Zero-Store Elimination and its Implications on the SIKE Cryptosystem

Lukas Gerlach
CISPA Helmholtz Center for Information Security

Niklas Flentje
Saarland University

Michael Schwarz
CISPA Helmholtz Center for Information Security

Abstract

Modern processors spend a significant amount of their execution cycles waiting on memory. Value-based optimizations tackle this bottleneck by optimizing for specific memory content patterns. Zero-store elimination, in particular, skips memory writes for redundant zero values, reducing memory pressure and boosting processor performance. We investigate the state of zero-store elimination in modern Intel processors and design experiments to reverse engineer its properties. We identify the conditions that trigger zero-store elimination and demonstrate how an attacker can selectively induce zero-store elimination. Similar to previous work on pointer prediction on Apple silicon, our analysis reveals that zero-store elimination has severe security implications, reaffirming Intel's decision to turn off this optimization via microcode updates. Our analysis reveals that value-based optimizations extend traditional side-channel attacker models, exposing partial information about the processed values (as opposed to just metadata).

This expanded attack surface, created by value-based optimizations, breaks constant-time programming techniques, enabling attacks such as key leakage from Supersingular Isogeny Key Encapsulation (SIKE). We design a zero-store elimination-based attack on SIKE that recovers 208 of the 217 bits of the secret key in 3.7 s. Additionally, we provide a dynamic analysis tool to detect zero-store elimination in programs and verify that it successfully detects SIKE's weakness toward zero-store elimination. We propose mitigations that allow a tradeoff between security and performance. Our findings caution against the broader implications of value-based optimizations and urge careful consideration of their security risks in future processor designs.

1 Introduction

Performance optimization remains an important goal of modern processor design. A steadily increasing gap between processor and memory speeds has been known for decades [16, 78] as processors prioritize speed, while memory prioritizes capacity. To close this performance gap, processors implement a wide range of optimizations that aim to keep memory close to the processor, primarily in various types of caches. Caches store frequently used values in quick and fast (cache)memory. Caches have been equipped with additional optimizations to make them even more performant, such as dedicated predictors that prefetch data likely to be used [6]. A special, recently implemented class of memory subsystem optimizations is value-based optimization, which takes the actual values stored in memory into account.

Zero-store elimination is an undocumented value-based optimization mechanism implemented on some Intel processors [68, 69]. This optimization leverages the fact that most workloads do not generate uniformly distributed data, with zero values being more likely [50, 72]. Therefore, trading off some implementation complexity can speed up computations involving zero values in the memory pipeline. Specifically, zero-store elimination speeds up cases where a zero-value store overwrites another zero value already in memory. The processor can track memory locations that are zero and avoid unnecessary writebacks when memory traverses the memory pipeline. This saves time when memory is evicted through the cache levels, reducing memory pressure and improving performance.

In this paper, we analyze the inner workings of zero-store elimination. We reverse-engineer the necessary conditions to trigger zero-store elimination. These conditions include the size of the stored value and the type of instruction that triggers the store. We demonstrate that zero-store elimination only triggers on zero writes of 64 bytes that are cache-line aligned. Additionally, we demonstrate that zero-store elimination can be triggered by both vector and non-vector instructions, but not by non-temporal store instructions. We also reproduce a previous analysis [68] that locates where zero-store elimination occurs in the memory pipeline and confirm that on the analyzed processors, the L3 cache is responsible for zero-store elimination. We analyze the security implications of zero-store elimination, with their severity confirmed by Intel's decision to turn off the optimization entirely via microcode updates [70]. Additionally, Intel issued a CVE [23] as well as a security advisory [1] for zero-store elimination. Moreover, we develop tools to triage which microcode version removes zero-store elimination on which processors. In the case of zero-store elimination, the Intel security advisory [1] provides a list of vulnerable microcode versions. However, this information does not list all microcode versions and their vulnerability status, which we provide.

Based on our reverse-engineering efforts, we propose an attacker model for zero-store elimination. Similar to other powerful software-based attacks based on power measurements [46, 51] or ciphertext side channels [48], value-based optimizations such as zero-store elimination can reveal direct information about the values a program processes. Therefore, zero-store elimination exposes an additional attack surface for side-channel attacks, resulting in different requirements for implementing cryptographic algorithms. Additionally, zero-store elimination bypasses common assumptions of software side-channel countermeasures and side-channel detection tools [24, 47, 75, 76].

We demonstrate that value-based optimization breaks Supersingular Isogeny Key Encapsulation (SIKE), a previously considered constant-time cryptographic implementation. While SIKE is cryptographically broken [17], it illustrates the circumstances under which zero-store elimination can lead to a successful attack. Attacks based on zero-store elimination can work with a single zero on zero write. Such zero-on-zero writes can either naturally occur in the program or be forced by an attacker. Additionally, the size of the store must be at least 64 bytes to trigger zero-store elimination reliably. The data processed by the SIKE implementation meets both these requirements. Previous work [28, 73] shows similar attacks on SIKE using (power) side channel attacks. Power side channels allow an attacker to observe energy differences induced by computations on zero versus non-zero values. We demonstrate that attacks on the SIKE implementation are possible in software and outside the power domain by utilizing zero-store elimination to compromise the isogeny evaluation of SIKE's decapsulation process. Using carefully crafted inputs, an attacker can conditionally force long runs of zeroes to occur during this isogeny evaluation [28, 73]. We can make these long runs observable via the timing difference induced by zero-store elimination The occurrence of these zero-runs, and thus the timing difference introduced by zero-store elimination, is key-dependent. Thus, an attacker can recover the SIKE secret key. We show that the timing difference induced by zero-store elimination allows leaking 208 of the 217 bits (95.85 %) of the SIKE key in 3.7 s. The remaining key bits are recovered using $2^9 = 512$ trial encryptions, which is feasible even for an attacker with limited resources.

We also provide a dynamic analysis tool to detect potential zero-store elimination in programs, using an approach similar to prior side-channel detection approaches [75, 76]. Our tool enables developers to analyze code, such as the SIKE implementation, for zero-store elimination and to detect side channels that arise from it. We benchmark our tool against both a sparse matrix multiplication library [25] and the SIKE implementation, and find that it can detect potential zero-store elimination with an average slowdown of 63.64× across these workloads. Future work could integrate our analysis approach into well-known frameworks such as DATA [75] or microwalk [76]. Based on our observations, we propose alternative mitigations to turning off zero-store elimination. While the performance impact of turning off zero-store elimination is small, our mitigations also apply to other value-based optimizations. Our mitigations include selectively turning off zero-store elimination for critical code segments, as well as spot fixes that mask critical values to prevent them from triggering zero-store elimination. Additionally, we benchmark the performance benefit of zero-store elimination using the SPEC CPU 2017 benchmark suite [21] and find that it yields a modest average speedup of only 0.27 %. This small performance benefit suggests that disabling zero-store elimination is a feasible option for security-critical applications where value-based side channels are a concern.

In summary, we analyze the implications of zero-store elimination on side-channel attacks on cryptographic implementations as an example of value-based optimization. While modern and older processors have disabled zero-store elimination via microcode updates, and SIKE as a scheme is cryptographically broken, we want to highlight the broader implications of our work. Zero-store elimination is not the only value-based prediction mechanism; others have already been implemented in current processor generations [18, 27, 41, 42]. Similarly, SIKE is not the only scheme where an attacker can force a specific internal state that leads to faster execution times; examples of such attacks have existed in the past [2, 34] and will likely occur in the future. Our results can also have a practical impact if microcode patches are not applied. It also serves as a general cautionary remark on the potential security implications of value-based prediction mechanisms.

Contributions. In this paper, we make the following key contributions:

1. We analyze zero-store elimination on 5 Intel processors.
2. We present an attack on SIKE that exploits zero-store elimination as a side channel, thereby bypassing classical constant-time assumptions.
3. We provide an automatic way to detect zero-store elimination and propose approaches to mitigate it.

