Journal of Hardware and Systems Security (2019) 3:397–425
https://doi.org/10.1007/s41635-019-00075-9

Analysis of Secure Caches Using a Three-Step Model for Timing-Based Attacks

Shuwen Deng[1] · Wenjie Xiong[1] · Jakub Szefer[1]

Received: 19 February 2019 / Accepted: 23 July 2019 / Published online: 20 November 2019

Abstract
Many secure cache designs have been proposed in literature with the aim of mitigating different types of cache timing–based 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 presents a novel three-step modeling approach that is used to exhaustively enumerate all the possible cache timing–based 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 vulnerabilities derived from the three-step model, this work further manually 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 using the new three-step model, this paper further summarizes different techniques gleaned from the secure cache designs and their ability help mitigate different types of cache timing–based vulnerabilities.

Keywords Secure caches · Timing-based attacks · Security analysis · Side channels · Covert channels

1 Introduction

Research on timing-based attacks in computer processor caches has a long history, e.g., [1, 3, 5, 19, 36], predating their recent use in Spectre [26] attacks. These past attacks have shown the possibility to extract sensitive information via the timing-based channels, and often the focus is on extracting cryptographic keys. In addition, due to the recent Spectre [26] attacks, there is now renewed interested in timing channels. Especially, the Spectre 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.

We have recently proposed a three-step model [11] in order to analyze cache timing–based side-channel attacks. The previous 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 [11] 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 (due to a cache coherence operation or a flush instruction, for example). 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

✉ Shuwen Deng
shuwen.deng@yale.edu

Wenjie Xiong
wenjie.xiong@yale.edu

Jakub Szefer
jakub.szefer@yale.edu

1 Yale University, New Haven, CT 06520, USA

operations and modeling of the cache attacks. Speculative execution has gotten increased attention due to recent Spectre [26] attacks, many of which depend on timing channels to actually extract information—speculation alone is not enough for most of these attacks. Our model considers timing channels in general, independent of whether it is a side or a covert channel.

In the process of development of the improved three-step model, we have uncovered 43 types of timing-based vulnerabilities which have not been previously exploited (in addition, there are 29 types that map to attacks already known in literature). We cannot directly compare the types of vulnerabilities found in this work and in our prior work [11] due to the improved and different categorizations of the states of the cache block.

To address the threat of the prior cache timing–based attacks, to date, 18 different secure cache designs have been presented in academic literature [7, 10, 12, 22, 23, 25, 27, 29, 30, 38, 48–53, 56, 57]. The 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 our three-step modeling approach to reason about all the possible timing-based vulnerabilities. Especially, since our work demonstrates a number of new timing-based attacks, the existing secure caches have never been analyzed with respect to these new attacks before. For this work, we manually reviewed and analyzed the 18 existing secure cache designs [7, 10, 12, 22, 23, 25, 27, 29, 30, 38, 48–53, 56, 57] in terms of the security features and implementations. Most of these designs do not have publicly available hardware implementation source code, so automatic analysis of the caches is not possible.

Based on the analysis, we summarize cache features that help improve security. Especially, we propose that an "ideal" secure caches and 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 placed in the cache. More details of the possible defenses are discussed in Section 5 and Section 6.

1.1 Contributions

The new contributions of this work over [11] 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 vulnerabilities (Section 3).
  - Inclusion of cache coherence issues into the three-step mode.
  - Expansion of the three-step model to consider both cases of normal and speculative execution attacks.
  - 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.
- Overview of the 18 secure cache designs that have been presented in academic literature (Section 4).
- Manual evaluation of 18 secure processor cache designs to determine how they can help prevent timing-based attacks and analysis of security features secure caches used (Sections 5 and 6).
- Discussion of "ideal" secure caches and the features they would need (Section 6).
- Attack strategies description and comparison among different attack strategies (Appendix A).
- Analysis of the soundness of the three-step model and why three-steps are able to describe all timing-based vulnerabilities (Appendix B).

2 Cache Timing–Based Attacks and the Threat Model

Modern processor caches are known to be vulnerable to timing-based 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. The cache coherence protocol can also change the cache states and affect the timing of the memory operations. The cache coherence may invalidate a cache block 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, for example. From these timing differences of memory-related operations, the attacker can infer a data's specific memory address or corresponding cache index value, and thus learn some information about the victim's secrets.

2.1 Threat Model

This work focuses only on timing-based attacks in processor caches. Numerous other types of side and covert channels that do not use timing or caches exist, e.g., power-based [9], EM-based [2] (including RF), thermal-based [33], and in processor channels based on features such as power state of the AVX unit [40], for example. This work aims to explore main cache attacks only, but similar approach can be done for the other buffers or cache-like structures, which may be target of attack once main processor caches are secured.

In our threat model, an attacker's objective is to retrieve victim's secret information using timing-based channels in the processor cache. Specifically, we consider the situation where the victim accesses an address u and the address depends on some secret information. The address u is within some set of physical memory locations x, which are known to the attacker. The goal of the attacker is to obtain the address u or at least partial bits of it which relate to the cache index of the address.

We assume the attacker knows some of the source code of the victim. Especially, the attacker can only learn some information [1] about the address u from the timing channels, but with knowledge of the source code, he or she can further infer the likely specific value of u, and thus infer the secret he or she is trying to learn.

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 how the timing of the memory-related operations depends on the cache states.

The attacker further is able to force the victim to execute a specific function. For example, the 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. The attacker is also assumed not to be able to undermine the physical implementation or change the hardware, e.g., he or she cannot influence randomness generated by any random number generators in hardware. Physical or invasive attacks are not in scope of this work. 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.

[1]For a hit-based vulnerabilities, the attacker is able to learn the full address of the victim's sensitive data, while for the miss-based vulnerabilities, the attacker usually can learn the cache index of the victim's sensitive data. For more details of these vulnerabilities' categorizations, please refer to Section 3.3.3.

2.2 Side and Covert Channels

This work focuses on both side and 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. For brevity, we just use the term "victim" in the text to represent both the victim (for side channels) and the sender (for covert 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 running once every time slice (when no hyperthreading is used). Our model can be applied to both of the scenarios since our model abstracts away how the sharing happens.

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

This section explains how we developed the three-step modeling approach and used it to model the behavior of the cache logic and to enumerate all the possible cache timing–based vulnerabilities.

3.1 Introduction of the Three-Step Model

We have observed that all of the existing cache timing–based 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 by the victim or the attacker on the same core or different cores. When the victim and the attacker are on different cores, cache coherence will also be triggered when one of the memory-related operations is performed.

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 block 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 [55] attack, in $Step$ 1, a cache block 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 block 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 block, 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, cache entry, or cache line). 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.

3.2 States of the Three-Step Model

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 for each step in our three-step model and their formal definitions. Figure 1 graphically shows for each possible state how the memory location maps to the cache block.

Table 1 The 17 possible states for a single cache block in our three-step model

State	Description
V_u	A memory location u 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 set x of memory locations, a set which is known to the attacker. It may have the same index as a or a^{alias}, and thus conflict with them in the cache block. The goal of the attacker is to learn the index of the address u. The attacker does not know the address u, hence there is no A_u in the model.
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 is known to the attacker.
$A_{a^{alias}}$ 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. The address a^{alias} is known to the attacker.
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. The address d is known to the attacker.
A^{inv} or V^{inv}	The cache block 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 data and its address are "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. The address a is known to the attacker.
$A_{a^{alias}}^{inv}$ or $V_{a^{alias}}^{inv}$	The cache block state can be anything except a^{alias} in this cache block now. The data and its address are "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. The address a^{alias} is known to the attacker.
A_d^{inv} or V_d^{inv}	The cache block state can be anything except d in this cache block now. The data and its address are "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. The address d is known to the attacker.
V_u^{inv}	The cache block state can be anything except u in the cache block. The data and its address are "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.

