Understanding the Insecurity of Processor Caches Due to Timing-Based Vulnerabilities

Shuwen Deng, Wenjie Xiong, and Jakub Szefer | Yale University

This article discusses a recently developed test suite for checking timing-based vulnerabilities in processor caches, which has revealed the insecurity of today's processor caches. The susceptibility of caches to these vulnerabilities calls for more research on secure processor caches.

Processor caches are the key component to improving the performance of today's processor. By caching frequently used data, they allow for a significant latency reduction in many memory-related operations. However, caches are of a finite size, and they cannot contain all of the data. This will cause the memory accesses in different cache levels or in the main memory to take varying amounts of time. As a result, the timing of memory operations in data caches, for example, the timing of cache hits versus cache misses, can reveal information about security-critical data. For instruction caches, it may be possible to reveal information about the execution as well.

The Threats of Timing-Based Vulnerabilities in Processor Caches

In general, two types of memory-related operations exhibit timing variations that can be abused to extract sensitive information using timing-based side or covert channels in processor caches. First, memory-access operations, such as loads and stores, can be fast (a cache hit) or slow (a cache miss). Second, invalidation-related operations, such as cache flushes, can also be fast (there is no dirty data in the cache so flush finishes quickly) or slow (there is dirty data in the cache, and it has to be written back, resulting in longer timing).

Digital Object Identifier 10.1109/MSEC.2021.3055799
Date of current version: 24 February 2021

Researchers have previously proposed to use these timing differences in memory-related operations to reveal sensitive information in software execution.[1] These timing-based, side-channel attacks often focus on cryptographic applications, for example, attacks on software using Advanced Encryption Standard (AES) with table lookups, which have the goal of extracting the AES key by analyzing the timing of the table lookups. Further, there are many timing-based, covert-channel attacks in which the sender and the receiver cooperate to break the isolation boundary and leak data.[2] Additionally, timing-based channels have recently been used as a part of the Spectre and Meltdown attacks.[3,4]

Yet, until recently, there has not been a systematic method to analyze whether a cache design is vulnerable to possible types of timing-based, side-channel and covert-channel attacks. While different publications have presented individual attacks on caches, cache security analysis has often been done in an ad hoc manner. One reason for the ad hoc analysis is that there have been no security tests that could be easily used to evaluate different processor caches for potential security vulnerabilities to all of the likely timing-based attacks.

A Security Test Suite for Timing-Based, Side-Channel Vulnerabilities in Caches

In our previous work,[5] we observed that all of the existing timing-based vulnerabilities in caches can be modeled with three "steps" of memory-related operations.

 Copublished by the IEEE Computer and Reliability Societies 1540-7993/21©2021IEEE

Further, since all of the cache blocks are updated following the same cache-state machine logic, it is sufficient to consider only one cache block when evaluating timing vulnerabilities.

Therefore, we proposed a three-step model, focusing on one cache block, to derive a set of possible cache-timing-based vulnerabilities in processors. Our work[5] put forth the first three-step theoretical model, and we recently updated it[6] to derive a set of 88 types of cache-timing-based vulnerabilities in modern processors.

Attack Model and Objectives

In timing-based attacks, there exists a victim and an attacker. The victim holds the security-critical data, such as the AES encryption key. The attacker attempts to learn security-critical data by observing the timing variation of operations, such as different memory accesses. We assume that the victim performs some memory accesses that involve security-critical data, and the goal for the attacker is to determine a particular memory address (or cache index) accessed by the victim. We assume that, to determine this address, the attacker and the victim can share the same cache, and, thus, the attacker can observe the timing of cache-related operations and guess which locations are accessed by the victim. The attacker is assumed to have some additional information. For example, to correlate the memory address or index to values of the security-critical data, he or she could know the specific version of AES implementation used. In our model, we assumed the worst-case scenario, where the attacker is able to derive the previously mentioned information. Being able to measure the timing of the victim is standard in the majority of cache-related attacks.

Our model enumerates all of the possible timing-based attacks in the L1 data cache. Our model assumes a multicore and possibly hyperthreading processor, with a cache hierarchy of a local and remote L1 cache, an L2 cache, and a shared L3 cache (which is possibly divided into different cache slices).