Outline. We introduce the required background in Section 2. We reverse engineer the parameters of zero-store elimination on 5 Intel processors in Section 3. We perform a case study on SIKE in Section 4 and outline other potential attack targets in Section 5. We discuss defenses against zero-store elimination-based attacks in Section 6 and conclude in Section 7.

Availability. We will make our artifacts available at https://github.com/cispa/ZeroStore on acceptance of the paper.

2 Background

In this section, we provide background on side-channel attacks, microarchitecture, and the SIKE cryptosystem.

2.1 Side-Channel Attacks

Side-channel attacks are a class of attacks that exploit implicit information leaked during the operation of a system. Attackers can exploit side-channels at either a software- or hardware-based level; however, in the remainder of this paper, we focus on software-based side channels, i.e., side-channels that are exploited through software. The most common software-based side channels are timing, control-, and data-flow-based side channels. These side-channel-based attacks exploit observable differences in the secret-dependent execution behavior of a program. An example of such observable differences is variation in execution time, which can be caused either directly by conditionally executed program paths of different lengths or indirectly through side effects in the memory hierarchy, i.e., cached and uncached code paths, or through variable-time instructions. Side-channel leakage can be mitigated by ensuring that execution time and memory-access patterns are independent of secret data, as well as by preventing secret data from being processed by variable-time instructions. Constant-time programming techniques [60] achieve this by eliminating secret-dependent branches, data-dependent memory accesses, and other leaky instructions with confidential inputs. While the underlying leakage source, cache, or instruction-based timing differences still exist, constant-time programming methods stop secret-dependent leakage. Although constant-time programming itself is fragile to implement and test [32,40,63], it can mitigate most currently known side channels if applied correctly.

However, side channels caused by value-based optimization, such as zero-store elimination, are not mitigated by constant-time programming. Even if the control and data flow of a program is linearized, making the program constant time, value-based side channels can still leak information about the secret values [18, 41]. Therefore, the attacker model behind side channels with value-based optimizations is more akin to power side channels [45] or ciphertext side-channel-based attacker models [49] and requires countermeasures such as masking [56, 77] to be mitigated.

2.2 Microarchitecture

Microarchitecture refers to the internal implementation of a processor's abstract specification, known as the instruction set architecture (ISA). While the ISA specifies the behavior of the instructions executed by a processor and their interactions, the microarchitecture is a specific instantiation of the ISA. Multiple microarchitectures can implement the same ISA, and architecturally, they should behave equivalently for all architecturally defined behaviors. However, as the microarchitecture is also responsible for making the processor fast and efficient, it can introduce side effects that can be observed and exploited by an attacker. While there are instances where microarchitectural attacks become architecturally visible [10, 80], they are typically only visible via timing differences. While the lack of timing differences is not mandated by the ISA (and doing so would make many implementations infeasible), they are still exploitable by attackers. Previous attacks have used them to leak information about secret values in cryptographic implementations [8,13,30,55,82] as well as confidential informatinon contained in the memory of a system [10, 15, 43, 49, 52, 64, 79].

2.3 SIKE Cryptosystem

Supersingular Isogeny Key Encapsulation (SIKE) is a post-quantum key encapsulation mechanism built on the mathematics of elliptic curves, and *isogenies* [4]. In SIKE, each supersingular elliptic curve acts as a point in a big, structured graph. Special maps called isogenies correspond to edges between these points. SIKE's security relies on the difficulty of reversing a random walk in the isogeny graph. Even with quantum computers, an attacker cannot feasibly recover the secret path between two given points. SIKE offered small key sizes and reached the finals of the NIST Post-Quantum Cryptography competition, but researchers later broke its security, leading to its withdrawal [17]. It operates analogously to Diffie-Hellman key exchange: each party uses a secret scalar to compute an isogeny from a shared base curve to a new curve, resulting in a shared key derived from the resulting elliptic curve's j-invariant. A detailed description can be found in Costello et al. [22], and a broader survey of isogeny-based cryptography is provided by De Feo et al. [26]. In the protocol, Alice chooses a secret integer k_A and uses it to define an isogeny ϕ_A that maps the base curve E to a new curve E_A. Her public key consists of $(E_A, \phi_A(P_B), \phi_A(Q_B))$, where (P_B, Q_B) are fixed public basis points used by Bob. Bob performs a symmetric operation to generate his key. After the exchange of public keys, each party recomputes a new isogeny using their secret and the other party's public data. For example, Alice forms a secret subgroup by computing $S'_A = \phi_B(P_A) + [k_A]\phi_B(Q_A)$ and derives an isogeny from this kernel, resulting in a final curve E_{AB}. Bob performs the analogous computation to obtain E_{BA}, and both derive the shared secret from the j-invariant of their respective curves, which are isomorphic. Crucially, the process involves mixing the party's secret with public inputs controlled by the other party: the scalar multiplication step that computes S'_A (and likewise S'_B) interleaves secret data

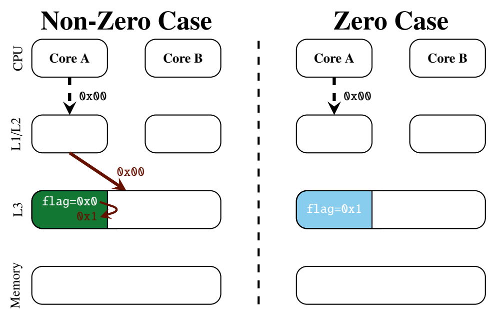

Figure 1: A hypothesis of how zero-store elimination works and why it introduces timing differences. Both cases depict a store of zero value; however, the cache state differs in each case. In the first case, the cache tracks that a non-zero value has been stored in it beforehand, as indicated by the value of the `flag` variable. Therefore the processor has to overwrite the value stored in the L3 in the process updating the `flag` variable. In case the processor already tacks a zero value in the cache, the cache line can be dropped without a writeback to the L3, which is faster. The presence of such a silent discard is hinted at by the `l2_lines_out.silent` performance counter event.

with potentially adversarial input. This structural property makes SIKE particularly sensitive to side-channel leakage. If an attacker can influence the public key and observe timing or power behavior during isogeny or scalar computations, they may be able to extract information about the secret scalar. Previous side-channel attacks exploited this dependency by providing specially crafted adversarial inputs [28, 73].

3 Demystifying Zero-Store Elimination

In this section, we analyze the inner workings of zero-store elimination on 5 Intel processors. Firstly, we hypothesize how zero-store elimination works in Section 3.1. We analyze the parameters of zero-store elimination in Section 3.2. We analyze the timing difference introduced by zero-store elimination, the location in the memory hierarchy where zero-store elimination is triggered, and the minimum store size required to trigger zero-store elimination. Additionally, we triage which microcode versions are affected by zero-store elimination in Section 3.3. We show that, given the correct microcode version, zero-store elimination is present on 12th generation Alder Lake (i9-12900K), 11th generation Tiger Lake (i7-1185G7), and 10th generation Ice Lake (i3-1005G1) Intel processors. We find that while the Intel security advisory on zero-store elimination [1] lists the i3-1005G1 and i7-1185G7 processors as vulnerable, it does not list the i9-12900K processor. Finally, we devise 2 generic ways to exploit zero-store elimination in Section 3.4.

3.1 Zero-store Elimination a Hypothesis

We hypothesize that zero-store elimination works as illustrated in Figure 1. Processors that implement zero-store elimination must check for zero values somewhere in the memory hierarchy. Our experiments show that this check is performed upon data entering the L3 cache, if a zero value is already present in the cache line that is about to be written. We assume that tracking zero values is achieved through an additional flag, in our case per cache line, that indicates whether the cache line contains zero values. In case zero on zero writes occur they can be silently discarded as they have no impact on the cache state. This is also hinted at by the `l2_lines_out.silent` performance counter event. As documented by Intel, this event documents the number of cache lines that are silently dropped by the L2 cache, i.e., , due to zero store elimination.

