	1MB, 16-way, max 16 requests	–	–	private 8-way 256 KB	8-way 256 KB	–	–	–	8-way 2 MB	private 8-way 256KB	shared 4-way 128 KB	–
	LLC	–	–	–	16-way 2MB	shared 16-way 2MB	shared 16-way 2MB	8MB 16-way LLC partitioned into core-local slices	coherent with I and D	shared 16-way 2MB	20-way 20 MB	shared 8× 16-way 2 MB	shared 16-way 2 MB/512 KB	–	–	–	–	shared 16-way 8MB	–	–
Benchmark		–	RSA, AES and MD5	MiBench ciphers and hash functions of OpenSSL	SPEC 2006	SPEC 2006	SPEC INT2006, SPEC FP2006 and PARSEC	SPEC INT2006	SPEC INT2006	SPEC INT2006, SPEC FP2006 and PARSEC	SPEC 2006 and PARSEC	PARSEC and GAP Benchmark Suite (GAPBS)	SPEC 2006	AES, SPEC 2000	AES, SPEC 2000	SPEC 2000	SPEC 2006	SPEC 2006 and GAP	AES cryptographic algorithm	–
Implementation		–	–	MIPS processor	Gem5 simulator [39]	Pin [40] based trace-driven x86 simulator	MARSS [41] cycle-level full-system simulator	Rocket Chip Generator [42]	RiscyOO processor [43] + Xilinx FPGA	Gem5 simulator + CACTI 5 [44]	Intel Xeon E5 2618L v3 processors	zsim [45] execution driven x86-64 simulator and Haswell [46] hardware	Cacti [47] version 6.5	M-Sim v2.0 [48]	M-Sim v2.0	CACTI 5.0	Gem5 simulator	Pin-based x86 simulator	HotLeakage simulator [49]	–
Performance Overhead		–	1%	–	12.5% better over static cache partitioning	1.2% average, 5% worst	3%-4%	–	–	reduce the execution slow-down of Spectre from 74% to only 21%	average slowdown of 0.7% for SPEC and 0.5% for PARSEC	L1 and L2 most 4%-7%	improves 10%	12%	0.3%, 1.2% worst	within the 10% range of the real miss rate	3.5%, 9% if setting the window size to be largest	1% for performance optimization	7% with simple benchmarks	–
Power		–	–	–	–	–	–	–	–	L1 0.56 mW, LLC 0.61 mW	–	–	–	–	average 1.5 nj	< 5% power	–	–	–	–
Area Overhead		–	–	–	–	–	–	–	–	L1-SB LLC-SB Area (mm2) 0.0174 0.0176	–	–	0.176%	–	–	–	–	–	–	–

Flush + Time) vulnerability to exist. CEASER cache uses encryption scheme plus dynamic remapping to randomize mapping from memory addresses to cache sets. However, this targets eviction based attacks and cannot preventing hit-based vulnerabilities such as $V_a \rightsquigarrow V_u^{inv} \rightsquigarrow V_a^{inv}$ (fast) (one type of Flush + Probe Invalidation). Non Deterministic cache totally randomizes timing of cache accesses by adding delays and can prevent all attacks (but at tremendous performance cost).

6.1 Estimated Performance and Security Tradeoffs.

Table 6 shows the implementation and performance results of the secure caches, as listed by the designer in the different papers. At the extreme end, there is the Non Deterministic cache: with complete randomization, the security is guaranteed for all cache timing-based side-channel vulnerabilities; but the performance can be seriously degraded when it is used for more complex application than AES algorithm. Disabling caches eliminates the attacks, but at a huge performance cost. Normally, a secure cache needs to sacrifice some performance in order to de-correlate memory access with the timing. The secure caches that tend to be able to prevent more vulnerabilities usually have weaker performance compared with other secure caches. E.g., more security seems to imply less performance.