Based on this model, we have generated a test suite containing systematically implemented individual tests to check for timing-based vulnerabilities. The goal of the test suite is to identify the sequences of memory-related operations made by the victim and the attacker that cause timing-based vulnerabilities in caches. The presented tests are not actual security exploits; rather, they implement sequences of memory-related operations that correspond to timing-based vulnerabilities. If a vulnerability is detected, we expect that it could be used for a real attack, but that is not the goal of the test suite. To analyze all of the possible vulnerabilities, we developed the test suite assuming a strong attacker scenario, where the attacker is able to control the synchronization between the victim and itself and is able to measure the victim's timing. Some of the vulnerabilities that our model predicts have been previously exploited in real attacks[1,7–10] while some are new and have not been considered before.

Our test suite uses sequences of instructions (called *steps* in our work[5]), which can lead to an attack if they have timing differences depending on the security-critical data. Each test outputs whether or not there is a statistically significant timing difference that the attacker could observe to extract information from the timing channel.

Modeling Accesses Leading to Vulnerabilities

In our model,[5,6] in step 1, a memory operation is performed, placing the cache in an initial state that is known to the attacker (for example, 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 its initial state. Finally, in step 3, a final memory operation is performed, and the timing of the final operation observed by the attacker reveals some information about the relationship among the addresses from steps 1–3.

We found that there are 17 possible states for each step.[5] This includes the cases in which the addresses are either unknown or known to the attacker and those in which the address is brought into the cache by either the victim or the attacker. Consequently, there are $17 \times 17 \times 17 = 4{,}913$ combinations of three steps in total. As is presented in Figure 1, we developed a cache simulator and a set of reduction rules to process all of the three-step combinations and to decide which ones can indicate an effective vulnerability.

Among the three steps, one or more should be the victim's access to an address that is protected from the attacker (denoted by V_u, where V stands for victim, and u represents the address unknown to the attacker), and the timing is observed in step 3. In the model, there are three possible cases for the address of V_u. The first is a, which represents an address known to the attacker. The second is a^{alias}, which refers to a different address than a but maps to the same cache set as it (thus, it can be used for cache line eviction in the attacks). The third is *not in block* (*NIB*), which refers to an address that does not map to the same cache set as a. If a vulnerability is effective, the attacker can infer whether V_u is a, a^{alias}, or *NIB*, based on the timing observations.

In step 3, the timing is observed. We found 66 possible types of timing observations by measuring the possible timings of reading, writing, or flushing one data address from different levels of the cache hierarchy (L1, L2, and the last-level cache). For details of the derivation process, one can refer to our prior work.[6]

The Derivation of All Possible Vulnerabilities

The exhaustive list of the 4,913 combinations of three-step patterns is first input into the cache simulator, where effective vulnerabilities are derived. The simulator takes all of the 4,913 combinations and 66 types of timing observations as inputs, checks them, and outputs the combinations that belong to preliminary strong vulnerability types. To derive the preliminary strong vulnerabilities, the simulator computes if there is a timing variance that the attacker could leverage.

A timing variance exists if different possible values (a, a^{alias}, and NIB) of u correspond to different timings out of the 66 types. We enumerate all of the possible operations (read/write for access and remote write/flush for invalidation) for a step and consider different timings for each operation. Therefore, each pattern may have different types of timing observations. If, for a specific combination, the attacker is able to unambiguously correlate the timing to one of the three values, he or she is able to learn the value of u. The corresponding combination belongs to the "Strong" type of vulnerability.

The reduction rules are then used to remove repeated and redundant patterns from the preliminary strong vulnerabilities. For example, unknown address u must be in one of the steps. If there is no unknown u in the steps, there is nothing for the attacker to learn. Furthermore, we prove the soundness of the model, and the analysis can be found in our prior work.[5]

The resulting 88 types of vulnerabilities can be further categorized into 27 "Vulnerability Types," listed in Table 1 according to their access pattern of each step. The vulnerability types are not actual attacks but are used to group the susceptibilities based on the common features used for each one. A type can be provided for the attacker to trigger real attacks. Broadly, the type names are either based on previous vulnerabilities proposed in literature or are new ones, which we are the first to propose.