3.2 Microbenchmarks on Zero-Store Elimination

We perform a series of experiments on 5 Intel processors (i3-1005G1, i9-12900K, i7-1185G7, i9-13900K, i7-11700) to determine the parameters of zero-store elimination. Among these parameters are the size of the timing difference introduced by zero-store elimination, the location in the memory hierarchy where zero-store elimination is triggered, the minimum and maximum store size, and the alignment required to trigger zero-store elimination. To ensure that we measure the effect of zero-store elimination as precisely as possible, we disable prefetching, pin our process to a single core, set the processor frequency, and disable Intel Turbo Boost. For processors that contain efficiency and performance cores, we test both core types. In our tests, we observe that on the i7-1185G7 processor, which has both performance and efficiency cores, zero-store elimination is only present on the performance cores. This behavior is likely due to the microarchitecture differences between performance and efficiency cores.

Location of Zero-Store Elimination in the Memory Hierarchy. We reproduce an experiment from previous work [68] to determine which part of the memory hierarchy performs zero-store elimination. The memory hierarchy of modern Intel processors comprises multiple levels of data and instruction caches with ascending sizes and decreasing speeds. All processors we test contain three levels of caches before the main memory, namely L1, L2, and L3 (also referred to as LLC) caches. We list the size of these caches in Table 1.

We perform stores of increasing sizes and flush the target memory range of the stores using the `clflush` instruction. We then measure the time it takes for the flush operation to complete for any given size and calculate the memory throughput of flushing the cache. Our experiment forces values through all successive cache levels, allowing us to observe potential timing differences between storing zero and non-zero values. If we see that the memory throughput for flushing the cache between storing zero and non-zero values changes at a specific memory size, we can infer that stores are eliminated upon entering the L3 cache level fitting our hypothesis illustrated in Figure 1. We illustrate the results of this experiment in Figure 3, where the experiment was performed on the i3-1005G1 processor. Zero-store elimination becomes active when our memory store operations become bigger than the L2 cache size, forcing memory into the L3 cache. We see this in the timing difference between zero and non zero stores starts to occur at 2^{19} B (=512 kB) in Figure 3 which is at the limit of the L2 cache size of the i3-1005G1 processor listed in Table 1. Thus, in the L1 and L2 cache, no zero-store elimination occurs. Additionally, we track the `l2_lines_out.silent` performance counter, which tracks the number of cache lines that are discarded silently from the L2 cache. We observe that this counter aligns with the timing difference induced by zero store elimination. As we want to build attacks based on zero store elimination and unprivileged attackers generally cannot read arbitrary performance counters [37], we focus on timing differences. We obtain comparable results on all tested processors listed in Table 1. Here, the timing difference becomes observable at a different memory size; however, the memory size where zero-store elimination becomes visible always closely matches the cache size of the L3 cache. We therefore conclude that zero-store elimination must happen on the transition between the L2 and L3 cache. For even larger stores, the main memory becomes the bottleneck for store throughput, which is observable by the timing difference between zero and non-zero stores becoming smaller at 2^{22} bytes. These results align with previous work on zero-store elimination [68].

Observable Timing Difference. The timing difference introduced by zero-store elimination is important from a performance perspective, as it determines the speedup gained by zero-store elimination. The timing difference is also important from an attacker's perspective, as higher timing differences lead to measurements that are distinguished more easily and, therefore, to attacks that require fewer measurements. Additionally, high timing differences enable attacks using low-resolution timers, such as those present in browsers [57, 65]. In the following, we experimentally evaluate the timing differences introduced by zero-store elimination. For this, we use the following setup, also illustrated in Listing 1. We first initialize a target cache line by overwriting it with zero values and making sure that it is marked as unmodified using the `clflush` instruction. Next, we perform a trigger store that overwrites our zeroed target cache line. Finally, we measure the time it takes to perform a second `clflush` operation on the target cache line. Upon performing this flush operation, the processor must normally perform a writeback to memory, as we have just performed a store on the target cache line. We perform this experiment with two different memory contents for the trigger store, overwriting the zero-valued target cache line with zero and non-zero values. In the case

```
memset(target,0,STORE_SIZE); // Set cacheline to zero
flush(target); // Prime cache
if(set_nonzero){ // Selectively set to one
  memset(target,0xff,STORE_SIZE);
}else{
  memset(target,0x00,STORE_SIZE);
}
start = rdtsc();
flush(target) // Trigger
end = rdtsc();
delta = end-start
```

Listing 1: The setup used in our tests for zero-store elimination. We first zero a cache line and flush it. Then we selectively set the cache line to zero or non-zero value and flush it again. The timing of the flush instruction reveals if zero-store elimination occured.

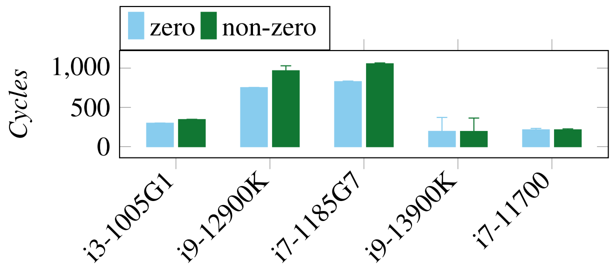

Figure 2: Average execution time of flushing a zero value to a zero and a non-zero memory region for all tested processors.

of the zero-on-zero write, zero-store elimination is triggered, eliminating the writeback to memory when we measure the execution time of the second `clflush` operation. The timing difference introduced by zero-store elimination is the difference between the execution times measured in the last step of the two experiments. We list the results of our measurements in Figure 2. We observe timing differences from 38 (i3-1005G1) to 228 (i7-1185G7) cycles. Such high timing differences are, theoretically, even measurable with coarse-grained timers in constrained environments [65]. However, we leave experiments that show concrete examples of zero-store elimination-based attacks in constrained environments to future work. We do not observe zero-store elimination on the i9-13900K and i7-11700 processors. Concluding our experiments, we find that a simple threshold-based distinguisher can detect zero-store elimination on the i3-1005G1, i9-12900K, and i7-1185G7 processors with F1 scores of 0.998, 0.999, and 0.999, respectively.

Minimum Store Size. We further analyze how the size of a store operation influences zero-store elimination. In all previous experiments, we successfully triggered zero-store elimination by storing at least 64 consecutive 64-byte-aligned zero bytes. We design an experiment to determine if there are any smaller stores that can trigger zero-store elimination. The experiment first sets a cache line to an all-one value (i.e., all

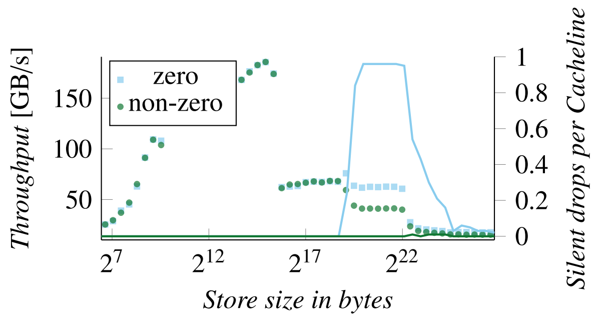

Figure 3: Average execution time (scatterplot) of flushing a zero and a non-zero value across different store sizes measured on the i3-1005G1 processor. Timing differences only become visible at $\approx 2^{19}$ B when stores of these enter the L3 cache. The `l2_lines_out.silent` performance counter (lineplot) alligns with the timing difference induced by zero-store elimination.