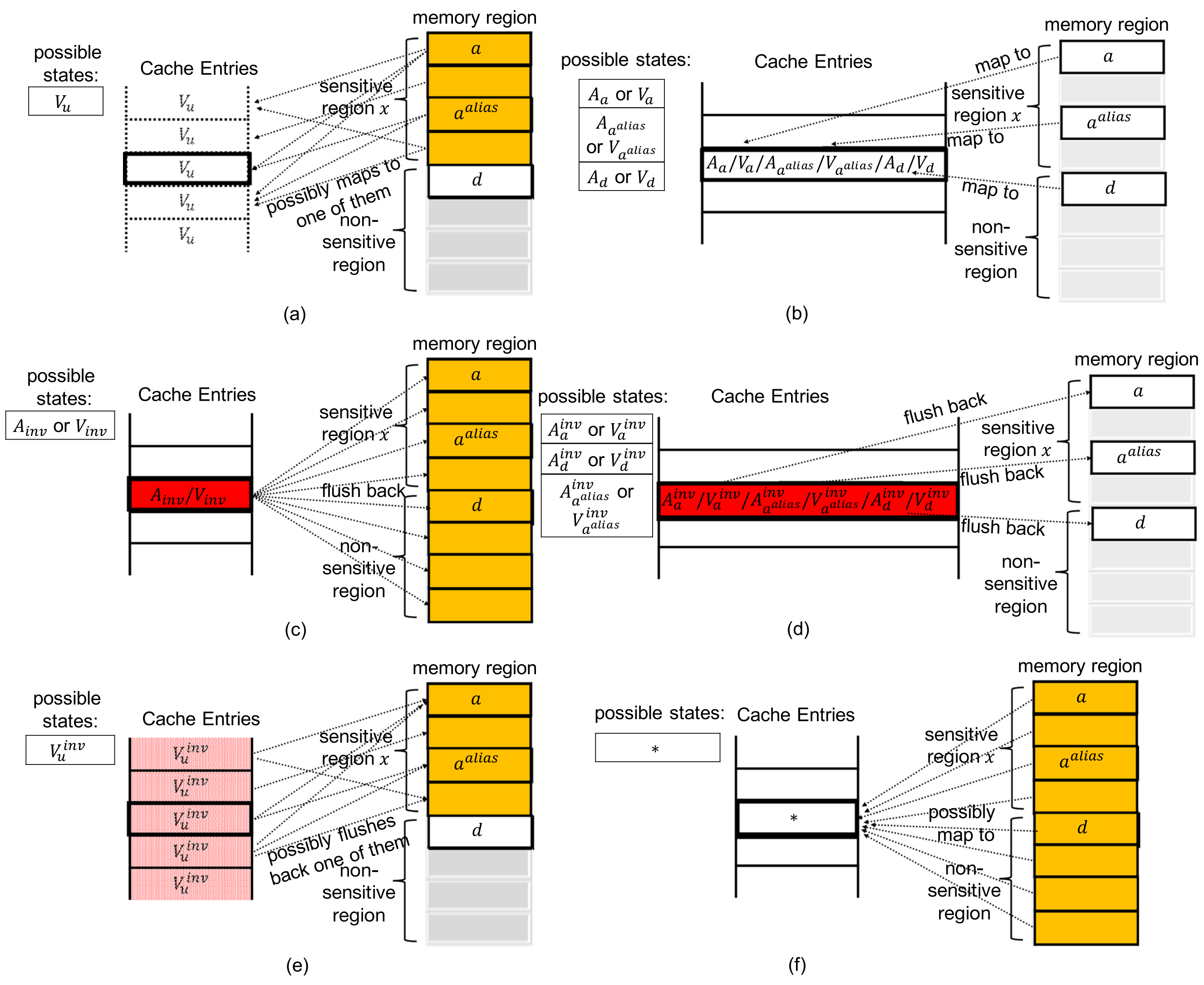

Fig. 1 The 17 possible states for a single cache block in our three-step model: **a** V_u; **b** $A_a/V_a/A_{a^{alias}}/V_{a^{alias}}/A_d/V_d$; **c** A^{inv}/V^{inv}; **d** $A_a^{inv}/V_a^{inv}/A_{a^{alias}}^{inv}/V_{a^{alias}}^{inv}/A_d^{inv}/V_d^{inv}$; **e** V_u^{inv}; **f** $\star$)

In each sub-figure of Fig. 1, the left-most part shows the possible state being described in the sub-figure. The middle part shows the possible situation of the cache state affected by each. For all sub-figures, the middle cache block (shown in bold) is the targeted cache block. Right-most part shows the memory region in relation to the cache block. Recall, the addresses a and a^{alias} are within the sensitive set of addresses x, while d is outside the set of sensitive addresses (for simplicity the set is shown as a contiguous region, but it can be any set). Also recall, A represents the operations performed by the attacker and V represents the victim's operations.

Figure 1a shows the description of the possible state V_u, where address u is within sensitive set and unknown to the attacker. Therefore, it can possibly map to any cache block including the target cache block shown in the middle. Since its position in the cache and specific address is unknown, we show V_u in dashed lines. Meanwhile, Fig. 1e shows the description of the possible state V_u^{inv}, which is result of the victim invalidating data at the sensitive address u and possibly invalidating some address within sensitive region. Further, Fig. 1f shows the description of the possible state $*$, which represents null knowledge of the address for the attacker to this corresponding cache block. Therefore, it can possibly refers to any address in the memory, or no valid address at all.

Figure 1b shows the description of the possible state $A_a/V_a/A_{a^{alias}}/V_{a^{alias}}/A_d/V_d$. Their addresses are all known to the attacker and map to the same targeted cache block. Both a and a^{alias} are within the sensitive set of addresses x and a^{alias}, as its name indicates, is a different address than a but still within set x and maps to the same cache block as a. Address d is outside of the set x. Meanwhile, Figure 1d shows the description of the possible state

$A_a^{inv}/V_a^{inv}/A_{a^{alias}}^{inv}/V_{a^{alias}}^{inv}/A_d^{inv}/V_d^{inv}$, which correspond to invalidation of the address shown in the subscript of the state. Some additional possible invalidation states, A^{inv}/V^{inv}, are shown in Figure 1c. These states indicate no valid address is in the cache block. Therefore, all the possible addresses that mapped to this cache block, e.g., a, a^{alias}, d, and u (if it mapped to this block), before the invalidation step A^{inv}/V^{inv} will be flushed back to the memory.

3.3 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. 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 Fig. 2, 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.

3.3.1 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 or a^{alias} and thus does not map to this cache block.

The cache three-step simulator is implemented in Python script and its pesudo implementation is shown in Algorithm 1. Simulator's inputs are 17 possible states for each of the step. Outputs are all the vulnerabilities that belong to the *Strong* or the *Weak* type or the *Ineffective* type. The simulator uses a nested `for` loop to check all possible combinations (4913) of the three step pattern. For each step of each pattern, if it is V_u, this step will be extended to be one of three candidates: V_a, $V_{a^{alias}}$ and V_{NIB}. If it is V_u^{inv}, this step will be extended to be one of three candidates: V_a^{inv}, $V_{a^{alias}}^{inv}$ and V_{NIB}^{inv}. We wrote a function *output_timing* that takes three known memory access steps as input and output whether fast or slow timing will be observed for the last step. In this case, for each of the u-related step's candidate, we can derive a timing observation. Using these timing observation, function $judge\_type$ decides 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 u.

The simulator categorizes all the three-step patterns into three categories, as listed below. Figure 3 shows two examples for the *Strong Vulnerability* (a, b), *Weak Vulnerability* (c, d), and *Ineffective Three-Step* (e, f), categories respectively.

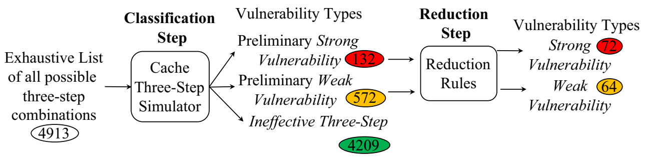

Fig. 2 Procedure to derive the effective types of three-step timing-based vulnerabilities. Ovals refer to the number of vulnerabilities in each category

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). 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 Fig. 3a, 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:* When fast or slow timing is observed by the attacker, he or she knows it corresponds to more than one possible value of u (e.g., a or a^{alias}). For these vulnerabilities, timing variation can still be observed due to different victim's behavior. However, the attacker cannot learn the value of the index of the address u unambiguously. For example, for type $\star \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ shown in Fig. 3c, when fast timing is observed, u possibly maps to a^{alias} or NIB (the reason for the possibility of u mapping to NIB to derive fast timing 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. For example, for type $A_a \rightsquigarrow V_u \rightsquigarrow A_d$ shown in Fig. 3f, no matter what the value of u is, attacker's observation is always slow timing.

Algorithm 1 Pseudo-code for the cache three-step simulator algorithm.

```
Input: $state[]$: a list containing 17 possible states for each of the step
Output: $strong[]$: a list containing all the vulnerabilities that belong to the Strong type
    $weak[]$: a list containing all the vulnerabilities that belong to the Weak type
    $ineffective[]$: a list containing all the ineffective typs
1: for $step1 \in len(state[])$ do
2:     for $step2 \in len(state[])$ do
3:         for $step3 \in len(state[])$ do
4:             $steps = [state[step1], state[step2], state[step3]]$
5:             $candidates = []$ // array to store all possible candidate combinations of this three-step pattern
6:             $res = []$ // array to store all possible timing observation regading different candidate combinations for this three-step pattern
7:             if ($u\_related(steps[0])$ or $u\_related(steps[1])$ or $u\_related(steps[2])$) then
8:                 for $possi\_candidate \in 3$ // $V_u$'s candidates are $V_a$, $V_{a^{alias}}$ and $V_{NIB}$; $V_u^{inv}$'s candidates are $V_a^{inv}$, $V_{a^{alias}}^{inv}$ and $V_{NIB}^{inv}$. Both candidate's number is 3. do
9:                     $candidates.append[[change\_u(steps[0], possi\_candidate),$
                       $change\_u(steps[1], possi\_candidate), change\_u(steps[2], possi\_candidate)]]$
10:                end for
11:                for $i \in 3$ do
12:                    $res.append(output\_timing(candidates[i]))$
13:                end for
14:                if $judge\_type(res) == Strong$ then
15:                    strong.append(steps)
16:                else
17:                    if $judge\_type(res) == Weak$ then
18:                        $weak.append(steps)$
19:                    else
20:                        $ineffective.append(steps)$
21:                    end if
22:                end if
23:            else
24:                $ineffective.append(steps)$
25:                continue
26:            end if
27:        end for
28:    end for
29: end for
```

After computing 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.