For example, there are vulnerabilities using types that map to the techniques used by existing attacks, such as the Cache Collision[7] vulnerability type, displayed in Figure 2(a). In step 1, the cache block's data are invalidated using one of various methods. Then, the victim accesses the security-critical data in step 2. Finally, in step 3, the victim accesses data at a known address to try to collide with the security-critical data. If there is a fast cache-hit timing in step 3, then it reveals to the attacker that there was an internal collision within the victim. The attacker learns the security-critical data based on the knowledge of whether or not there was an internal collision.

Using our model,[6] we have discovered new vulnerabilities that have not been previously explored in literature. For instance, one of the new weaknesses falls into the "Flush and Time" type, presented in Figure 2(b). This vulnerability requires the victim to perform the access on the same security-critical data in step 1 and step 3. In step 2, the attacker invalidates a known address on the remote core with the use of a flush or write. If the victim's access in step 3 returns a cache miss, and a longer timing is observed by the attacker, the security-critical address maps to the known address in step 2. Otherwise, if there is a shorter timing, the two addresses do not map to each other. In this way, the attacker is able to derive the information of the security-critical address. The details of all of the vulnerability types are in our work.[5,6]

Toward Testing Vulnerabilities on Real Processors

To generate an actual test suite from the theoretical model, we considered different variants of how a processor's memory-related operations could be executed to find all possible scenarios that could lead to attacks. Our model considers that: 1) there are 66 possible timing observations in the cache hierarchy of multicore processors among local and remote cores, 2) the victim and attacker can be running in hyperthreading or time-slicing settings, 3) both read and write operations can be memory accesses for testing for potential vulnerabilities, and 4) two types of cache-invalidation operations are possible—through flush instruction or through cache coherence by writing on a remote core to invalidate the local core's cache lines.

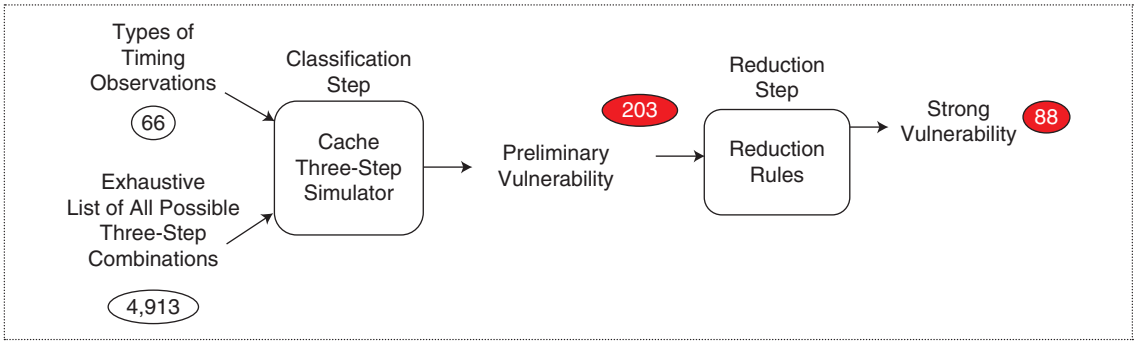

Figure 1. The procedure to derive the effective types of timing-based vulnerabilities. The red ovals refer to the number of vulnerabilities.

Table 1. The evaluation results of different vulnerability types for nine processor configurations.