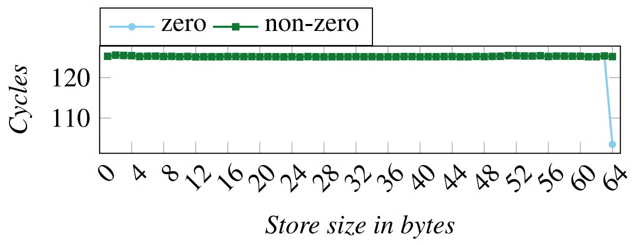

Figure 4: Time to flush a zero and a non-zero memory region for store sizes from 1 to 64 bytes on i3-1005G1. Zero-store elimination becomes active at 64 bytes (cache-line granularity).

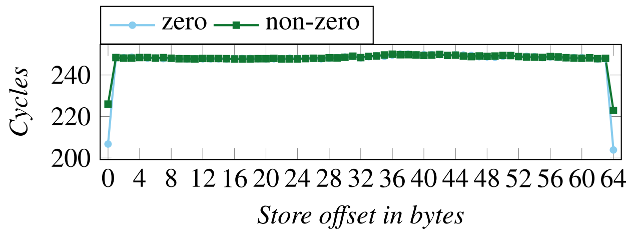

Figure 5: Time to flush a 64 byte zero and a non-zero memory region with 1 byte disalignments on i3-1005G1. Only cache-line-aligned stores are eliminated.

bytes `0xff`), which avoids triggering zero-store elimination. Then our experiment uses bytewise writes to set between 1 and 64 bytes of the cache line to zero. A first flush primes the zero store mechanism of the cache. We then bring the 64-byte line into the cache without modifying its value by accessing it and rewriting it with the same value it had before. Lastly, we measure the timing of a `clflush` instruction to the cache line to determine if zero-store elimination occurred. We repeat our experiment 10 000 000 times on the i3-1005G1 processor. Our results for the experiment are illustrated in Figure 4. The memory region size at which we observe a timing difference is 64 bytes of consecutive zeros, which is the size of a single cache line. This is consistent on all tested systems where zero-store elimination is active. These observations support our hypothesis, as illustrated in Figure 1, where one bit per cache line is required to store whether the line is zero. This is in contrast to previous work where it was hypothesized that zero-store elimination can be triggered by smaller stores [10], we were not able to observe this behavior with our testing setup on any of the tested machines.

Type of Store instruction. We additionally analyze if the type of store instruction, i.e., the size of the store and whether it is a non-temporal store, influences zero-store elimination.

Table 1: Cache size and µarch of all tested cores.

Core	µarch	Cache Size		
		L1	L2	L3
i3-1005G1	Ice Lake	48 kB	512 kB	4 MB
i7-11700	Rocket Lake	48 kB	512 kB	16 MB
i7-1185G7	Alder Lake	48 kB	1280 kB	12 MB
i9-12900K	Tiger Lake	48 kB	1280 kB	30 MB
i9-13900K	Raptor Lake	48 kB	2048 kB	36 MB

For this experiment, we store one 4 kB memory page filled with zero values and one page filled with non-zero values to a previously zeroed memory region. We perform this 4 kB store using instructions that store different sizes, i.e., 8, 16, 32, 64, 128, 256, and 512 bytes. For the 64, 128, 256, and 512 Byte store sizes, we also test non-temporal store operations. Our testing setup is similar to the one used in Listing 1. Our experimental results show that store operations are eliminated, independent of operand width, as long as the memory region to which they store is large enough. An exception is non-temporal stores, which are never eliminated. We hypothesize that non-temporal stores bypass the cache, which is responsible for zero-store elimination.

Store Alignment. Lastly, we aim to determine whether zero-store elimination occurs for 64-byte memory that crosses cache line boundaries. For this, we misalign a 64-byte byte-wise store operation by offsetting it in one cache line. As the store is 64-byte offsetting, it forces it to cross the cache line boundary because a single cache line is only 64 bytes big. The remaining measurement setup is identical to the previous experiments. We illustrate the results of this experiment in Figure 5. The plot illustrates that only cache-line-aligned stores (offsets of 0 and 64 bytes) are eliminated by zero-store elimination. This observation supports our hypothesis from Figure 1 that stores are tracked per cache line.

3.3 Affected Microcode Releases

Finally, we test which microcode versions are affected by zero-store elimination. Microcode allows the update of functionalities of an already shipped processor and can sometimes be used to patch critical security vulnerabilities in a processor [9]. Intel provides a repository that collects all microcode releases.[1] Using these publicly available microcode releases, we develop a microcode test triager. Our microcode triager is a small program that installs a cron job to reboot the system. On each reboot, the triager installs a different microcode version and reboots the system again. Experimental results are collected locally in files on the system. For each microcode release, the tool executes a callback function. We use this callback to run our zero-store-elimination detection tool. We publish our triager as open source software.

We illustrate the results of this experiment in Figure 6 While we observe that microcode patches are disabling zero-store elimination on all processors that support it, we also notice that this occurred at different timestamps. The fact that Intel turned off zero-store elimination is in line with documentation from other sources [70]. On the i9-12900K, we see that the default microcode version contained in the BIOS differs from all publicly available microcode versions and is additionally affected by zero-store elimination. Although Intel launched the i9-12900K processor in the 4th quarter of 2021[2] (i.e., after zero-store elimination was patched on the i3-1005G1 and i7-1185G7 processors), they only turned off zero-store elimination in release 220510, leaving an unexpected wide vulnerability window. Lastly, even if microcode updates are available, this does not mean that they are applied. As the operating system is responsible for applying such microcode updates, systems could stay vulnerable for extended periods.

3.4 Exploitability of Zero-Store Elimination

In this section, we propose two generic methods for exploiting zero-store elimination. A passive attacker relies on the victim to trigger zero-store elimination, whereas an active attacker attempts to trigger it selectively.

Passive Attacker. A passive attacker can observe the timing difference introduced by zero-store elimination. The victim program induces the timing difference when it self-evicts a previously zeroed memory region. This also requires the victim to self-overwrite this memory region with zero values, before the memory is evicted. This attacker model is very similar to a remote timing attacker [8, 12, 14, 44], where the attacker uses the victim program's memory accesses to self-evict values from the cache. Therefore, this model has the benefit of being easily transferable to remote scenarios. Furthermore, this attacker model is similar to previous work on frequency scaling-based power side channels, such as Hertzbleed [73, 74], which were also utilized in remote settings.

Active Attacker. In cases where the victim program does not self-evict the target cache line, a passive attacker cannot observe a timing difference due to zero-store elimination. Here, an active attacker is required to make the timing difference of zero-store elimination visible by forcing a writeback to memory. An active attacker first waits for the victim to overwrite zero values with zero values, setting up the requirements for zero-store elimination. Afterwards, the active attacker evicts the cache line and observes the timing difference induced by zero-store elimination. Such an active attacker model is more powerful as it does not require the victim to self-evict to trigger zero-store elimination. However, it requires the attacker to have some control over the victim program's memory. Namely, the attacker requires some knowledge about the victim's memory layout to use Prime+Probe [54] based techniques to force writeback from the cache. For this, the attacker requires code execution or adequate gadgets on the victim system to force cache evictions; therefore, such active attacks do not easily transfer to remote scenarios.

4 An Attack on SIKE with Zero-Store Elimination

In this section, we demonstrate that the value-based leakage introduced by zero-store elimination can be exploited in a practical scenario. As an attack target, we choose the reference implementation of the Supersingular Isogeny Key Encapsulation (SIKE) [4]. While SIKE is mathematically broken [17], its code has been widely audited against side-channel attacks, and it serves as a great example for the attack primitive that zero-store elimination introduces. We first discuss the threat model and setup of our attack, then illustrate how to trigger and observe zero-store elimination in the SIKE implementation, and lastly discuss recovering the SIKE key from the side-channel observations.