3.3.2 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. Figure 2 shows how the output of the simulator is filtered through the reduction rules to get the final list of vulnerabilities. Reduction's goal is to remove vulnerabilities of repeating or redundant types from the lists to form effective *Strong Vulnerability* or *Weak Vulnerability* output. 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

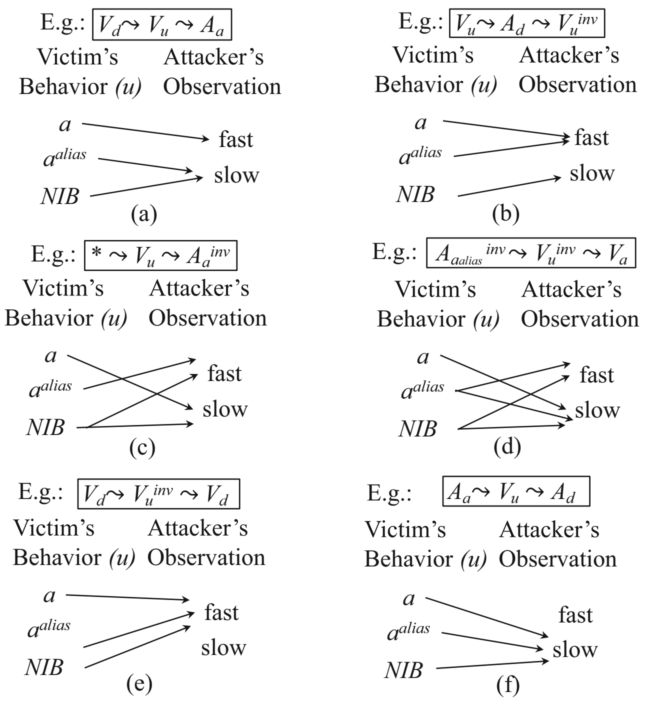
E.g.: $V_d \rightsquigarrow V_u \rightsquigarrow A_a$
Victim's Behavior (u)
Attacker's Observation
a
a^alias
NIB
fast
slow
(a)
E.g.: $V_u \rightsquigarrow A_d \rightsquigarrow V_u^{inv}$
(b)
E.g.: $* \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$
(c)
E.g.: $A_{a_{alias}}^{inv} \rightsquigarrow V_u^{inv} \rightsquigarrow V_a$
(d)
E.g.: $V_d \rightsquigarrow V_u^{inv} \rightsquigarrow V_d$
(e)
E.g.: $A_a \rightsquigarrow V_u \rightsquigarrow A_d$
(f)

Fig. 3 Examples of relations between victim's behavior (u) and attacker's observation for each vulnerability type: **a**, **b** *Strong Vulnerability*; **c**, **d** *Weak Vulnerability*; **e**, **f** *Ineffective Three-Step*

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. Three-step patterns with a 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. Three-step patterns with steps V_u and V_u^{inv} in adjacent consecutive steps with each other will only keep the latter step and eliminate the first step. For example, $A_a \rightsquigarrow V_u \rightsquigarrow V_u^{inv}$ can be reduced to $A_a \rightsquigarrow V_u^{inv}$ and further equivalent to $\star \rightsquigarrow A_a \rightsquigarrow V_u^{inv}$. So $A_a \rightsquigarrow V_u \rightsquigarrow V_u^{inv}$ can be eliminated.

3.3.3 Categorization of *Strong* Vulnerabilities

As is shown in Fig. 2, after applying the reduction rules, there are 72 types of *Strong* vulnerabilities remaining. In Appendix B, 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. And if there is a vulnerability, it can always be reduced to a model that requires only three steps. 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 Tables 2 and 3), these strategies correspond to well-known names for the attacks, if such exist, otherwise we provide a new name. In Appendix A, we provide description for each attack strategy to show the main idea behind them. We use existing names for attack strategies where such existed before, even if similar attacks, e.g., attacks differing in only one step, have been given different names before. We use these established names to avoid confusion, but detail some of the similarities in Appendix A as a clarification.

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 the Macro Type column of Tables 2 and 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[2]. 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. For example, the Cache Collision attack [5] is effectively based on the Internal Collision, and it maps to types labeled (2) in the Attack column in Tables 2 and 3. The types labeled **new** correspond to new attack not previously discussed in literature. We believe these 43 are new attacks not previously analyzed nor known.

4 Secure Caches

Having explained the three-step model, we now explore the various secure caches which have been presented in literature to date [7, 10, 12, 22, 23, 25, 27, 29, 30, 38, 48–53, 56, 57]. Later, 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.

[2]Invalidation is fast when the corresponding address which is to be invalidated does not exist in the cache since no operation is needed for the invalidation.

Table 2 The table shows all the cache timing-based cache vulnerabilities where the last step is a memory access related operation. The *Attack Strategy* column gives a 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 to derive sensitive address information, 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 [34]; (2) Cache Internal Collision attack [5]; (3) Bernstein's attack [3]; (4) Prime + Probe attack [34, 36], Alias-driven attack [18]; (5) Flush + Reload attack [54, 55], Evict + Reload attack [17]; (6) SpectrePrime, MeltdownPrime attack [46]

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	Vulnerability Type			Macro	Attack
	$Step$ 1	$Step$ 2	$Step$ 3		
Strategy				Type	
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 [16]

This section gives brief overview of the 18 secure cache designs that have been presented in academic literature in the last 15 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., [23, 48]. Some have been realized in FPGA, e.g., [7], and a few have been realized in real ASIC hardware, e.g., [31]. No specific secure caches have been implemented in commercial processors to the best of our knowledge; however, CATalyst [29] 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 levels of caches, 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 [20, 27][3] 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 [56, 57] 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 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 [48] 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 [12] 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 [53] uses both partitioning and randomization techniques to prevent victim's data from being evicted

[3]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 [27] with secure cache features of the Static-Partitioned cache from [20].

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 ith bit in the bitmap is set if the line is present in ith 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 [10] 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 implement 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 [24, 44] 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 [24, 44] 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 [7] is part of the memory hierarchy of the MI6 processor, which combines Sanctum [10] 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 [24, 44] 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 [52] 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 [29] uses partitioning, especially Cache Allocation Technology (CAT) [21] 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. That is, all data in the range x can fit in the secure partition.
Assumption 3. The victim 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 [25] (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 [22] (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 [49] provides isolation by partitioning the 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 [49] 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 hits need 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, 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 [31, 50] 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, 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 [30] 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 [30], 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 [38] 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) [6], 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.

SCATTER Cache [51] uses cache set randomization to prevent timing-based attacks. It builds upon two ideas. First, a mapping function is used to translate memory address and process information to cache set indices, the mapping is different for each program or security domain. Second, the mapping function also calculates a different index for each cache way, in a similar way to the skewed associative caches [41]. The mapping function can be keyed hash or keyed permutation derivation function—a different key is used for different application or security domain resulting in a different mapping from address to cache sets for each. Software (e.g., the operating system) is responsible for managing the security domains and process IDs which are used to differentiate the different software and assign it different keys for the mapping. For the hardware extension, a cryptographic primitive such as hashing and an index decoder for each scattered cache way is added. SCATTER cache also stores the index bits of the physical address to efficiently perform lookups and writebacks. There is also one bit per page-table entry added to allow the kernel to communicate with the user space for security domain identification.

Non Deterministic Cache [23] 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 that is less than the maximum value of the global counter. In this case, the cache delay is changed to be randomized. 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 manually evaluate the effectiveness of the 18 secure caches [7, 10, 12, 22, 23, 25, 27, 29, 30, 38, 48–53, 56, 57]. We analyze how well the different caches can protect against the 72 types of vulnerabilities defined in Tables 2 and 3, which cover all the possible *Strong* (according to the definition in Section 3) cache timing–based vulnerabilities. Following the analysis, discuss 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

Tables 4 and 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 [34]) 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 [3]) vulnerability.

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

1. A cache can prevent a timing attack if 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, in a regular set-associative cache, the $V_d \rightsquigarrow V_u \rightsquigarrow A_a$ (fast) (one type of Flush + Reload [55]) 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, in case of the RP cache [49] 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. A cache can prevent a timing attack if 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 [30], for example 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 would be cached instead.
3. A cache can prevent a timing attack if it disallows certain steps from the three-step model to be executed, thus prevents the corresponding vulnerability. For instance, when PL cache [49] 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 perspective, the entries of the secure cache in Tables 4 and 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 vulnerability that corresponds to the red cell can attack the system.

The third column in Tables 4 and 5 shows a normal set associative cache, which cannot defend against any type of timing-based vulnerabilities. Meanwhile, the last column of Tables 4 and 5 shows the situation where the cache is fully disabled. As is expected, the timing-based 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 a tremendous cost to security when the application is complex.

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

Table 4 Existing secure caches' protection against all possible timing-based vulnerabilities with last step to be memory access related operations.Single ✓ in a green cell means this cache can prevent the corresponding vulnerability. ○ in a pink cell means this cache can prevent the corresponding vulnerability in some degree. 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		Invi-siSpec Cache[9]		CAT-alyst Cache[2,5]		DAWG Cache		RIC[8]		PL Cache		RP Cache		New-cache		Random Fill Cache		CEASER Cache		SCATTER Cache		Non Determ-inistic Cache[13]		Cache Disabled	
Cache Internal Collision	$A_{inv} \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10]	✓[10]	✓[10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	×	×	×	×	✓	✓	×	×	×[3]	×[3]	○	○	✓	✓
	$V_{inv} \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	×	×	×	×	✓	✓	×	×	×[3]	×[3]	○	○	✓	✓
	$A_a^{inv} \rightsquigarrow V_u \rightsquigarrow V_a$ (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]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×[3]	×[3]	○	○	✓	✓
	$V_{inv} \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	×	×	×	×	✓	✓	✓	✓	×	×	×	×	✓[10]	✓[10,]	✓[10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×[3]	×[3]	○	○	✓	✓
	$A_a^{inv} \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$V_a^{inv} \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	×	×	×	×	✓	✓	✓	✓	×	×	×	×	✓[10]	✓[10,]	✓[10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$A_d \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓[8]	✓[8]	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$V_d \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	×	×	×	×	✓	✓	✓	✓	×	×	×	×	✓[10]	✓[10,]	✓[10]	✓	×	✓	✓[8]	✓[8]	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$A_a^{alias} \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓[10]	✓[10]	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$V_a^{alias} \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	×	×	×	×	✓	✓	✓	✓	×	×	×	×	✓[10]	✓[10,]	✓[10]	✓	×	✓	✓[10]	✓[10]	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
Reload + Time	$V_u^{inv} \rightsquigarrow A_a \rightsquigarrow V_u$ (fast)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$V_u^{inv} \rightsquigarrow V_a \rightsquigarrow V_u$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×[3]	×[3]	○	○	✓	✓
Flush + Probe	$A_a \rightsquigarrow V_u^{inv} \rightsquigarrow A_a$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$A_a \rightsquigarrow V_u^{inv} \rightsquigarrow V_a$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$V_a \rightsquigarrow V_u^{inv} \rightsquigarrow A_a$ (slow)	×	×	×	×	✓	✓	×	×	×	×	×	×	✓[11]	✓[11]	✓[11]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$V_a \rightsquigarrow V_u^{inv} \rightsquigarrow V_a$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×[3]	×[3]	○	○	✓	✓
Evict + Time	$V_u \rightsquigarrow A_d \rightsquigarrow V_u$ (slow)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓[4]	✓[4]	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓[8]	✓[8]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$V_u \rightsquigarrow A_a \rightsquigarrow V_u$ (slow)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓[4]	✓[4]	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓[8]	✓[8]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
Prime + Probe	$A_d \rightsquigarrow V_u \rightsquigarrow A_d$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓[4]	✓[4]	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓[8]	✓[8]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$A_a \rightsquigarrow V_u \rightsquigarrow A_a$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓[4]	✓[4]	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓[10]	✓[10]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
Bernstein's Attack	$V_u \rightsquigarrow V_a \rightsquigarrow V_u$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$V_u \rightsquigarrow V_d \rightsquigarrow V_u$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓[8]	✓[8]	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$V_d \rightsquigarrow V_u \rightsquigarrow V_d$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓[8]	✓[8]	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$V_a \rightsquigarrow V_u \rightsquigarrow V_a$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
Evict + Probe	$V_d \rightsquigarrow V_u \rightsquigarrow A_d$ (slow)	×	×	×	×	✓	✓	×	×	×	×	×	×	✓[10]	✓[10,]	✓[10]	✓	×	✓	✓[8]	✓[8]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$V_a \rightsquigarrow V_u \rightsquigarrow A_a$ (slow)	×	×	×	×	✓	✓	×	×	×	×	×	×	✓[10]	✓[10,]	✓[10]	✓	×	✓	✓[10]	✓[10]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
Prime + Time	$A_d \rightsquigarrow V_u \rightsquigarrow V_d$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓[4]	✓[4]	✓[10]	✓[10]	✓[10]	✓	×	✓	✓[8]	✓[8]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$A_a \rightsquigarrow V_u \rightsquigarrow V_a$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓[4]	✓[4]	✓[10]	✓[10]	✓[10]	✓	×	✓	✓[10]	✓[10]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
Flush + Time	$V_u \rightsquigarrow A_a^{inv} \rightsquigarrow V_u$ (slow)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓[4]	✓[4]	✓[10]	✓[10,]	✓[10]	✓	×	✓	✓[8]	✓[8]	✓	✓	×	×	✓	✓	✓	✓	✓	✓	×	×	×	×	✓[12]	✓[12]	○	○	✓	✓
	$V_u \rightsquigarrow V_a^{inv} \rightsquigarrow V_u$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓[8]	✓[8]	×	×	×	×	✓	✓	✓	✓	✓	✓	×	×	×	×	×[3]	×[3]	○	○	✓	✓