Vulnerability type	Corresponding vulnerability [6]	Intel Xeon E5-1620	Intel Xeon E5-2667 on-chip	Intel Xeon E5-2667 inter-chip	Intel Xeon E5-2690	Intel Core i5-4570	Intel Xeon E5-2686	Intel Xeon P-8175	AMD FX-8150	AMD EPYC 7571	Found in all tested CPUs	Found in at least one CPU
Cache Collision [7]	1–4	●	●	●	●	●	●	●	●	●	●	●
Flush and Reload [8]	5–8	●	●	●	●	●	●	●	●	●	●	●
Reload and Time [5]	9, 10	●	●	◐	●	●	●	●	●	●	◐	●
Flush and Probe [11]	11–14	●	●	●	●	●	●	●	●	●	●	●
Flush and Time [5]	15, 16	●	●	●	●	●	●	●	●	●	●	●
Cache Coherence Flush and Reload [6]	17–20	●	●	●	●	●	●	●	○	●	○	●
Cache Coherence Prime and Probe [6]	21–28	◐	◐	◐	◐	◐	◐	◐	○	◐	○	◐
Cache Coherence Evict and Time [6]	29–32	◐	●	◐	●	◐	◐	●	○	●	○	●
Bernstein's Attack [1]	33–36	◐	◐	◐	○	◐	●	●	◐	○	○	●
Evict and Probe [5]	37, 38	○	○	○	○	○	○	○	○	○	○	○
Prime and Time [5]	39, 40	●	◐	●	◐	○	◐	●	◐	◐	○	●
Evict and Time [9]	41, 42	◐	◐	○	◐	○	◐	●	○	○	○	●
Prime and Probe [9]	43, 44	●	○	◐	○	○	◐	◐	○	○	○	●
Cache Collision Inv. [5]	45, 46	●	●	●	●	●	●	●	●	●	●	●
Flush and Flush [10]	47–50	●	●	●	●	●	●	●	●	●	●	●
Flush and Reload Inv. [5]	51, 52	●	●	●	●	●	●	●	●	●	●	●
Reload and Time Inv. [5]	53, 54	●	●	●	●	●	●	●	●	●	●	●
Flush and Probe Inv. [5]	55–58	●	●	●	●	●	●	●	●	●	●	●
Flush and Time Inv. [5]	59, 60	●	●	●	●	●	●	●	●	●	●	●
Cache Coherence Flush and Reload Inv. [6]	61–64	●	●	●	●	●	●	●	●	●	●	●
Cache Coherence Prime and Probe Inv. [6]	65–72	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐
Cache Coherence Evict and Time Inv. [6]	73–76	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	●
Bernstein's Inv. [5]	77–80	●	◐	◐	◐	◐	◐	◐	◐	◐	◐	●
Evict and Probe Inv. [5]	81, 82	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐
Prime and Time Inv. [5]	83, 84	●	◐	◐	◐	◐	◐	◐	◐	◐	◐	●
Evict and Time Inv. [5]	85, 86	●	●	●	◐	◐	◐	◐	◐	◐	◐	●
Prime and Probe Inv. [5]	87, 88	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐

Inv.: Invalidation.

As stated previously, there are three steps for each vulnerability, and each step can be one of 17 possible states. Each state can be achieved by two possible means: a read or a write access for a memory-access operation, or a flush or write from the remote core for an invalidation-related operation. Thus, there is a total of $2^3 = 8$ different variants. Additionally, if vulnerabilities contain both the victim and attacker, running either locally or remotely, these two parties can run in either a time-slicing or hyperthreading setting. Consequently, for one vulnerability, there are possibly eight to 16 vulnerability variants. In total, considering different variants, we generated 1,094 test programs that correspond to the 88 types of vulnerabilities. Different test programs correspond to the same vulnerability because one possible state can be achieved by different memory-access or invalidation-related operations, as previously stated.

Security Evaluation of Real Processors

In our test suite,[6] the theoretical steps[5] are translated into concrete assembly instructions to test the corresponding memory operations of vulnerability variants. Given the large number of variants, we wrote scripts to automatically generate the C program for each one of each vulnerability type.

Evaluating if a Cache Is Vulnerable

For the victim's unknown memory operation state (as is discussed in "Modeling Accesses Leading to Vulnerabilities"), it has three candidates: a, a^{alias}, and *NIB*. The tests separately check the timing of each candidate. For all of the tests, the timing of step 3 is used to try to recover the security-critical data. We use read time-stamp counter (rdtsc) instruction in our test suite to measure time. We run it 300 times for each of the variants in which the unknown address is a, a^{alias}, and *NIB*, respectively. Then, we use Welch's t-test[12] to distinguish the distributions of the measured timings for each candidate value. We consider two timing distributions to be significantly different from each other if the probability that the observed data comes from the same distribution is less than 0.05% (this number is set to be very small to reduce false positives).