Threat Model and Setup. We assume a side-channel attacker model where the attacker and the victim run on the same machine, but the attacker does not have physical access to the machine. While a minimal attacker could use eviction-based strategies similar to Prime+Probe [54] to trigger and observe zero-store elimination, we use the `clflush` instruction instead. Since targeted flushing requires the memory to be shared between the victim and the attacker, we modify the SIKE implementation slightly so that this is the case for our target address. We never read from or write to that shared memory and leak the SIKE encapsulation key only via side-channel information. We attack a static key version of SIKE with 217-bit secret keys. Therefore, the attacker gains the ability to observe multiple decapsulation processes under the same key. Similar to previous papers, we attack a

[1]https://github.com/intel/Intel-Linux-Processor-Microcode-Data-Files

[2]https://www.intel.com/content/www/us/en/products/sku/134599/intel-core-i912900k-processor-30m-cache-up-to-5-20-ghz/specifications.html

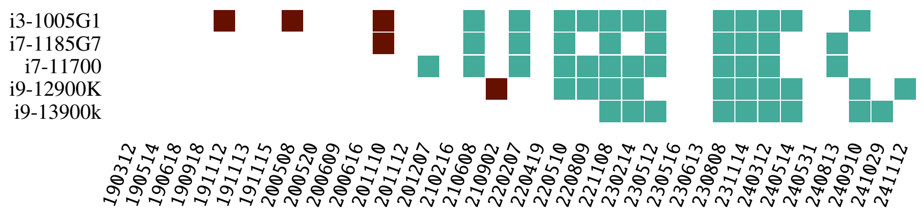

Figure 6: Heatmap of microcode releases affected by zero-store elimination on i3-1005G1, i7-1185G7, i7-11700, i9-12900K, and i9-13900K. Cells illustrate that a microcode version is affected or not-affected by zero-store elimination. Empty cells do not support the microcode release for the processor.

side-channel-hardened version of SIKE [5]. We perform our attack on two different Intel processors, an i3-1005G1 and an i9-12900K, showing that the attack is not specific to a single microarchitecture.

Triggering Zero-store Elimination in SIKE We attack the isogeny evaluation during SIKE's decapsulation process. While other attack points, such as curve point multiplication, exist in SIKE, previous work has shown that the isogeny evaluation step is the most vulnerable to side-channel attacks [28]. In the following, we define the target's secret key as $sk = sk_{216}, \dots, sk_0$, where each sk_x is a single bit of the key. Using the method described in previous works [28], we can force zero values into the isogeny evaluation if the value of a single key bit sk_x equals zero. This is possible, for all sk_x with $0 \leq x \leq 208$, if we know all previous bits sk_y with $y < x$.

Specifically, our attack targets the intermediate variable `f2elem_t R->X`, which is a 112 B large memory region. Using methods described in previous work [28], we can force zero on zero writes to occur on this variable, dependent on a single key bit sk_x. We refer to such a write as the critical write and illustrate it in Listing 1. This critical write triggers zero-store elimination, resulting in a timing difference when the variable is written back to memory, as detailed in Section 3.2. Therefore, we can force zero-store elimination to occur on `f2elem_t R->X` during the isogeny evaluation, in direct correlation with the value of a single key bit sk_x.

To ensure that zero-store elimination is triggered, our attack must ensure that the cache lines corresponding to the variable `f2elem_t R->X` remain unmodified before the critical write occurs. In our experiments, we ensure this by inserting a `clflush` instruction into the SIKE implementation as illustrated in Listing 2. However, an attacker could also use an eviction-based strategy to ensure that this condition holds, with similar reliability, but at the cost of additional implementation complexity [11, 61, 71]. We use the `clflush` instruction to simplify our experiment. Even though we use the `clflush` instruction, classical cache attacks based on it, such as Flush+Reload, would not yield the same information as our zero-store elimination-based attack. In our case, SIKE always accesses the same memory region, and we can only distinguish a zero from a non-zero value, which is not possible with a classical cache attack that targets differences in access patterns.

Recovering the SIKE Key For the concrete attack, we leak each bit sk_x separately, starting at the least-significant bit. Therefore, for each bit sk_x we know all previous bits sk_y with $0 \leq y < x$, as required by the method we use to force key-dependent zeros. Using said method, we generate a malicious public key that forces the critical write to occur on `f2elem_t R->X`, if sk_x is zero. Next, we execute the decapsulation process with our crafted input, which invokes the isogeny evaluation as shown in Listing 2. To observe whether zero-store elimination was triggered on `f2elem_t R->X`, we measure the execution of `clflush(R->X)`, after the decapsulation computation finishes. If zero-store elimination was triggered, the flush executes faster than normal since no write-back to memory needs to be performed. Therefore, we conclude that sk_x equals 0 if the execution time of `clflush(R->X)` is below a certain threshold. Otherwise, we conclude that sk_x equals 1. By repeating this procedure, we can iteratively leak all secret key bits up to sk_{208}.

Due to the difficulty of determining a perfect threshold, we may misclassify a few 0 bits as 1. To counter such errors, we use the fact that we can only force zero values depending on sk_x, if all sk_y with $0 \leq y < x$ are known [28]. Therefore, we cannot force any more zero values after misclassifying a key bit. Consequently, we classify all bits as 1, after having misclassified a single bit. Statistically, it is unlikely to have 20 consecutive 1 bits in the secret key. Therefore, we reset our attack code once we observe such a sequence in our classification.

Results. We can reliably leak the 208 least significant bits of the 217 bit SIKE key. The remaining 9 bits need to be brute forced, which requires 512 tries and is thus clearly feasible, as the position of the unknown bits is fixed and known. On the i3-1005G1, we can reliably leak the 208 least significant bits with a little more than 5 measurements per bit. On the i9-12900K, we can reliably leak the 208 least significant bits within an average of 3.7634 seconds ($n = 10^5$, $\sigma = 0.0665$), requiring a little more than 1 measurement per leaked bit. Our attack is successful in 99.756 % of the cases ($n = 10^5$).

```
f2elm_t R->X;
for (r=1; r<max_r; r++) {
  fp2copy(pts[npts-1]->X, R->X);

  ...
}
```

Listing 1: Snippet from the SIKE isogeny evaluation attacked in our case study.

```
f2elm_t R->X;
for (r=1; r<max_r; r++) {
  clflush(R->X)
  fp2copy(pts[npts-1]->X, R->X);
  ...
}
```

Listing 2: Snippet with added `clflush` instruction to ensure that zero-store elimination is triggered.

Algorithm 1: Left-to-right binary multiplication on elliptic curves using side-channel resistant double-and-add-always, as decribed by Coron [20]

```
Data: $P$ an elliptic curve point, and a positive integer
      in binary notation $e = (e_t, e_{t-1}, \dots, e_1, e_0)_2$
Result: $[e]P$
$Q_0 \leftarrow P$;
for $i = t \rightarrow 0$ do
    $Q_0 \leftarrow [2]Q_0$;
    $Q_1 \leftarrow Q_0 + P$;
    $Q_0 \leftarrow Q_{e_i}$;
end
return $Q_0$;
```

We illustrate an example leakage trace on the i9-12900K in Figure 7.

5 Other Potential Attacks using Zero-Store Elimination

In this section, we describe two other potential attacks possible with zero-store elimination. First, related to SIKE and zero-point attacks on elliptic curves, we discuss how zero-store elimination could be utilized to perform zero-point attacks on elliptic curves from a software scenario. Second, we devise an attack on RSA decryption, showing that a constant-time square-and-always-multiply implementation can be attacked using zero-store elimination.

Zero Point Attacks on Elliptic Curves. In elliptic-curve cryptography, one of the central operations is scalar multiplication of a curve point denoted $[n]P$, where P is a curve point and n is a scalar value. The scalar value n serves as the secret key in cryptosystems such as ECDSA. The scalar multiplication of curve points is often implemented similarly to the classical square-and-multiply implementation for binary exponentiation. The pseudocode for such an implementation is outlined in Algorithm 1. So-called zero-point attacks first discussed by Akishita et al. [2] exploit the fact that for an attacker-chosen index i the attacker can provide a ciphertext that leads to a zero curve point when bit n_i is processed. Additionally, recent work analyzed a wide range of elliptic curves and potential implementations for their susceptibility to zero-point attacks [67]. The fact that a fully zero curve point is processed can be observed when the curve points are sufficiently large. This is, e.g., , the case if curves such as `sect571k1` with bit sizes of 571 are used. We leave an implementation of the attack open for future work.