[a]Dynamic adjustment of ways for different threads is assumed to be properly used according to the running program's cache usage

[b]Some software assumptions listed in the entries in this column have been implemented by the cache's related software

[c]Flush is disabled, but cache coherence might be used to do the data removal

[d]For L1 cache and TLB, flushing is done during context switch

[e]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

[f]The technique is now only implemented in last-level cache

[g]The technique now only targets shared cache

[h]The technique only targets inclusion last-level cache

[i]The technique targets data cache hierarchy

[j]For the last-level cache, cache is partitioned between the victim and the attacker

[k]The technique can control the probabilities of the vulnerability to be successful to be extremely small

[l]The technique can work in shared, read only memory while not working in shared, writable memory

[m]Random delay but not random mapping can only decrease the probabilities of attacker in some limited degree

Table 5 Existing secure caches' protection against all possible timing-based vulnerabilities with last step to be invalidation related operations. Single ✓ in a green cell means this cache can prevent the corresponding vulnerability. ○ in a pink cell means this cache can prevent the corresponding vulnerability in some degree. 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		SCATTER Cache		Non Deterministic Cache[13]		Cache Disabled	
Cache Internal Collision Invalidation	$A_{inv} \rightsquigarrow V_u \rightsquigarrow V_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10]	✓[10]	✓[10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×[3]	×[3]	○	○	✓	✓
	$V_{inv} \rightsquigarrow V_u \rightsquigarrow V_a^{inv}$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×[3]	×[3]	○	○	✓	✓
	$A_d \rightsquigarrow V_u \rightsquigarrow V_a^{inv}$ (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]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	○	○	✓	✓
	$V_a^{inv} \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	○	○	✓	✓
	$A_a^{inv} \rightsquigarrow V_u \rightsquigarrow V_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10]	✓[10]	✓[10]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×[3]	×[3]	○	○	✓	✓
	$V_a^{inv} \rightsquigarrow V_u \rightsquigarrow V_a^{inv}$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×[3]	×[3]	○	○	✓	✓
Flush + Reload Invalidation	$A_{inv} \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$V_{inv} \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$A_d \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$V_d \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$A_a^{alias} \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$V_a^{alias} \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (slow)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
Reload + Time Invalidation	$V_u^{inv} \rightsquigarrow A_a \rightsquigarrow V_u^{inv}$ (slow)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$V_u^{inv} \rightsquigarrow V_a \rightsquigarrow V_u^{inv}$ (slow)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×[3]	×[3]	○	○	✓	✓
Flush + Probe Invalidation	$A_a \rightsquigarrow V_u^{inv} \rightsquigarrow A_a^{inv}$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$A_a \rightsquigarrow V_u^{inv} \rightsquigarrow V_a^{inv}$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$V_a \rightsquigarrow V_u^{inv} \rightsquigarrow A_a^{inv}$ (fast)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓	✓	✓[11]	✓[11]	✓[11]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓[12]	✓[12]	○	○	✓	✓
	$V_a \rightsquigarrow V_u^{inv} \rightsquigarrow V_a^{inv}$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×[3]	×[3]	○	○	✓	✓
Evict + Time Invalidation	$V_u \rightsquigarrow A_d \rightsquigarrow V_u^{inv}$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓[4]	✓[4]	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$V_u \rightsquigarrow A_a \rightsquigarrow V_u^{inv}$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓[4]	✓[4]	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
Prime + Probe Invalidation	$A_d \rightsquigarrow V_u \rightsquigarrow A_d^{inv}$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓[4]	✓[4]	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓[8]	✓[8]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$A_a \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓[4]	✓[4]	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
Bernstein's Invalidation Attack	$V_u \rightsquigarrow V_a \rightsquigarrow V_u^{inv}$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$V_u \rightsquigarrow V_d \rightsquigarrow V_u^{inv}$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$V_d \rightsquigarrow V_u \rightsquigarrow V_d^{inv}$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓[8]	✓[8]	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$V_a \rightsquigarrow V_u \rightsquigarrow V_a^{inv}$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
Evict + Probe Invalidation	$V_d \rightsquigarrow V_u \rightsquigarrow A_d^{inv}$ (fast)	×	×	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓[10]	✓[10,]	✓[10]	✓	×	✓	✓[8]	✓[8]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$V_a \rightsquigarrow V_u \rightsquigarrow A_a^{inv}$ (fast)	×	×	✓	✓	✓	✓	×	×	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
Prime + Time Invalidation	$A_d \rightsquigarrow V_u \rightsquigarrow V_d^{inv}$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓[4]	✓[4]	✓[10]	✓[10]	✓[10]	✓	×	✓	✓[8]	✓[8]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
	$A_a \rightsquigarrow V_u \rightsquigarrow V_a^{inv}$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓[11]	✓[11]	○	○	✓	✓
Flush + Time Invalidation	$V_u \rightsquigarrow A_a^{inv} \rightsquigarrow V_u^{inv}$ (fast)	×	×	✓	✓	✓	✓	✓	✓	✓[1]	✓[1]	✓	✓	✓[10,]	✓[10,]	✓[10,]	✓	×	✓	✓	✓	✓	✓	×	×	✓	✓	✓	✓	✓	✓	×	×	×	×	✓[12]	✓[12]	○	○	✓	✓
	$V_u \rightsquigarrow V_a^{inv} \rightsquigarrow V_u^{inv}$ (fast)	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓	×	×	×	×	✓	✓	✓	✓	✓	✓	×	×	×	×	×[3]	×[3]	○	○	✓	✓

[a]Dynamic adjustment of ways for different threads is assumed to be properly used according to the running program's cache usage

[b]Some software assumptions listed in the entries in this column have been implemented by the cache's related software

[c]Flush is disabled, but cache coherence might be used to do the data removal

[d]For L1 cache and TLB, flushing is done during context switch

[e]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

[f]The technique is now only implemented in last-level cache

[g]The technique now only targets shared cache

[h]The technique only targets inclusion last-level cache

[i]The technique targets data cache hierarchy

[j]For the last-level cache, cache is partitioned between the victim and the attacker

[k]The technique can control the probabilities of the vulnerability to be successful to be extremely small

[l]The technique can work in shared, read only memory while not working in shared, writable memory

[m]Random delay but not random mapping can only decrease the probabilities of attacker in some limited degree

utilize: differentiating sensitive data, partitioning, and randomization.

Differentiating Sensitive Data (columns for CATalyst cache to columns for Random Fill cache in Tables 4 and 5) allows the victim or attacker software or management software to explicitly label a certain range of the data of victim which they think is 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. For example, 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 the 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 [3]) 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 [34]) 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 Tables 4 and 5). For example, 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 [49]. Dynamic partitioning can help limit the negative performance and space impacts, but it could be at a cost of revealing some 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 [5]) 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 [55]). 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 [3]) 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, it 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 with 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 space in the cache will be either belongs to *High* 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 [55]). 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 focuses 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 [3]) 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 [5]) vulnerability is prevented.