For a vulnerability to be judged effective on a particular processor, the timing distribution of one of the three candidates for the victim's unknown memory operation (a, a^{alias}, or *NIB*) should be statistically different from the other two candidates. That is, the timing of the execution of the test suite when u is equal to a, a^{alias}, or *NIB*, should be distinguishable. At the end of each test run, the test

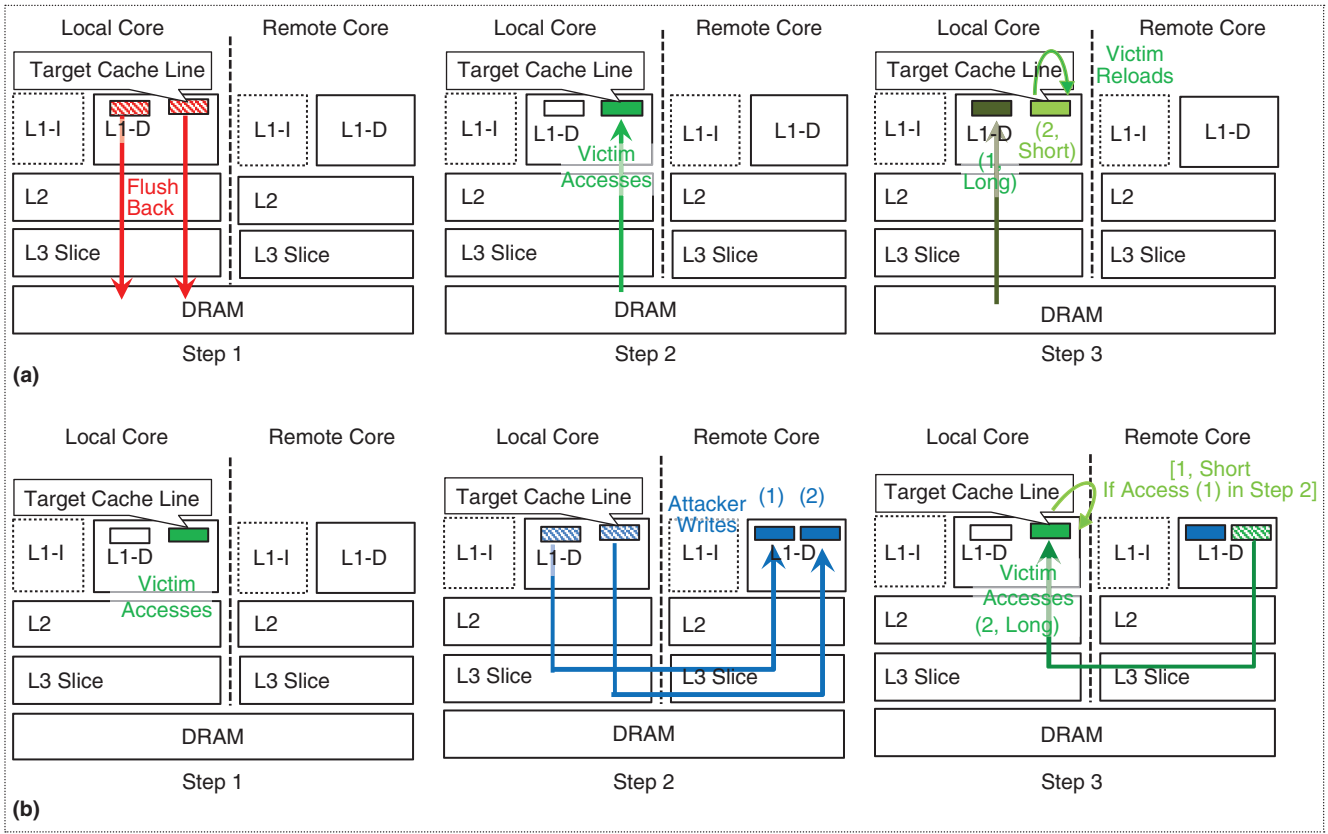

Figure 2. Some examples of vulnerabilities of different types, including: (a) the vulnerability using techniques of the previously proposed Cache Collision[7] attack and (b) a new vulnerability, falling into the "Flush and Time" type, derived from our model. DRAM: dynamic random-access memory.

outputs whether there is a significant timing difference (vulnerability found) or not (vulnerability not found).