6.2 Secure Cache Features for Defending Timing-Based Attacks.

Based on the above analysis, a good secure cache should consider all the 72 types of *Strong* vulnerabilities, e.g., external-interference and internal-interference, hit-based and miss-based vulnerabilities. Considering all factors and based on Table 4 and Table 5, we have several suggestions and observations for a secure cache design which can defend timing-based attacks:

- Internal interference is important for caches to prevent timing-based attacks and is the weak point of most of the secure caches. To prevent this, the following three subpoints should be considered:
  - Miss-based internal interference can be solved by randomly evicting data to de-correlate memory access with timing information when either data to be evicted or data to be cached is sensitive, e.g., Newcache prevents $V_u \rightsquigarrow V_a \rightsquigarrow V_u$ (slow) (one type of Bernstein's Attack [5]) vulnerability.
  - Hit-based internal interference can be solved by randomly bringing data into the cache, e.g., Random Fill cache prevents $A_d \rightsquigarrow V_u \rightsquigarrow V_a$ (fast) (Cache Internal Collision) vulnerability.
  - To limit internal interference at lower performance cost, rather than simply assume all of victim's data is sensitive, it is better to differentiate real sensitive data from other data in the victim code. However, identification of sensitive information needs to be carefully used, e.g., Random Fill cache is vulnerable to $V_u \rightsquigarrow A_d \rightsquigarrow V_u$ (fast) (one type of Evict + Time [27]) vulnerability which most of the secure caches are able to prevent.
- Direct partitioning between the victim and the attacker, although may hurt cache space utilization or performance, is good at disallowing attacker to set known initial state to victim's partition and therefore prevents external interference. Alternatively, careful use of randomization can also prevent external interference.

It should be noted that some cache designs only focus on certain levels, e.g., CATalyst cache only works at the last level cache. In order to fully protect the whole cache system from timing-based side-channel attacks, all levels of caches in the hierarchy should be protected with related security features. E.g., Sanctum is able to prevent all levels of caches from L1 to last-level cache. Consequently, secure cache design needs to be realizable at all levels of the cache hierarchy.

7 Related Work

There is a lot of existing attacks exploring timing-based cache side channels, e.g., [3, 4, 5, 6, 7, 27, 32, 26, 29, 30, 28]. Furthermore, our recent paper [25] has summarized cache timing-based side-channel vulnerabilities using a three-step model, and inspired this work on checking which side channel vulnerability types are truly defeated by the secure caches in context of timing-based attacks. [50] used finite-state machine to model cache architectures and leveraged mutual information to measure potential side-channel leakage of the modeled cache architectures. Meanwhile, [34] modeled interference using probabilistic information flow graph, and used attacker's success probability to estimate different caches' ability to defend against some cache timing-based side-channel attacks. However, they did not explore all the possible vulnerabilities due to cache timing-based side channels.

There is also some other work focusing on cache side channel verification [51, 52, 53]. Among these, CacheAudit [51] efficiently computes possible side-channel observations using abstractions in a modular way. Bit-level and arithmetic reasoning is used in [52] for memory accesses in the presence of dynamic memory allocation. CacheD [53] detects potential cache differences at each program point and leverages symbolic execution and constraint solving.

Hardware transactional memory has also been leveraged to prevent timing-based cache side-channel attacks [54, 55]. Hardware transactional memory (HTM) is available on modern commercial processors, such as Intel's Transactional Synchronization Extensions (TSX). Its main feature is to abort the transaction and roll back the modifications whenever a cache block contained in the read set or write set is evicted out of the cache. In [54], HTM was combined with preloading strategy for code and data to prevent Flush + Reload attacks in the local setting, and Prime and Probe attacks in the cloud setting. In [55], the software-level solution targets system calls, page faults, code refactoring, and abort reasoning to eliminate not only Prime + Probe, Flush + Reload, but also Evict + time and Cache Collision attacks.

8 Conclusion

This paper first proposed a new single-cache-block three-step model in order to model all possible cache timing side-channel vulnerabilities. It further provided the cache three-step simulator and reduction rules to derive effective vulnerabilities including existing attacks and ones that have not been exploited in literature. With exhaustive effective vulnerability types listed, this paper presented analysis of 17 secure processor cache designs with respect to how well they can defend against these timing-based side-channel vulnerabilities. Our work showed that vulnerabilities based on internal interference of the victim application are difficult to protect against and many secure cache designs fail in this. We also provided a summary of secure processor cache features that could be integrated to make an ideal secure cache that is able to defend timing-based attacks. Overall, implementing a secure cache in a processor can be a viable alternative to defend timing-based side-channel attacks. However, it requires design of an ideal secure cache, or correction of existing secure cache designs to eliminate the few attacks that they don't protect against.