Randomization-Based Caches (columns for SHARP cache, and columns for RP cache to columns for Non Deterministic cache in Tables 4 and 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 [23]. 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 [3]) 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 [5]). 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 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 prevent hit-based vulnerabilities such as $V_a \rightsquigarrow V_u^{inv} \rightsquigarrow V_a^{inv}$ (fast) (one type of Flush + Probe Invalidation). SCATTER cache encrypts both the cache address and process ID when mapping into different cache index to further prevent more hit-based vulnerabilities for shared and read only memory. 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 designers in the different papers. At the extreme end, there is

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

Metric		Set Associative Cache	SP* Cache [27, 20]	SecVerilog Cache [57, 56]	SecDCP Cache [48]	NoMo Cache [12]	SHARP Cache [53]	Sanctum Cache [10]	MI6 Cache[7]	InvisiSpec Cache [52]	CATalyst Cache [29]	DAWG Cache [25]	RIC [22]	PL Cache [49]	RP Cache [49]	Newcache [50, 31]	Random Fill Cache [30]	CEASER Cache [38]	SCATTER Cache[51]	Non Deterministic Cache [23]	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	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	16-way 2MB	–	–
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	SPEC 2017	AES cryptographic algorithm	–
Implementation		–	–	MIPS processor	Gem5 simulator [4]	Pin [32] based trace-driven x86 simulator	MARSS [35] cycle-level full-system simulator	Rocket Chip Generator [28]	RiscyOO processor [58] + Xilinx FPGA	Gem5 simulator + CACTI 5 [45]	Intel Xeon E5 2618L v3 processors	zsim [39] execution driven x86-64 simulator and Haswell [37] hardware	Cacti [43] version 6.5	M-Sim v2.0 [42]	M-Sim v2.0	CACTI 5.0	Gem5 simulator	Pin-based x86 simulator	Gem5 simulator	HotLeakage simulator [60]	–
Performance Overhead		–	1%	–	12.5% better over static cache partitioning	1.2% average, 5% worst	3%-4%	–	–	reduce the execution slowdown 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	3.5% 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%	–	–	–	–	–	–	–	–

the Non Deterministic cache: with random delay, the secure cache can prevent all the cache timing–based vulnerabilities in some degree—while their paper reports only 7% degradation in performance, we expect it to be much more 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. For example, more security seems to imply less performance.

6.2 Towards Ideal Secure Cache

Based on the above analysis, a good secure cache should consider all the 72 types of *Strong* vulnerabilities, e.g., external and internal interference, and hit-based and miss-based vulnerabilities. Considering all factors and based on Tables 4 and 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 [3]) 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 [34]) 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 attacks, all levels of caches in the hierarchy should be protected with related security features. For example, 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 are a lot of existing attacks exploring timing-based cache channels, e.g., [1, 3, 5, 16–19, 34, 36, 54, 55]. Furthermore, our recent paper [11] has summarized cache timing–based side-channel vulnerabilities using a three-step model, and inspired this work on checking which vulnerability types are truly defeated by the secure caches in context of timing-based attacks. In other work, Zhang and Lee [59] used finite-state machine to model cache architectures and leveraged mutual information to measure potential side-channel leakage of the modeled cache architectures. Meanwhile, He and Lee [20] 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 possible vulnerabilities due to cache timing–based channels.

There is also some other work focusing on cache side-channel verification [13, 14, 47]. Among these, CacheAudit [14] efficiently computes possible side-channel observations using abstractions in a modular way. Bit-level and arithmetic reasoning is used in [13] for memory accesses in the presence of dynamic memory allocation. CacheD [47] detects potential cache differences at each program point leveraging symbolic execution and constraint solving.

Hardware transactional memory has also been leveraged to prevent timing-based cache side-channel attacks [8, 15]. 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 [15], 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 [8], 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 three-step model in order to model all possible cache timing vulnerabilities. It further provided a cache three-step simulator and reduction rules to derive effective vulnerabilities, allowing us to find ones that have not been exploited in literature. With exhaustive effective vulnerability types listed, this paper presented analysis of 18 secure processor cache designs with respect to how well they can defend against these timing-based 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 attacks. However, it requires design of an ideal secure cache, or correction of existing secure cache designs to eliminate the few attacks that they do not protect against.

Acknowledgments This work was support in part by NSF grants 1651945 and 1813797; and through Semiconductor Research Corporation (SRC) under GRC Task 2844.001.

Appendix A: Attack Strategies Descriptions

This appendix gives overview of the attack strategies, shown in Tables 2 and 3 in Section 3. For each attack strategy, an overview of the three steps of the strategy is given. Some of the strategies are similar, and some may not be precise, but we keep and use the original names as they were assigned in prior work. One advantage of our three-step model is that it gives precise definition of each attack. Nevertheless, the attack strategy names used before (and added by us for strategies which did not have such names) may be useful to recall the attacks' high-level operation.

Cache Internal Collision In $Step$ 1, cache block's data is invalidated by flushing or eviction done by either the attacker or the victim. Then, the victim accesses secret data in $Step$ 2. Finally, the victim accesses data at a known address in $Step$ 3, if there is a cache hit, then it reveals that there is an internal collision and leaks value of u.

Flush + Reload In $Step$ 1, either the attacker or the victim invalidates the cache block's data by flushing or eviction. Then, the victim access secret data in $Step$ 2. Finally, the attacker tries to access some data in $Step$ 2 using a known address. If a cache hit is observed, then addresses from last two steps are the same, and the attacker learns the secret address. This strategy has similar $Step$ 1 and $Step$ 2 as **Cache Internal Collision** vulnerability, but for $Step$ 3, it is the attacker who does the reload access.

Reload + Time (New Name Assigned in this Paper) In $Step$ 1, secret data is invalidated by the victim. Then, the attacker does some known data access in $Step$ 2 that could possibly bring back the invalidated the victim's secret data in $Step$ 1. In $Step$ 3, if the victim reloads the secret data, a cache hit is observed and the attacker can derive the secret data's address.

Flush + Probe (New Name Assigned in this Paper) In $Step$ 1, the victim or the attacker access some known address. In $Step$ 2, the victim invalidates secret data. In $Step$ 3, reloading of $Step$ 1's data and observation of a cache miss will help the attacker learn that the secret data maps to the known address from $Step$ 1.

Evict + Time In $Step$ 1, some victim's secret data is put into the cache by the victim itself. In $Step$ 2, the attacker evicts a specific cache set by performing a memory related operation that is not a flush. In $Step$ 3, the victim reloads secret data, and if a cache miss is observed, the will learn the secret data's cache set information. This attack has similar $Step$ 1 and $Step$ 3 as **Flush + Time** vulnerability, but for $Step$ 2, in **Evict + Time**, the attacker invalidates some known address allowing it to find the full address of the secret data, instead of evicting a cache set to only find the secret data's cache index as in the **Flush + Time** attack.

Prime + Probe In $Step$ 1, the attacker primes the cache set using data at address known to the attacker. In $Step$ 2, the victim accesses the secret data, which possibly evicts data from $Step$ 1. In $Step$ 3, the attacker probes each cache set and if a cache miss is observed, the attacker knowns the secret data maps to the cache set he or she primed.

Bernstein's Attack This attack strategy leverages the victim's internal interference to trigger the miss-based attack. For one case, the victim does the same secret data access in $Step$ 1 and $Step$ 3 while in $Step$ 2, the victim tries to evict one whole cache set's data by known data accesses. If cache miss is observed in $Step$ 3, that will tell the attacker the cache set is the one secret data maps to. For another case, the victim primes and probe a cache set in $Step$ 1 and $Step$ 3 driven by the attacker while in $Step$ 2, the victim tries to access the secret data. Similar to the first case, observing cache miss in $Step$ 3 tells the attacker the cache set is the one secret data maps to.

Evict + Probe (New Name Assigned in this Paper) In $Step$ 1, victim evict the cache set using the access to a data at an address known to the attacker. In $Step$ 2, the victim accesses secret data, which possibly evicts data from $Step$ 1. In $Step$ 3, the attacker probes each cache set using the same data as in $Step$ 1, if a cache miss is observed the attacker knowns the secret data maps to the cache set he or she primed. This attack strategy has similar $Step$ 2 and $Step$ 3 as **Prime + Probe** attack, but for $Step$ 1, it is the victim that does the eviction accesses.

Prime + Time (New Name Assigned in this Paper) In $Step$ 1, the attacker primes the cache set using access to data at an address known to the attacker. In $Step$ 2, the victim accesses secret data, which possibly evicts data from $Step$ 1. In $Step$ 3, the victim probes each cache set using the same data $Step$ 1, if a cache miss is observed the attacker knowns the secret data maps to the cache set he or she primed in $Step$ 1. This attack strategy has similar $Step$ 1 and $Step$ 2 as **Prime + Probe** attack, but for $Step$ 3, it is the victim that does the probing accesses.

Flush + Time (New Name Assigned in this Paper) The victim accesses the same secret data in $Step$ 1 and $Step$ 3; while in $Step$ 2, the attacker tries to invalidate data at a known address. If cache miss is observed in $Step$ 3, that will tell the attacker the data address he or she invalidated in $Step$ 2 maps to the secret data.

Invalidation related **(new names assigned in this paper):** Vulnerabilities that have names ending with "invalidation" in Table 3 correspond to the vulnerabilities that have the same name (except for the "invalidation" part) in Table 2. The difference between each set of corresponding vulnerabilities is that the vulnerabilities ending with "invalidation" use invalidation related operation in the last step to derive the timing information, rather than the normal memory access related operations.

Appendix B: Soundness Analysis 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 timing-based cache vulnerabilities in normal caches. If there is a vulnerability that is represented using more than three steps, the steps can be reduced to only three steps, or a three-step sub-pattern can be found in the longer representation.

In the below analysis, we use β to denote the number of memory-related operations, i.e., steps, in a representation of a vulnerability. We show that $\beta = 1$ is not sufficient to represent a vulnerability, $\beta = 2$ covers some vulnerabilities but not all, $\beta = 3$ represents all the vulnerabilities, and $\beta > 3$ can be reduced to only three steps, or a three-step sub-pattern can be found in the longer representation. Known addresses refer to all the cache states that interference with the data a, a^{alias} and d Unknown address refers u. An access to a known memory address is denoted as $known\_access\_operation$, and an invalidation of a known memory address is denoted as $known\_inv\_operation$. The $known\_access\_operation$ and $known\_inv\_operation$ together make up $not_u_operation$s. An unknown memory related operation (containing u) is denoted as $u\_operation$.