The Susceptibility of Real Processors

We evaluated the vulnerabilities by running the experiments on nine processor configurations and display the results in Table 1. For each type of processor, a full-solid circle showing up in the entry means that the machine is susceptible to all of the vulnerabilities of that attack type. A half-solid circle and hollow circle mean that the machine is vulnerable to partial and no vulnerabilities of that attack type, respectively. The Intel Xeon E5-2667 in our lab has two sockets. Therefore, the local and remote cores can both be in one socket (that is, they can run on-chip), or the local and remote core can be in different sockets (that is, they can run interchip). Table 1 presents that the 27 Vulnerability Types are mostly found in all of the tested CPUs. Since our new cache simulator considers the ideal case—where all possible timing observations within the cache hierarchy have unique results—it outputs all of the possible vulnerability types. For commodity processors, a different subset of vulnerability types is shown to be effective on different processors. This is likely due to the microarchitectural variations and that some timing measurements are not differentiable on various CPUs.

The Insecurity of Processor Caches

Based on the evaluation results, we found that commercial caches are generally susceptible to many cache-timing-based vulnerabilities, which demonstrates the insecurity of processor caches. In this section, we summarize some lessons learned from our work.

The Impact of Architectural Designs on Security

We observed that processors from the same family have similar evaluation results, based on common architectural features implemented for the whole memory hierarchy. Different implementations will be reflected in different results for the various vulnerability types.

For example, according to Table 1, Bernstein's Invalidation Attack is usually observed on Intel E5-1620 and only sometimes observed on Intel E5-2690. Based on our analysis, flushing L1 data to DRAM and flushing L2 data to DRAM have large timing differences for Intel E5-1620 (1,036 versus 985 average cycles) but are nondifferentiable for Intel E5-2690 (872 versus 879 average cycles). With the smaller difference, it is not possible to distinguish the timing with high confidence, and corresponding attacks are not exploitable on this processor.

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

These timing variations could possibly be used to fingerprint the processors or even reverse-engineer the implementations. Based on our prior example, if a machine is evaluated and is found to be fully vulnerable to this attack, the machine may have a fast clock speed and may not support Flex Memory Access either.

Existing Processors and the Need for Secure Caches

Due to the insecurity of current commercial caches, previous work has developed hardware secure caches.[13] There are three main techniques that the secure caches utilize: partitioning, randomization, and differentiating the security-critical data. These security features can help prevent the corresponding vulnerabilities if our test suite finds the processor to be susceptible to them.

Partitioning-based caches usually limit the victim and attacker to only being able to access their own assigned set of cache blocks. For example, partitioning can be used to prevent "Flush and Reload"-related vulnerabilities—where the cache is flushed in step 1, then, the victim process accesses data u (which could possibly be a, a^{alias}, or *NIB*) in step 2, and the attacker process accesses data a in step 3. When u is equal to a, the data a will be put into the victim partition of the cache, and the attacker will not see a in its own partition in step 3. This will lead to cache-miss timing always being observed in step 3, which is the same as the case when u is equal to a^{alias} or *NIB*. Consequently, there is no observed timing difference, which helps the cache prevent this vulnerability.

Randomization-based caches inherently decorrelate the relationship between the address information of the victim's security-critical data and the observed timing from the cache hit or miss, or between the address and the observed timing of the flush or cache-coherence operations. For example, for the "Prime and Probe" vulnerabilities, the attacker accesses the same address a to prime and probe a specific cache set in step 1 and step 3 while the victim accesses u in step 2, where u possibly maps to the same cache set as a, where a slower cache-miss timing is observed by the attacker in Step 3. Otherwise, faster cache-hit timing is observed. However, if some random data (rather than the data that maps to the same cache set as a) is filled into the cache in step 2, cache-miss timing observation does not deterministically correspond to the result that a and u map to the same cache set, which limits this attack.

Differentiating the security-critical data is a mechanism that could allow the victim program or system

management to explicitly label a certain range of the victim's security-critical data. Using this, we can apply randomization or a partitioning scheme to only the labeled data for better efficiency and security. Cache-specific instructions could be used to protect the data and to limit the internal interference among a victim's own data. For instance, the Cache Collision type that has victim operations in all of the three steps could likely be prevented by combing randomization techniques and by labeling the security-critical data to randomize the victim's access to every step to decorrelate the relations between the timing observation and victim's behavior.