Acknowledgment

This work was supported by NSF grant number 1813797 and through SRC award number 2844.001.

References

[1] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre Attacks: Exploiting Speculative Execution,” *ArXiv e-prints*, Jan. 2018.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, “Meltdown,” *ArXiv e-prints*, Jan. 2018.

[3] D. Gullasch, E. Bangerter, and S. Krenn, “Cache games–Bringing access-based cache attacks on AES to practice,” in *Security and Privacy (SP), 2011 IEEE Symposium on*, pp. 490–505, IEEE, 2011.

[4] C. Percival, “Cache missing for fun and profit,” 2005.

[5] D. J. Bernstein, “Cache-timing attacks on AES,” 2005.

[6] J. Bonneau and I. Mironov, “Cache-collision timing attacks against AES,” in *International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 201–215, Springer, 2006.

[7] O. Aciiçmez and Ç. K. Koç, “Trace-driven cache attacks on AES (short paper),” in *International Conference on Information and Communications Security*, pp. 112–121, Springer, 2006.

[8] D. Zhang, A. Askarov, and A. C. Myers, “Language-based control and mitigation of timing channels,” *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 99–110, 2012.

[9] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, “A hardware design language for timing-sensitive information-flow security,” in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 503–516, ACM, 2015.

[10] Z. Wang and R. B. Lee, “New cache designs for thwarting software cache-based side channel attacks,” in *ACM SIGARCH Computer Architecture News*, vol. 35, pp. 494–505, ACM, 2007.

[11] Z. Wang and R. B. Lee, “A novel cache architecture with enhanced performance and security,” in *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pp. 83–93, IEEE, 2008.

[12] F. Liu and R. B. Lee, “Random fill cache architecture,” in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pp. 203–215, IEEE, 2014.

[13] V. Costan, I. A. Lebedev, and S. Devadas, “Sanctum: Minimal Hardware Extensions for Strong Software Isolation.,” in *USENIX Security Symposium*, pp. 857–874, 2016.

[14] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, “SecDCP: secure dynamic cache partitioning for efficient timing channel protection,” in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*, pp. 1–6, IEEE, 2016.

[15] R. B. Lee, P. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang, “Architecture for protecting critical secrets in microprocessors,” in *ACM SIGARCH Computer Architecture News*, vol. 33, pp. 2–13, IEEE Computer Society, 2005.

[16] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, “Secure Hierarchy-Aware Cache Replacement Policy (SHARP): Defending Against Cache-Based Side Channel Attacks,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 347–360, ACM, 2017.

[17] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, p. 35, 2012.

[18] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pp. 406–418, IEEE, 2016.

[19] G. Keramidas, A. Antonopoulos, D. N. Serpanos, and S. Kaxiras, "Non deterministic caches: A simple and effective defense against side channel attacks," *Design Automation for Embedded Systems*, vol. 12, no. 3, pp. 221–230, 2008.

[20] M. Kayaalp, K. N. Khasawneh, H. A. Esfeden, J. Elwell, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, "RIC: relaxed inclusion caches for mitigating LLC side-channel attacks," in *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*, pp. 1–6, IEEE, 2017.

[21] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors,"

[22] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 428–441, IEEE, 2018.

[23] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, S. Devadas, *et al.*, "MI6: Secure Enclaves in a Speculative Out-of-Order Processor," *arXiv preprint arXiv:1812.09822*, 2018.

[24] M. K. Qureshi, "CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 775–787, IEEE, 2018.

[25] S. Deng, W. Xiong, and J. Szefer, "Cache timing side-channel vulnerability checking with computation tree logic," in *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, no. 2, ACM, 2018.