B.1 Patterns with $\beta = 1$

When $\beta = 1$, there is only one memory-related operation, and it is not possible to create 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 $Step$ 1 and $Step$ 2 being $\star$, since the cache state $\star$ gives no information, and $Step$ 3 being the one operation. These types of patterns are all examined by the cache three-step simulator and none of these types are found to be effective. Consequently, a vulnerability cannot exit when $\beta = 1$.

B.2 Patterns with $\beta = 2$

When $\beta = 2$, it satisfies the minimum requirement of an interference for memory related operations and corresponds to the three-step cases where $Step$ 1 is $\star$, and $Step$ 2 and $Step$ 3 are the two operations. These types are all examined by the cache three-step simulator and some of them belong to *Weak Vulnerabilities*, like $\{\star \rightsquigarrow A_a \rightsquigarrow V_u\}$. Therefore, three-step cases where $Step$ 1 is $\star$ have corresponding effective vulnerabilities shown in Table 2. Consequently, $\beta = 2$ can represent some weak vulnerabilities, but not all vulnerabilities as there exist some that are represented with three steps, as discussed next.

B.3 Patterns with $\beta = 3$

When $\beta = 3$, we have tested all possible combinations of three-step memory related operations in Section 3.3 using our cache simulator for the three-step model. We found that there are in total 72 types of *Strong Vulnerabilities* and 64 types of *Weak Vulnerabilities* that are represented by patterns with $\beta = 3$ steps. Consequently, $\beta = 3$ can represent all the vulnerabilities (including some weak ones where $Step$ 1 is $\star$). Using more steps to represent vulnerabilities is not necessary, as discussed next.

B.4 Patterns with $\beta > 3$

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

B.4.1 Subdivision Rules

First, a set of subdivision rules is used to divide the long pattern into shorter patterns, following the below rules. Each subdivision rule should be applied recursively before applying the next rule.

Subdivision Rule 1: If the longer pattern contains a sub-pattern such as $\{... \rightsquigarrow \star \rightsquigarrow ...\}$, the longer pattern can be divided into two separate patterns, where $\star$ is assigned as $Step$ 1 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 applied until there are no sub-patterns left with a $\star$ in the middle or as last step ($\star$ in the last step will be deleted) in the longer pattern.

Subdivision Rule 2: Next, if a pattern (derived after recursive application of the *Rule 1* contains a sub-pattern such as $\{... \rightsquigarrow A_{inv}/V_{inv} \rightsquigarrow ...\}$, the longer pattern can be divided into two separate patterns, where A_{inv}/V_{inv} is assigned as $Step$ 1 of the second pattern. This is because A_{inv}/V_{inv} will flush all the timing information of the current block and it can be used as the flushing step for $Step$ 1, e.g., vulnerability $\{A_{inv} \rightsquigarrow V_u \rightsquigarrow A_a(fast)\}$ shown in Table 2. A_{inv}/V_{inv} cannot be a candidate for middle steps or the last step because it will flush all timing information, making the attacker unable to deduce the final timing with victim's sensitive address translation information. This rule should be recursively applied until there are no sub-patterns left with a A_{inv}/V_{inv} in the middle or the last step (A_{inv}/V_{inv} in the last step will be deleted).

B.4.2 Simplification Rules

For each of the patterns resulting from the subdivision of the original pattern, we define *Commute Rule*s, *Union Rule*s, and *Reduction Rule*s for a each set of two adjacent steps in these remaining patterns. In Table 7, we show all the possible cases of the rule applying conditions for each adjacent two steps, regardless of the attacker's access (A) or the victim's access (V). The table shows whether the corresponding two steps can be commuted, reduced or unioned (and the reduced or the unioned result if the rules can be applied).

B.4.2.1 Commute Rules

Suppose there are two adjacent steps M and N for a memory sequences $\{... \rightsquigarrow M \rightsquigarrow N \rightsquigarrow ...\}$. If commuting M and N lead to the same observation result, i.e., $\{... \rightsquigarrow M \rightsquigarrow N \rightsquigarrow ...\}$ and $\{... \rightsquigarrow N \rightsquigarrow M \rightsquigarrow ...\}$ will have the same timing observation information in the final step for the attacker, we can freely exchange the place of M and N in this pattern. In this case, we have more chance to *Reduce* and *Union* the steps within the memory sequence by the following *Rule*s. In the possible commuting process, we will try every possible combinations to commute different pairs of two steps that are able to apply the *Commute Rule*s and then further apply *Reduce Rule*s and *Union Rule*s to see whether the commute is effective, i.e., there can be steps reduced or unioned after the proper commuting process. The following two adjacent memory-related operations can be commuted:

- *Commute Rule 1*: For two adjacent steps, if one step is a $known\_access\_operation$ and another step is a $known\_inv\_operation$. and the addresses they refer to are different, these two steps can be commuted no matter which position of the two steps they are in within the whole memory sequence. It will show a "yes" for the corresponding two-step pattern for the *Commute Rule 1* column if these two can be commuted in Table 7.
- *Commute Rule 2*: A superset of two-step patterns that can apply *Commute Rule 1* can be commuted if the second step of these two adjacent steps is not the last step in the whole memory sequence. There are some two adjacent steps that can only be commuted if the second step of these two adjacent steps is not the last step in the whole memory sequence. There will be a "yes" for the corresponding two-step pattern for the *Commute Rule 2* column and a "no" for the corresponding two-step pattern for the *Commute Rule 1* column in Table 7.

B.4.2.2 Reduction Rules

If the memory sequence after applying *Commute Rule*s have a sub-pattern that has two adjacent steps both related to known addresses or both related to unknown address (including repeating states), the two adjacent steps can be reduced to only one following the reduction rules (if the two-step pattern has "yes" for the Column "*Union Rule or*

Table 7 Rules for combining two adjacent steps

First	Second	*Commute Rule 1*	*Commute Rule 2*	*Union Rule* or *Reduce Rule*	Combined Step	First	Second	*Commute Rule 1*	*Commute Rule 2*	*Union Rule* or *Reduce Rule*	Combined Step
a	a	yes	yes	yes	a	a^{inv}	a	no	no	yes	a
a	a^{alias}	no	no	yes	a^{alias}	a^{inv}	a^{alias}	yes	yes	yes	a^{alias}
a	d	no	no	yes	d	a^{inv}	d	yes	yes	yes	d
a	u	no	no	no	–	a^{inv}	u	no	no	no	–
a	a^{inv}	no	no	yes	a^{inv}	a^{inv}	a^{inv}	yes	yes	yes	a^{inv}
a	${a^{alias}}^{inv}$	yes	yes	yes	a	a^{inv}	${a^{alias}}^{inv}$	yes	yes	yes	$Union(a^{inv}, {a^{alias}}^{inv})$
a	d^{inv}	yes	yes	yes	a	a^{inv}	d^{inv}	yes	yes	yes	$Union(a^{inv}, d^{inv})$
a	u^{inv}	no	no	no	–	a^{inv}	u^{inv}	no	yes	no	–
a^{alias}	a	no	no	yes	a	${a^{alias}}^{inv}$	a	yes	yes	yes	a
a^{alias}	a^{alias}	yes	yes	yes	a^{alias}	${a^{alias}}^{inv}$	a^{alias}	no	no	yes	a^{alias}
a^{alias}	d	no	no	yes	d	${a^{alias}}^{inv}$	d	yes	yes	yes	d
a^{alias}	u	no	no	no	–	${a^{alias}}^{inv}$	u	no	no	no	–
a^{alias}	a^{inv}	yes	yes	yes	a^{alias}	${a^{alias}}^{inv}$	a^{inv}	yes	yes	yes	$Union(a^{inv}, {a^{alias}}^{inv})$
a^{alias}	${a^{alias}}^{inv}$	no	no	yes	${a^{alias}}^{inv}$	${a^{alias}}^{inv}$	${a^{alias}}^{inv}$	yes	yes	yes	${a^{alias}}^{inv}$
a^{alias}	d^{inv}	yes	yes	yes	a^{alias}	${a^{alias}}^{inv}$	d^{inv}	yes	yes	yes	$Union(d^{inv}, {a^{alias}}^{inv})$
a^{alias}	u^{inv}	no	no	no	–	${a^{alias}}^{inv}$	u^{inv}	no	yes	no	–
d	a	no	no	yes	a	d^{inv}	a	yes	yes	yes	a
d	a^{alias}	no	no	yes	a^{alias}	d^{inv}	a^{alias}	yes	yes	yes	a^{alias}
d	d	yes	yes	yes	d	d^{inv}	d	no	no	yes	d
d	u	no	no	no	–	d^{inv}	u	no	yes	no	–
d	a^{inv}	yes	yes	yes	d	d^{inv}	a^{inv}	yes	yes	yes	$Union(a^{inv}, d^{inv})$
d	${a^{alias}}^{inv}$	yes	yes	yes	d	d^{inv}	${a^{alias}}^{inv}$	yes	yes	yes	$Union(d^{inv}, {a^{alias}}^{inv})$
d	d^{inv}	no	no	yes	d^{inv}	d^{inv}	d^{inv}	yes	yes	yes	d^{inv}
d	u^{inv}	no	yes	no	–	d^{inv}	u^{inv}	no	yes	no	–
u	a	no	no	no	–	u^{inv}	a	no	no	no	–
u	a^{alias}	no	no	no	–	u^{inv}	a^{alias}	no	no	no	–
u	d	no	no	no	–	u^{inv}	d	no	yes	no	–
u	u	yes	yes	yes	u	u^{inv}	u	no	no	yes	u
u	a^{inv}	no	no	no	–	u^{inv}	a^{inv}	no	yes	no	–
u	${a^{alias}}^{inv}$	no	no	no	–	u^{inv}	${a^{alias}}^{inv}$	no	yes	no	–
u	d^{inv}	no	yes	no	–	u^{inv}	d^{inv}	no	yes	no	–
u	u^{inv}	no	no	yes	u^{inv}	u^{inv}	u^{inv}	yes	yes	yes	u^{inv}

Reduce Rule" and has no *Union* result for the "Combined Step" Column in Table 7.