The Impact of Attack Discovery on Processor Caches

Previous discovery of a type of cache-based timing side-channel can influence commercial processor designs that are proposed or released later. For example, Intel's Cache Allocation Technology (CAT), available today in Intel Xeon E5 2618 L *v*3 processors,[14] can be used to realize the partitioning approach mentioned in "Existing Processors and the Need for Secure Caches" to prevent interference among different processes in the cache hierarchy. Although it is unclear as to whether CAT was implemented for performance or security, it is one example of a cache feature that can help security.

Customized Hardware and Software Defenses

Our test suite results have shown that different processors are vulnerable to different attacks. Consequently, customized software or hardware defenses could be deployed for each processor, based on the evaluation results, rather than defenses against vulnerabilities not present in the specific processor's caches. In general, however, customized defenses may be expensive in hardware, and hardware secure caches may be preferable.

For some of the software defenses, the access patterns from the test suite could be used as a reference for scanning software to find if it has similar patterns, for example, to find malicious software that has similar attack patterns. Further, evaluation results and our model have uncovered attack types which were unknown before and, thus, were not considered by existing defenses. With an understanding of the 88 types of attacks, new software defenses leveraging performance counters can be deployed, for example, scanning for the write accesses that can be used in the attacks, which were not considered before.

Securing Future Processor Caches

From our evaluation results, timing-based vulnerabilities exist in all of the tested commercial processors' caches. Therefore, we advocate for more vendors to consider this threat and to build secure cache features into commercial processors in the near future. The systematic modeling of all of the possible cache-timing vulnerabilities is necessary to understand processor security. In addition, our mostly automated approach helped find new potential vulnerabilities, further highlighting the dangers resulting from set-associative caches.

The Applicability of Our Test Suite

To always keep the system secure, defenses need to protect against all types of attacks, anticipating what the attacker can do. Hence, a systematic approach, such as the one we have presented, is needed to first find all of the vulnerabilities and, then, to use the results to build defenses. In this case, our proposed model and the test suite should be used to examine all of the possible cache-timing-based attacks for a processor and, then, give insights about the needed defenses.

As mentioned in "The Impact of Attack Discovery on Processor Caches," no actual secure cache designs have been incorporated into commercial processors. The main reasons deduced from the evaluation could be the following:

- In going from software to hardware, it becomes harder to modify the system. It requires more complicated verification work to guarantee that the whole processor will keep the original functionality when new security features are added, hence the slow adoption of secure caches.
- Many secure cache designs require 2–10% performance overhead, which is not tolerable for many commercial usage scenarios.
- Vendors usually prioritize equipping machines with performance features rather than security features.

However, since the Spectre[3] and Meltdown[4] discovery, more companies have been paying attention to the problem of side channels and are working on incorporating hardware defenses. Our work, in this case, can help motivate the establishment of these hardware defenses as we thoroughly demonstrate the scope of timing-based vulnerabilities. Furthermore, our test suite shows that there are real side-channel problems on commercial processors and gives tangible results in terms of identifying which processors may be vulnerable to which types of attacks.

Future Direction and Challenges

Apart from applying our methodology to the L1 data caches, we have also employed the modeling approach in our work on translation lookaside buffers (TLBs) to enumerate and understand all possible timing-based vulnerabilities in TLBs and to provide corresponding hardware defenses.[15] In that work, security microbenchmarks are automatically generated based on the

model. New secure TLB designs are proposed accordingly and evaluated using the test suite. Furthermore, we believe that our modeling approach and automatic generation framework of the test suite can be extended to other levels of caches and cache-like structures, such as the branch target buffer of branch prediction units, or even other microarchitectural features.

The main challenges of extending the modeling approach and test suite framework are the variants of different microarchitectures. For example, the current model only considers the data address of the cache line. If the cache coherence bits and replacement policy are included to be treated as part of cache states, possible steps of our model need to be customized, and it may require adding a fourth step to capture all of the vulnerabilities as the model is expanded. In addition, in branch prediction units, pattern history could be considered instead of the actual address of the data in the cache, so the model would have to be changed. However, we still believe that our work—providing a modeling approach and test suite framework for timing-based channels—can serve as a catalyst to help direct secure hardware design. ■

Acknowledgments

This work was supported by NSF Grant 1813797 and through Semiconductor Research Corporation award 2844.001.