[26] Y. Yarom and K. Falkner, "FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack.," in *USENIX Security Symposium*, pp. 719–732, 2014.

[27] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *Cryptographers' Track at the RSA Conference*, pp. 1–20, Springer, 2006.

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

[29] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are Coherence Protocol States Vulnerable to Information Leakage?," in *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, pp. 168–179, IEEE, 2018.

[30] D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches.," in *USENIX Security Symposium*, pp. 897–912, 2015.

[31] C. Trippel, D. Lustig, and M. Martonosi, "MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols," *arXiv preprint arXiv:1802.03802*, 2018.

[32] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+ Flush: a fast and stealthy cache attack," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 279–299, Springer, 2016.

[33] F. Liu, H. Wu, K. Mai, and R. B. Lee, "Newcache: Secure cache architecture thwarting cache side-channel attacks," *IEEE Micro*, vol. 36, no. 5, pp. 8–16, 2016.

[34] Z. He and R. B. Lee, "How secure is your cache against side-channel attacks?," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 341–353, ACM, 2017.

[35] R. E. Kessler and M. D. Hill, "Page placement algorithms for large real-indexed caches," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 4, pp. 338–359, 1992.

[36] G. Taylor, P. Davies, and M. Farmwald, "The TLB slice-a low-cost high-speed address translation mechanism," in *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, pp. 355–363, IEEE, 1990.

[37] I. R.-T. P. by Utilizing, "Cache Allocation Technology," *Intel Corporation, Apr*, 2015.

[38] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, *et al.*, "Prince–a low-latency block cipher for pervasive computing applications," in *International Conference on the Theory and Application of Cryptology and Information Security*, pp. 208–225, Springer, 2012.

[39] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[40] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, pp. 190–200, ACM, 2005.

[41] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSS: a full system simulator for multicore x86 CPUs," in *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pp. 1050–1055, IEEE, 2011.

[42] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanović, and K. Asanović, "A 45nm 1.3 GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators," in *European Solid State Circuits Conference (ESSCIRC), ESSCIRC 2014-40th*, pp. 199–202, IEEE, 2014.

[43] S. Zhang, A. Wright, T. Bourgeat, and A. Arvind, "Composable Building Blocks to Open up Processor Design," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 68–81, IEEE, 2018.

[44] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "CACTI 5.1," tech. rep., Technical Report HPL-2008-20, HP Labs, 2008.

[45] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *ACM SIGARCH Computer architecture news*, vol. 41, pp. 475–486, ACM, 2013.

[46] I. X. Processor, "E5-2680 v3."

[47] P. Shivakumar and N. P. Jouppi, "Cacti 3.0: An integrated cache timing, power, and area model," 2001.

[48] J. Sharkey, D. Ponomarev, and K. Ghose, "M-sim: a flexible, multithreaded architectural simulation environment," *Techenical report, Department of Computer Science, State University of New York at Binghamton*, 2005.

[49] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan, "Hotleakage: A temperature-aware model of subthreshold and gate leakage for architects," *University of Virginia Dept of Computer Science Tech Report CS-2003*, vol. 5, 2003.

[50] T. Zhang and R. B. Lee, "New models of cache architectures characterizing information leakage from cache side channels," in *Proceedings of the 30th Annual Computer Security Applications Conference*, pp. 96–105, ACM, 2014.

[51] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke, "Cacheaudit: A tool for the static analysis of cache side channels," *ACM Transactions on Information and System Security (TISSEC)*, vol. 18, no. 1, p. 4, 2015.

[52] G. Doychev and B. Köpf, "Rigorous analysis of software countermeasures against cache attacks," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 406–421, ACM, 2017.

[53] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "CacheD: Identifying Cache-Based Timing Channels in Production Software," in *26th USENIX Security Symposium. USENIX Association*, 2017.

[54] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *USENIX Security Symposium*, pp. 217–233, 2017.

[55] S. Chen, F. Liu, Z. Mi, Y. Zhang, R. B. Lee, H. Chen, and X. Wang, "Leveraging Hardware Transactional Memory for Cache Side-Channel Defenses," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pp. 601–608, ACM, 2018.