- *Reduction Rule 1*: For two $u_operation$s, although u is unknown, both of the accesses target on the same u so can be reduced to only keep the second access in the memory sequence.
- *Reduction Rule 2*: For two known adjacent memory access–related operations ($known\_access\_operation$), they always result in a deterministic state of the second memory access–related cache block, so these two steps can be reduced to only one step.
- *Reduction Rule 3*: For two adjacent steps, if one step is $known\_access\_operation$ and another one is $known\_inv\_operation$, no matter what order they have, and the address they refer to is the same, these two can be reduced to one step, which is the second step.

B.4.2.3 Union Rules

Suppose there are two adjacent steps M and N for memory sequences $\{... \rightsquigarrow M \rightsquigarrow N \rightsquigarrow ...\}$. If combining M and N leads to the same timing observation result, i.e., $\{... \rightsquigarrow M \rightsquigarrow N \rightsquigarrow ...\}$ and $\{... \rightsquigarrow Union(M, N) \rightsquigarrow ...\}$ will have the same timing observation information in the final step for the attacker, we can combine step M and N to be a joint one step for this memory sequence, defined as $Union(M, N)$. Two adjacent steps that can be combined are discussed in the following cases:

- *Union Rule 1*: Two invalidations to two known different memory addresses can be applied *Union Rule 1*. $known\_inv\_operation$ are two operations both invalidating some known address; therefore, they can be combined to only one step. The *Union Rule* can be continuously done to union all the adjacent invalidation step that invalidates known different memory addresses.

B.4.2.4 Final Check Rules

Each long memory sequence will recursively apply these three categorizations of the rules in the order: *Commute Rule*s first to put $known_access_operation$s and $known\_inv\_operation$ that targets the same address as near as possible, and $u_operation$s and $not_u_operation$s are putting together as much as possible. The *Reduced Rule*s are then checked and applied to the processed memory sequence to reduce the steps. Then, the *Union Rule* is applied to the processed memory sequence.

The recursion at each application to these three categorizations of the rules should be always applied and reduce at least one step until the resulting sequence matches one of the two possible cases:

- the long ($\beta > 3$) memory sequence with $u\_operation$ and $not\_u\_operation$ is further reduced to a sequence where there are at most three steps in the following patterns, or less:
  - $u\_operation \rightsquigarrow not\_u\_operation \rightsquigarrow u\_operation$
  - $not\_u\_operation \rightsquigarrow u\_operation \rightsquigarrow not\_u\_operation$

  There might be possible extra $\star$ or A^{inv}/V^{inv} before these three-step pattern, where:

  - An extra $\star$ in the first step will not influence the result and can be directly removed.
  - If an extra A^{inv}/V^{inv} in the first step:
    - If followed by $known\_access\_operation$, A^{inv}/V^{inv} can be removed due to the actual state further put into the cache block.
    - If followed by $known\_inv\_operation$ or V_u^{inv}, A^{inv}/V^{inv} can also be removed since the memory location is repeatedly flushed by the two steps.
    - If followed by V_u, worst case will be $A^{inv}/V^{inv} \rightsquigarrow V_u \rightsquigarrow not\_u\_operation \rightsquigarrow u\_operation$, which is either an effective vulnerability according to Table 2 and reduction rules shown in Section 3.3 or $A^{inv}/V^{inv} \rightsquigarrow V_u \rightsquigarrow A_d^{inv}/V_d^{inv} \rightsquigarrow u\_operation$, where $V_u \rightsquigarrow A_d^{inv}/V_d^{inv}$ can further be applied *Commute Rule 2* to reduce and be within three steps.

  In this case, the steps are finally within three steps and the checking is done.
- There exist two adjacent steps that cannot be applied any *Rule*s above and requires the *Rest Checking*.

The only left two adjacent steps that cannot be applied by any of the three categorizations of the *Rule*s are the following:

- $\{... \rightsquigarrow A_a/V_a/A_{a^{alias}}/V_{a^{alias}}/A_d/V_d/A_a^{inv}/V_a^{inv}/A_{a^{alias}}^{inv}/V_{a^{alias}}^{inv} \rightsquigarrow V_u \rightsquigarrow ...\}$
- $\{... \rightsquigarrow A_a/V_a/A_{a^{alias}}/V_{a^{alias}} \rightsquigarrow V_u^{inv} \rightsquigarrow ...\}$
- $\{... \rightsquigarrow V_u \rightsquigarrow\rightsquigarrow ...A_a/V_a/A_{a^{alias}}/V_{a^{alias}}/A_d/V_d/A_a^{inv}/V_a^{inv}/A_{a^{alias}}^{inv}/V_{a^{alias}}^{inv}\}$
- $\{... \rightsquigarrow V_u^{inv} \rightsquigarrow A_a/V_a/A_{a^{alias}}/V_{a^{alias}} \rightsquigarrow ...\}$

We manually checked all of the two adjacent step patterns above and found that adding extra step before or after these

Algorithm 2 β-Step ($\beta > 3$) pattern reduction.

```
Input: $\beta$: 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. $step\_list[1]$, $step\_list[2]$, ... are 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: $reduce\_list = \emptyset$
 2: while $step\_list$.contain($\star$) and $\star$.index not 0 do
 3:     $step\_list = Subdivision\_Rule\_1\ (step\_list)$
 4: end while
 5: while ($step\_list$.contain($A_{inv}$) and $A_{inv}$.index not 0) or ($step\_list$.contain($V_{inv}$) and $V_{inv}$.index not 0) do
 6:     $step\_list = Subdivision\_Rule\_2\ (step\_list)$
 7: end while
 8: while !($step\_list.set\_list$.is_ineffecitve or $step\_list.set\_list$.has_interval_effective_three_steps) do
 9:     $step\_list = Commute\_Rules\ (step\_list)$
10:     $step\_list = Reduction\_Rules\ (step\_list)$
11:     $step\_list = Union\_Rule\ (step\_list)$
12:     if !($step\_list.set\_list$.is_ineffecitve or $step\_list.set\_list$.has_interval_effective_three_steps) then
13:         $reduce\_list$ += $Rest\_Checking\ (step\_list)$
14:     end if
15: end while
16: return $reduce\_list$
```

two steps can either generate two adjacent step patterns that be processed by the three *Rule*s, where further step can be reduced, or construct effective vulnerability according to Table 2 and reduction rules shown in Section 3.3, where the corresponding pattern can be treated effective and the checking is done.

B.4.3 Algorithm for Reducing and Checking Memory Sequence

Algorithm 2 is used to (i) reduce a β-step ($\beta > 3$) pattern to a three-step pattern, thus demonstrating that the corresponding $\beta > 3$ step pattern actually is equivalent to the output three-step pattern and represents a vulnerability that is captured by an existing three-step pattern, or (ii) demonstrate that the β-step pattern can be mapped to one or more three-step vulnerabilities. It is not possible for a β-step vulnerability pattern to not be either (i) or (ii) after doing the *Rule* applications Key outcome of our analysis is that any β-step pattern is not a vulnerability, or if it is a vulnerability it maps to either outputs (i) or (ii) of the algorithm.

Inside the Algorithm 2, contain() represents a function to check if a list contains a corresponding state, is_ineffective() represents a function that checks the corresponding memory sequence does not contain any effective three-steps. has_interval_effective_three_steps() represents a function that check if the corresponding memory sequence can be mapped to one or more three-step vulnerabilities.

B.4.4 Summary

In conclusion, the three-step model can model all possible timing-based cache vulnerability in normal caches. Vulnerabilities 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.

References

1. Acıiçmez O, Koç ÇK (2006) Trace-driven cache attacks on AES (short paper). In: International conference on information and communications security. Springer, pp 112–121
2. Agrawal D, Archambeault B, Rao JR, Rohatgi P (2002) The EM side-channel (s). In: International workshop on cryptographic hardware and embedded systems. Springer, pp 29–45
3. Bernstein DJ (2005) Cache-timing attacks on AES
4. Binkert N, Beckmann B, Black G, Reinhardt SK, Saidi A, Basu A, Hestness J, Hower DR, Krishna T, Sardashti S et al (2011) The gem5 simulator. ACM SIGARCH computer architecture news 39(2):1–7
5. Bonneau J, Mironov I (2006) Cache-collision timing attacks against AES. In: International workshop on cryptographic hardware and embedded systems. Springer, pp 201–215
6. Borghoff J, Canteaut A, Güneysu T, Kavun EB, Knezevic M, Knudsen LR, Leander G, Nikov V, Paar C, Rechberger C et al (2012) Prince–a low-latency block cipher for pervasive computing applications. In: International conference on the theory and application of cryptology and information security. Springer, pp 208–225