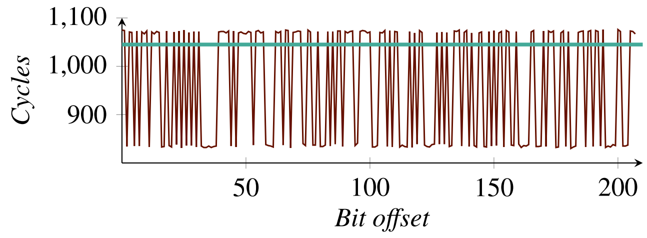

Figure 7: A zero-store elimination SIKE leakage trace on an i9-12900K. A threshold value of 1045 recovers 208 bits of the SIKE encapsulation key correctly.

Store Elimination Attacks on RSA. Similarly, the key operation in the RSA cryptosystem is modular exponentiation. During the decryption process, this occurs in the form $c^d \bmod N$, where c is an attacker-controlled ciphertext, d is the secret decryption key, and N is the publicly known modulus. Our proposed attack works on a decryption oracle, i.e., a program where the attacker can decrypt arbitrary ciphertexts under a static key. We target a side-channel hardened multiply-always exponentiation algorithm outlined in Algorithm 2. While this algorithm does not expose any control- or data-flow leakage, an attacker can observe the number of leading zeroes in the intermediate variable R_0 in every loop operation due to zero-store elimination. We also assume that R_0 is misaligned, such that m bits of R_0 are in the same cache line, and the rest of the cache line is always zero. An attacker can then determine if the leading bits of R_0 are zero by observing the timing difference of the store operation. As only m bits of R_0 are contained in the same cache line, the attacker can brute force a ciphertext where zero-store elimination is triggered in 2^m tries. The effectiveness of our attack is determined by how well an attacker can misalign the intermediate value R_0. Future work could explore whether lifting the misalignment assumption is possible. It is conceivable that the mathematical structure of the decryption operation allows for forcing small intermediate values in arbitrary positions given control over the ciphertext c.

6 Defending Against Zero-Store Elimination Attacks

In this section, we show how to defend against zero-store elimination attacks. First, we provide a tool based on Intel Pin [38] to detect where zero-store elimination can happen in a given binary. Second, we show how to perform spot fixes in critical code sections to avoid zero-store elimination, and how a system-wide mitigation could selectively disable zero-store elimination.

6.1 Detecting Code Vulnerable to Zero-Store Elimination

Intel Pin [38] is a dynamic binary instrumentation framework that allows writing instrumentation passes for binary code. We provide a Pin-tool plugin that intercepts store operations in a binary and detects potential zero-store elimination. Our pintool is similar to tools that check if binaries are constant-time before running them [75,76]. The tool works by instrumenting all store operations in a binary and checking if the value to be stored is zero and the store location is also zero. Additionally, we check if the stored block is cache-line aligned.

We evaluate our Pin plugin on the CSparse demo programs of the SuiteSparse [25] sparse matrix operation suite. As sparse matrix operations inherently process a large number of zero values, they are an ideal benchmark target for our dynamic binary instrumentation approach. We execute the benchmark on a single core with otherwise default parameters on an Intel Xeon Gold 6346 processor. On average, the benchmark runs for 62.88 s with, and for 2.01 s without our Pin plugin. Therefore, our Pin plugin induces a 31.28× slowdown. We observe that zero-store elimination is triggered during the benchmark in 2631 cases.

We also evaluate the plugin on the SIKE implementation, once while executing with random values and once while forcing zero values, as presented in our attack in Section 4. We detect zero-store elimination in 360 cases when forcing zero values and in 274 cases when not ̇The residual 274 times zero-store elimination occurs independently of any key-related information and can be explained by the basic operation of SIKE. When executed without our plugin, the attack on the SIKE implementation terminates in 0.02 s. With our pintool it executes in 1.92 s; therefore, our pintool induces a 96× slowdown. Therefore, it can be used similarly to previous Pin-based analysis passes to check binaries for constant-time implementation [75, 76]. We leave an extensive analysis of cryptographic libraries to future work, as such an analysis also requires complex input generation that triggers zero intermediate states in the tested algorithms. As indicated by previous work [31], dynamic approaches do not provide formal guarantees. However, given that the internal workings of zero-store elimination are not precisely known and can change with every processor generation or even microcode updates, obtaining formal guarantees would likely require a coarse overapproximation to be useful.

Algorithm 2: Left-to-right Multiply Always Exponentiation as described by Clavier et al. [19]

```
Data: $g, n \in G$ where $G$ is a multiplicative group, and a
      positive integer in binary notation
      $e = (e_t, e_{t-1}, \dots, e_1, e_0)_2$
Result: $g^n \mod n$
$R_0 \leftarrow 1$; $R_1 \leftarrow m$; $i \leftarrow k-1$; $t \leftarrow 0$;
while $i \geq 0$ do
    $R_0 \leftarrow R_0 \cdot R_t \mod n$;
    $t \leftarrow t \oplus e_i$;
    $i \leftarrow i - 1 + t$;
end
return $R_0$;
```

6.2 Mitigating Zero-Store Elimination Based Attacks

We propose multiple ways to mitigate attacks based on zero-store elimination. While we benchmark the performance impact of turning off zero-store elimination entirely, we also propose alternative mitigation strategies for further exploration in future work.

Turning off Zero-Store Elimination. The most radical way to mitigate zero-store elimination attacks is to turn off the optimization entirely. Intel has taken this way in the past via microcode updates [70]. While this mitigation is effective, it also comes with a performance penalty. The performance penalty of turning off zero-store elimination on the SPEC CPU 2017 Intspeed benchmark [21] is illustrated in Figure 8. The performance penalty is comparatively small with 0.27 % on average. However, for some applications that process a large amount of zero data, performance penalties are likely to be higher. The lowest overhead is measured for the xz file compression benchmark with 0.16 %, while the highest overhead is measured for the gcc compilation benchmark with 0.74 %. Overall, the overhead of turning off zero-store elimination is small.

Selectively Turning Off Zero-Store Elimination. Instead of completely removing zero-store elimination, another approach is to deactivate it only when critical data is processed. Such defenses have been shown to be effective against prefetching-based side channels [62]. Implementing such a defense means that non-security-critical code can still benefit from the performance benefits that zero-store elimination offers, while critical data remains protected. To realize this approach, we suggest adding a model-specific register (MSR) that turns off zero-store elimination. If a certain bit in the MSR is set, the processor does not allow zero-store elimination. Such MSRs already exist for similar value-based prediction mechanisms [3, 39]. A developer can then use two system calls to set the MSR before every critical code section and reset it afterwards.

Spot Fixes. A simple spot fix for critical code sections can be to add a dummy store operation that writes a non-zero value to the memory location. This way, the zero-store elimination optimization is not triggered, and the side channel is mitigated. This mitigation is straightforward to implement and can be applied in a targeted manner to critical code sections. Such a mitigation, in the form of a C preprocessor macro, is illustrated in Listing 2. Instead of doing a direct store operation, a macro is used to overwrite the memory location with a non-zero value. This way, the zero-store elimination optimization is not triggered, and the side channel is mitigated. Similar mitigations could be integrated into other places where secret values are stored. Alternatively, a compiler extension could mask every critical store, similar to mitigations such as Cipherfix [77]. By XORing data with a pseudorandom stream on store and load, long runs of zero values that trigger zero-store elimination are prevented, similar to memory scrambling [7].