References

1. D. J. Bernstein, "Cache-timing attacks on AES," Preprint, 2005. [Online]. Available: http://cr.yp.to/papers.html#cachetiming
2. M. Clémentine, N. Christoph, H. Olivier, and F. Aurélien, "C5: Cross-cores cache covert channel," in *Proc. Int. Conf. Detection Intrusions and Malware, and Vulnerability Assessment*, 2015, pp. 46–64.
3. P. Kocher et al., " Spectre attacks: Exploiting speculative execution," in *Proc. Symp. Security and Privacy*, 2019, pp. 1–19.
4. M. Lipp et al., " Meltdown: Reading kernel memory from user space," in *Proc. USENIX Security Symp.*, 2018, pp. 973–990.
5. S. Deng, W. Xiong, and J. Szefer, "Analysis of secure caches using a three-step model for timing-based attacks," *J. Hardw. Syst. Security*, vol. 3, no. 4, pp. 397–425, 2019. doi: 10.1007/s41635-019-00075-9.
6. D. Shuwen, X. Wenjie, and S. Jakub, "A benchmark suite for evaluating caches' vulnerability to timing attacks," in *Proc. Int. Conf. Arch. Support Programming Languages and Operating Syst.*, 2020, pp. 683–697.
7. B. Joseph and M. Ilya, "Cache-collision timing attacks against AES," in *Proc. Int. Workshop on Cryptographic Hardw. and Embedded Syst.*, 2006, pp. 201–215.
8. Y. Yuval and F. Katrina, "Flush+ Reload: A high resolution, low noise, L3 cache side-channel attack," in *Proc. USENIX Security Symp.*, 2014, pp. 719–732.
9. A. O. Dag, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *Proc. Cryptographers' Track at the RSA Conf.*, 2006, pp. 1–20.
10. G. Daniel, M. Clémentine, W. Klaus, and M. Stefan, "Flush+ Flush: A fast and stealthy cache attack," in *Proc. Int. Conf. Detection Intrusions and Malware, and Vulnerability Assessment*, 2016, pp. 279–299. 2016.
11. T. Caroline, L. Daniel, and M. Margaret, "MeltdownPrime and SpectrePrime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols," 2018, arXiv:1802.03802.
12. L. W. Bernard, "The generalization of student's problem when several different population variances are involved," *Biometrika*, vol. 34, no. 1/2, pp. 28–35, 1947. doi: 10.2307/2332510.
13. J. Szefer, *Principles of Secure Processor Architecture Design*. San Rafael, CA: Morgan & Claypool, 2018.
14. F. Liu et al., "CATalyst: Defeating last-level cache side-channel attacks in cloud computing," in *Proc. Int. Symp. High Performance Comput. Arch.*, 2016, pp. 406–418.
15. S. Deng, W. Xiong, and J. Szefer, "Secure TLBs," in *Proc. Int. Symp. Comput. Arch.*, 2019, pp. 346–359.

Shuwen Deng is a Ph.D. candidate in the Department of Electrical Engineering at Yale University, New Haven, Connecticut, 06520, USA, working with Prof. Jakub Szefer. Her research interests include developing and verifying secure processor microarchitectures by self-developing timing side-channel, vulnerability-checking schemes as well as languages and tools for practical and scalable security hardware and architecture verification. Deng received a B.Sc. in microelectronics from Shanghai Jiao Tong University. Contact her at shuwen.deng@yale.edu.

Wenjie Xiong is a postdoctoral researcher. Her research interests include physically unclonable functions and side-channel attacks and defenses in computer architecture. Xiong received a Ph.D. from the Department of Electrical Engineering at Yale University, New Haven, Connecticut, USA. Contact her at wenjie.xiong@aya.yale.edu.

Jakub Szefer is an associate professor in the Electrical Engineering Department at Yale University, New Haven, Connecticut, 06520, USA, where he leads the Computer Architecture and Security Laboratory. His research interests are at the intersection of computer architecture, hardware security, and field-programmable gate array security. Szefer received a Ph.D. in electrical engineering from Princeton University. Contact him at jakub.szefer@yale.edu.