7. Bourgeat T, Lebedev I, Wright A, Zhang S, Devadas S et al (2019) MI6: Secure enclaves in a speculative out-of-order processor. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. ACM, pp 42–56
8. Chen S, Liu F, Mi Z, Zhang Y, Lee RB, Chen H, Wang X (2018) Leveraging hardware transactional memory for cache side-channel defenses. In: Proceedings of the 2018 on Asia conference on computer and communications security. ACM, pp 601–608
9. Clark SS, Ransford B, Rahmati A, Guineau S, Sorber J, Xu W, Fu K (2013) Wattsupdoc: power side channels to nonintrusively discover untargeted malware on embedded medical devices. In: Presented as part of the 2013 USENIX workshop on health information technologies
10. Costan V, Lebedev IA, Devadas S (2016) Sanctum: minimal hardware extensions for strong software isolation. In: USENIX security symposium, pp 857–874
11. Deng S, Xiong W, Szefer J (2018) 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. ACM, p. 2
12. Domnitser L, Jaleel A, Loew J, Abu-Ghazaleh N, Ponomarev D (2012) Non-monopolizable caches: low-complexity mitigation of cache side channel attacks. ACM Transactions on Architecture and Code Optimization (TACO) 8(4):35
13. Doychev G, Köpf B (2017) Rigorous analysis of software countermeasures against cache attacks. In: Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation. ACM, pp 406–421
14. Doychev G, Köpf B, Mauborgne L, Reineke J (2015) CacheAudit: a tool for the static analysis of cache side channels. ACM Transactions on information and system security (TISSEC) 18(1):4
15. Gruss D, Lettner J, Schuster F, Ohrimenko O, Haller I, Costa M (2017) Strong and efficient cache side-channel protection using hardware transactional memory. In: USENIX security symposium, pp 217–233
16. Gruss D, Maurice C, Wagner K, Mangard S (2016) Flush+ flush: a fast and stealthy cache attack. In: International conference on detection of intrusions and malware, and vulnerability assessment. Springer, pp 279–299
17. Gruss D, Spreitzer R, Mangard S (2015) Cache template attacks: automating attacks on inclusive last-level caches. In: USENIX security symposium, pp 897–912
18. Guanciale R, Nemati H, Baumann C, Dam M (2016) Cache storage channels: alias-driven attacks and verified countermeasures. In: 2016 IEEE symposium on security and privacy (SP). IEEE, pp 38–55
19. Gullasch D, Bangerter E, Krenn S (2011) Cache games–bringing access-based cache attacks on AES to practice. In: 2011 IEEE symposium on security and privacy (SP). IEEE, pp 490–505
20. He Z, Lee RB (2017) How secure is your cache against side-channel attacks? In: Proceedings of the 50th annual IEEE/ACM international symposium on microarchitecture. ACM, pp 341–353
21. Intel C (2015) Improving real-time performance by utilizing cache allocation technology. Intel Corporation
22. Kayaalp M, Khasawneh KN, Esfeden HA, Elwell J, Abu-Ghazaleh N, Ponomarev D, Jaleel A (2017) RIC: relaxed inclusion caches for mitigating LLC side-channel attacks. In: 2017 54th ACM/EDAC/IEEE design automation conference (DAC). IEEE, pp 1–6
23. Keramidas G, Antonopoulos A, Serpanos DN, Kaxiras S (2008) Non deterministic caches: a simple and effective defense against side channel attacks. Des Autom Embed Syst 12(3):221–230
24. Kessler RE, Hill MD (1992) Page placement algorithms for large realindexed caches. ACM Trans Comput Syst (TOCS) 10(4):338–359
25. Kiriansky V, Lebedev I, Amarasinghe S, Devadas S, Emer J (2018) DAWG: a defense against cache timing attacks in speculative execution processors. In: 2018 51st Annual IEEE/ACM international symposium on microarchitecture (MICRO). IEEE, pp. 974–987
26. Kocher P, Genkin D, Gruss D, Haas W, Hamburg M, Lipp M, Mangard S, Prescher T, Schwarz M, Yarom Y (2018) Spectre attacks: exploiting speculativeexecution. In: 2019 IEEE Symposium on Security and Privacy (SP). IEEE, pp 1–19
27. Lee RB, Kwan P, McGregor JP, Dwoskin J, Wang Z (2005) Architecture for protecting critical secrets in microprocessors. In: ACM SIGARCH computer architecture news, vol 33. IEEE Computer Society, pp 2–13
28. Lee Y, Waterman A, Avizienis R, Cook H, Sun C, Stojanović V, Asanović K (2014) 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. IEEE, pp 199–202
29. Liu F, Ge Q, Yarom Y, Mckeen F, Rozas C, Heiser G, Lee RB (2016) CATalyst: defeating last-level cache side channel attacks in cloud computing. In: 2016 IEEE international symposium on high performance computer architecture (HPCA). IEEE, pp 406–418
30. Liu F, Lee RB (2014) Random fill cache architecture. In: 2014 47th annual IEEE/ACM international symposium on microarchitecture (MICRO). IEEE, pp 203–215
31. Liu F, Wu H, Mai K, Lee RB (2016) Newcache: secure cache architecture thwarting cache side-channel attacks. IEEE Micro 36(5):8–16
32. Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K (2005) Pin: building customized program analysis tools with dynamic instrumentation. In: ACM sigplan notices, vol 40. ACM, pp 190–200
33. Masti RJ, Rai D, Ranganathan A, Müller C., Thiele L, Capkun S (2015) Thermal covert channels on multi-core platforms. In: 24th USENIX security symposium (USENIX security 15), pp 865–880
34. Osvik DA, Shamir A, Tromer E (2006) Cache attacks and countermeasures: the case of AES. In: Cryptographers' track at the RSA conference. Springer, pp 1–20
35. Patel A, Afram F, Chen S, Ghose K (2011) MARSS: A full system simulator for multicore x86 CPUs. In: 2011 48th ACM/EDAC/IEEE design automation conference (DAC). IEEE, pp 1050–1055
36. Percival C (2005) Cache missing for fun and profit. BSDCan 2005, Ottawa, 2005. http://www.daemonology.net/papers/htt.pdf
37. Hammarlund P, Martinez AJ, Bajwa AA, Hill DL, Hallnor E, Jiang H, Dixon M, Derr M, Hunsaker M, Kumar R, Osborne RB (2014) Haswell: The fourth-generation intel core processor. IEEE Micro. 10;34(2):6–20
38. Qureshi MK (2018) CEASER: mitigating conflict-based cache attacks via encrypted-address and remapping. In: 2018 51St annual IEEE/ACM international symposium on microarchitecture (MICRO). IEEE, pp 775–787
39. Sanchez D, Kozyrakis C (2013) ZSIm: fast and accurate microarchitectural simulation of thousand-core systems. In: ACM SIGARCH computer architecture news, vol 41. ACM, pp 475–486
40. Schwarz M, Schwarzl M, Lipp M, Gruss D (2018) NetSpectre: read arbitrary memory over network. In: European Symposium on Research in Computer Security. Springer, pp 279–299
41. Seznec A (1993) A case for two-way skewed-associative caches. ACM SIGARCH computer architecture news 21(2):169–178
42. Sharkey J, Ponomarev D, Ghose K (2005) M-sim: a flexible, multithreaded architectural simulation environment. Techenical report, Department of Computer Science State University of New York at Binghamton. In: Technical Report 2001/2, Compaq Computer Corporation, Aug. 2001

43. Shivakumar P, Jouppi NP (2001) Cacti 3.0: an integrated cache timing, power and area model. In: Technical Report 2001/2, Compaq Computer Corporation, Aug. 2001.
44. Taylor G, Davies P, Farmwald M (1990) The TLB slice-a low-cost high-speed address translation mechanism. In: 17th annual international symposium on computer architecture, 1990. Proceedings. IEEE, pp 355–363
45. Thoziyoor S, Muralimanohar N, Ahn JH, Jouppi NP (2008) CACTI 5.1. Technical Report HPL-2008-20, HP Labs
46. Trippel C, Lustig D, Martonosi M (2018) MeltdownPrime and SpectrePrime: automatically-synthesized attacks exploiting invalidation-based coherence protocols. arXiv:1802.03802
47. Wang S, Wang P, Liu X, Zhang D, Wu D (2017) CacheD: identifying cache-based timing channels in production software. In: 26th USENIX security symposium. USENIX association
48. Wang Y, Ferraiuolo A, Zhang D, Myers AC, Suh GE (2016) SecDCP: secure dynamic cache partitioning for efficient timing channel protection. In: 2016 53nd ACM/EDAC/IEEE design automation conference (DAC). IEEE, pp 1–6
49. Wang Z, Lee RB (2007) New cache designs for thwarting software cache-based side channel attacks. In: ACM SIGARCH computer architecture news, vol 35. ACM, pp 494–505
50. Wang Z, Lee RB (2008) A novel cache architecture with enhanced performance and security. In: 2008 41st IEEE/ACM international symposium on Microarchitecture, 2008. MICRO-41. IEEE, pp 83–93
51. Werner M, Unterluggauer T, Giner L, Schwarz M, Gruss D, Mangard S (2019) ScatterCache: thwarting cache attacks via cache set randomization. In: 28th USENIX security symposium (USENIX Security 19). USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/usenixsecurity19/presentation/werner
52. Yan M, Choi J, Skarlatos D, Morrison A, Fletcher C, Torrellas J (2018) InvisiSpec: making speculative execution invisible in the cache hierarchy. In: 2018 51st annual IEEE/ACM international symposium on microarchitecture (MICRO). IEEE, pp 428–441
53. Yan M, Gopireddy B, Shull T, Torrellas J (2017) 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. ACM, pp 347–360
54. Yao F, Doroslovacki M, Venkataramani G (2018) Are coherence protocol states vulnerable to information leakage? In: 2018 IEEE international symposium on high performance computer architecture (HPCA). IEEE, pp 168–179
55. Yarom Y, Falkner K (2014) FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack. In: USENIX security symposium, pp 719–732
56. Zhang D, Askarov A, Myers AC (2012) Language-based control and mitigation of timing channels. ACM SIGPLAN Not 47(6):99–110
57. Zhang D, Wang Y, Suh GE, Myers AC (2015) A hardware design language for timing-sensitive information-flow security. In: ACM SIGARCH computer architecture news, vol 43. ACM, pp 503–516
58. Zhang S, Wright A, Bourgeat T, Arvind A (2018) Composable building blocks to open up processor design. In: 2018 51st annual IEEE/ACM international symposium on microarchitecture (MICRO). IEEE, pp 68–81
59. Zhang T, Lee RB (2014) New models of cache architectures characterizing information leakage from cache side channels. In: Proceedings of the 30th annual computer security applications conference. ACM, pp 96–105
60. Zhang Y, Parikh D, Sankaranarayanan K, Skadron K, Stan M (2003) HotLeakage: a temperature-aware model of subthreshold and gate leakage for architects. University of Virginia Dept of Computer Science Tech Report CS-2003 5