In-cache Computations. As determined by previous research [68] and validated by our experiments in Section 6, zero-store elimination is only active after the L3 cache level. Therefore, a simple way to mitigate zero-store elimination is to keep critical data in the L1 or L2 cache, to ensure that an attacker cannot evict it. One possibility is memoryless encryption, where secrets are stored in registers instead of memory and never leave the registers [29, 58]. To prevent cases in which an attacker evicts values from the cache to trigger zero-store elimination, a partitioned cache can be utilized. This way, the attacker and victim do not share cache lines and, therefore, an attacker cannot, via `flush` instructions or cache eviction, remove cache lines of the victim from the cache. Therefore, the only risk in such a scenario is self-eviction of zero values by the victim, which could be prevented by careful program design. Partitioning the cache is possible via hardware mechanisms such as Intel CAT [36] or by using cache coloring mechanisms [35]. Intel CAT is a hardware mechanism that allows the partitioning of the L3 cache and has been proposed to mitigate cache-based attacks in previous work [53]. Previously it has been shown that Intel CAT cannot entirely prevent cache attacks, attack on the cache directory still remain possible because it is not partitioned by Intel CAT [59]. Therefore, the ability of Intel CAT to protect against zero-store elimination attacks depends on the exact point at which zero-store elimination is applied in the memory hierarchy and whether partitioning is possible at this point. Cache coloring works without dedicated hardware features by choosing physical addresses that map to specific cache sets or slices [33, 66, 81], separating them between different security domains. However, both Intel CAT and cache coloring reduce the cache size and, therefore, induce additional performance penalties.

```
#define MEMSET(b,c,len) \
  memset(b,0xff,len); \
  mfence(); \
  memset(b,c,len);
```

Listing 2: A C preprocessor macro that adds a dummy store operation to a critical code section, preventing zero-store elimination.

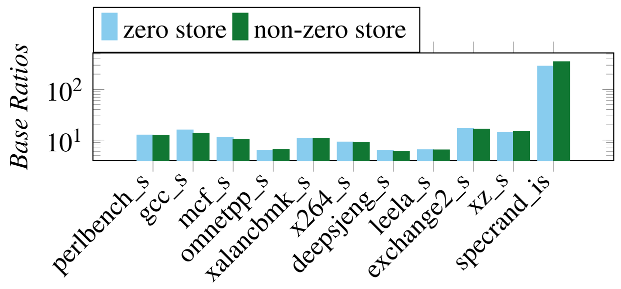

Figure 8: Performance penalty of turning off zero-store elimination on the SPEC CPU 2017 Intspeed benchmark. Disabling zero-store elimination leads to an median performance penalty of 0.8 % and average performance penalty of 0.27 %.

7 Conclusion

In this paper, we investigated the zero-store elimination mechanism via a series of microarchitectural experiments. We evaluated the store sizes at which zero-store elimination is triggered, the timing difference introduced by zero-store elimination, and the microcode versions affected by zero-store elimination. We showed that, depending on the microcode version, zero-store elimination is present on Alder Lake, Tiger Lake, and Ice Lake processors. Our analysis revealed significant security implications of zero-store elimination in a side-channel context. We showed how value-based optimizations break traditional side-channel attacker models, exposing partial value information (as opposed to metadata). We applied zero-store-based attacks to leak the keys of Supersingular Isogeny Key Encapsulation (SIKE). Using our attack, we leaked 208 bits of a 217-bit SIKE key with an average of 1 measurement per bit. We additionally analyzed elliptic curve cryptography and RSA decryption as potential attack targets. We proposed a tool based on Intel Pin to detect zero-store elimination in binaries and proposed mitigations to defend against zero-store elimination attacks. Our findings caution against the broader implications of value-based optimizations and urge careful consideration of their security risks in future processor designs.

References

[1] "INTEL-SA-00464: 2021.1 IPU - Intel Processor Advisory," 2021, accessed 2025-06-03. [Online]. Available: https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00464.html

[2] T. Akishita and T. Takagi, "Zero-value point attacks on elliptic curve cryptosystem," in *ISC*, 2003.

[3] ARM, "DIT, Data Independent Timing," 2021. [Online]. Available: https://developer.arm.com/documentation/ddi0595/2021-06/AArch64-Registers/DIT--Data-Independent-Timing

[4] R. Azarderakhsh, M. Campagna, C. Costello, L. D. Feo, B. Hess, A. Jalali, D. Jao, B. Koziel, B. LaMacchia, P. Longa *et al.*, "Supersingular isogeny key encapsulation," *Submission to the NIST Post-Quantum Standardization project*, 2017.

[5] R. Azarderakhsh, B. Koziel, M. Campagna, B. LaMacchia, C. Costello, P. Longa, L. De Feo, M. Naehrig, B. Hess, J. Renes *et al.*, "Supersingular isogeny key encapsulation (2017)," 2017. [Online]. Available: http://sike.org

[6] J.-L. Baer and T.-F. Chen, "Effective hardware-based data prefetching for high-performance processors," *IEEE Transactions on Computers*, May 1995.

[7] J. Bauer, M. Gruhn, and F. C. Freiling, "Lest we forget: Cold-boot attacks on scrambled ddr3 memory," *Digital Investigation*, 2016.

[8] D. J. Bernstein, "Cache-Timing Attacks on AES," 2005.

[9] P. Borrello, C. Easdon, M. Schwarzl, R. Czerny, and M. Schwarz, "CustomProcessingUnit: Reverse Engineering and Customization of Intel Microcode," in *WOOT*, 2023.

[10] P. Borrello, A. Kogler, M. Schwarzl, M. Lipp, D. Gruss, and M. Schwarz, "ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture," in *USENIX Security*, 2022.

[11] S. Briongos, P. Malagón, J. M. Moya, and T. Eisenbarth, "RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks," in *USENIX Security*, 2020.

[12] B. B. Brumley and N. Tuveri, "Remote timing attacks are still practical," in *ESORICS*, 2011.

[13] D. Brumley and D. Boneh, "Remote Timing Attacks Are Practical," in *USENIX Security*, 2003.

[14] ——, "Remote timing attacks are practical," *Computer Networks*, 2005.

[15] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking data on meltdown-resistant cpus," in *CCS*. ACM, 2019.

[16] C. Carvalho, "The gap between processor and memory speeds," in *ICCA*, 2002.

[17] W. Castryck and T. Decru, "An efficient key recovery attack on sidh," in *Eurocrypt*, 2023.

[18] B. Chen, Y. Wang, P. Shome, C. W. Fletcher, D. Kohlbrenner, R. Paccagnella, and D. Genkin, "Gofetch: Breaking constant-time cryptographic implementations using data memory-dependent prefetchers," in *USENIX Security*, 2024.

[19] C. Clavier, B. Feix, G. Gagnerot, M. Roussellet, and V. Verneuil, "Square always exponentiation," in *INDOCRYPT*, 2011.

[20] J.-S. Coron, "Resistance against differential power analysis for elliptic curve cryptosystems," in *CHES*, 1999.

[21] S. P. E. Corporation, "SPEC CPU 2017," 2017. [Online]. Available: https://www.spec.org/cpu2017/

[22] C. Costello, "Supersingular isogeny key exchange for beginners," in *SAC*, 2020.

[23] "CVE-2020-24512," CVE, 2020.

[24] L.-A. Daniel, S. Bardin, and T. Rezk, "Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level," in *S&P*, 2020.

[25] T. A. Davis, "Algorithm 1000: SuiteSparse: GraphBLAS: Graph algorithms in the language of sparse linear algebra," *TOMS*, 2019.

[26] L. De Feo, "Mathematics of isogeny based cryptography," *arXiv preprint arXiv:1711.04062*, 2017.

[27] S. Deng and J. Szefer, "New predictor-based attacks in processors," in *DAC*, 2021.

[28] L. D. Feo, N. E. Mrabet, A. Genêt, N. Kaluđerović, N. L. de Guertechin, S. Pontié, and Élise Tasso, "SIKE channels," Cryptology ePrint Archive, Paper 2022/054, 2022.

[29] B. Garmany and T. Müller, "Prime: private rsa infrastructure for memory-less encryption," in *ACSAC*, 2013.

[30] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware," *Journal of Cryptographic Engineering*, 2016.

[31] A. Geimer, M. Vergnolle, F. Recoules, L.-A. Daniel, S. Bardin, and C. Maurice, "A systematic evaluation of automated tools for side-channel vulnerabilities detection in cryptographic libraries," in *SIGSAC*, 2023.

[32] L. Gerlach, R. Pietsch, and M. Schwarz, "Do compilers break constant-time guarantees?" in *FC*, 2025.

[33] L. Gerlach, S. Schwarz, N. Faroß, and M. Schwarz, "Efficient and Generic Microarchitectural Hash-Function Recovery," in *S&P*, 2024.

[34] L. Goubin, "A refined power-analysis attack on elliptic curve cryptosystems," in *PCK*, 2002.

[35] J. Hofmann, C. Fournet, B. Köpf, and S. Volos, "Gaussian elimination of side-channels: Linear algebra for memory coloring," in *CCS*, 2024.

[36] Intel, "Improving real-time performance by utilizing cache allocation technology: Enhancing performance via allocation of the processor's cache," 2015. [Online]. Available: https://www.intel.com/content/www/us/en/communications/cache-allocation-technology-white-paper.html

[37] Intel, "Intel 64 and IA-32 Architectures Software Developer′s Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4," 2024.

[38] Intel Corporation, "Pin - A Dynamic Binary Instrumentation Tool," 2012. [Online]. Available: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

[39] ——, "Data Operand Independent Timing Instruction Set Architecture (ISA) Guidance," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html

[40] T. Kaufmann, H. Pelletier, S. Vaudenay, and K. Villegas, "When constant-time source yields variable-time binary: Exploiting curve25519-donna built with MSVC ," in *CANS*, 2016.

[41] J. Kim, J. Chuang, D. Genkin, and Y. Yarom, "Flop: Breaking the apple m3 cpu via false load output predictions," in *USENIX Security*, 2025.

[42] J. Kim, D. Genkin, and Y. Yarom, "Slap: Data speculation attacks via load address prediction on apple silicon," in *S&P*, 2025.

[43] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *S&P*, 2019.

[44] P. C. Kocher, "Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems," in *CRYPTO*, 1996.

[45] P. C. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *CRYPTO'99*, 1999.

[46] A. Kogler, J. Juffinger, L. Giner, L. Gerlach, M. Schwarzl, M. Schwarz, D. Gruss, and S. Mangard, "Collide+Power: Leaking Inaccessible Data with Software-based Power Side Channels," in *USENIX Security*, 2023.

[47] A. Langley, "Checking that functions are constant time with Valgrind," 2023. [Online]. Available: https://github.com/agl/ctgrind

[48] M. Li, L. Wilke, J. Wichelmann, T. Eisenbarth, R. Teodorescu, and Y. Zhang, "A systematic look at ciphertext side channels on amd sev-snp," in *S&P*, 2022.

[49] M. Li, Y. Zhang, H. Wang, K. Li, and Y. Cheng, "Cipherleaks: Breaking constant-time cryptography on amd sev via the ciphertext side channel," in *USENIX Security*, 2021.

[50] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value locality and load value prediction," *ACM SIGPLAN Notices*, 1996.

[51] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, "PLATYPUS: Software-based Power Side-Channel Attacks on x86," in *S&P*, 2020.

[52] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *USENIX Security*, 2018.

[53] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *HPCA*, 2016.

[54] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *S&P*, 2015.

[55] X. Lou, T. Zhang, J. Jiang, and Y. Zhang, "A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography," *ACM CSUR*, 2021.

[56] S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer Science & Business Media, 2008.

[57] Mozilla, "performance.now resolution," 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Performance/now

[58] T. Müller, F. C. Freiling, and A. Dewald, "TRESOR Runs Encryption Securely Outside RAM," in *USENIX Security*, 2011.

[59] Pawel Wieczorkiewicz and Rodrigo Branco and Ben Lee, "On the Effectiveness of Intel's CAT as a Side-Channel Mitigation Technology," 2024. [Online]. Available: https://langsechq.gitlab.io/spw24/papers/LangSec2024-Branco-CAT-paper.pdf

[60] T. Pornin, "Why Constant-Time Crypto?" 2022. [Online]. Available: https://www.bearssl.org/constanttime.html

[61] A. Purnal, F. Turan, and I. Verbauwhede, "Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks," in *CCS*, 2021.

[62] T. Schlüter and N. O. Tippenhauer, "Prefence: A fine-grained and scheduling-aware defense against prefetching-based attacks," 2025.

[63] M. Schneider, D. Lain, I. Puddu, N. Dutly, and S. Capkun, "Breaking bad: How compilers break constant-time˜ implementations," *arXiv preprint arXiv:2410.13489*, 2024.

[64] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-Privilege-Boundary Data Sampling," in *CCS*, 2019.

[65] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript," in *FC*, 2017.

[66] J. Shi, X. Song, H. Chen, and B. Zang, "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring," in *DSN-W*, 2011.

[67] V. Suchánek, V. Sedláček, and M. Sýs, "Decompose and conquer: ZVP attacks on GLV curves," Cryptology ePrint Archive, Paper 2025/076, 2025.

[68] "Hardware Store Elimination," Travis Downs, 2020. [Online]. Available: https://travisdowns.github.io/blog/2020/05/13/intel-zero-opt.html

[69] "Ice Lake Store Elimination," Travis Downs, 2020. [Online]. Available: https://travisdowns.github.io/blog/2020/05/18/icelake-zero-opt.html

[70] "Your CPU May Have Slowed Down on Wednesday," Travis Downs, 2021. [Online]. Available: https://travisdowns.github.io/blog/2021/06/17/rip-zero-opt.html

[71] P. Vila, B. Köpf, and J. Morales, "Theory and Practice of Finding Eviction Sets," in *S&P*, 2019.

[72] K. Wang and M. Franklin, "Highly accurate data value prediction using hybrid predictors," in *MICRO*, 1997.

[73] Y. Wang, R. Paccagnella, E. He, H. Shacham, C. W. Fletcher, and D. Kohlbrenner, "Hertzbleed: Turning power side-channel attacks into remote timing attacks on x86," in *USENIX Security*, 2022.

[74] Y. Wang, R. Paccagnella, A. Wandke, Z. Gang, G. Garrett-Grossman, C. W. Fletcher, D. Kohlbrenner, and H. Shacham, "DVFS frequently leaks secrets: Hertzbleed attacks beyond SIKE, cryptography, and CPU-only data," in *S&P*, 2023.

[75] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl, "DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries," in *USENIX Security*, 2018.

[76] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar, "MicroWalk: A Framework for Finding Side Channels in Binaries," in *ACSAC*, 2018.

[77] J. Wichelmann, A. Pätschke, L. Wilke, and T. Eisenbarth, "Cipherfix: Mitigating ciphertext side-channel attacks in software," in *USENIX Security*, 2023.

[78] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH*, 1995.

[79] R. Zhang, L. Gerlach, D. Weber, L. Hetterich, Y. Lü, A. Kogler, and M. Schwarz, "CacheWarp: Software-based Fault Injection using Selective State Reset," in *USENIX Security*, 2024.

[80] R. Zhang, T. Kim, D. Weber, and M. Schwarz, "(M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels," in *USENIX Security*, 2023.

[81] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *EuroSys*, 2009.

[82] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM Side Channels and Their Use to Extract Private Keys," in *CCS*, 2012.