Spec-o-Scope: Cache Probing at Cache Speed

Gal Horowitz
Tel-Aviv University
Tel-Aviv, Israel
galhorowitz1@mail.tau.ac.il

Eyal Ronen
Tel-Aviv University
Tel-Aviv, Israel
eyalronen@tauex.tau.ac.il

Yuval Yarom
Ruhr University Bochum
Bochum, Germany
yuval.yarom@rub.de

ABSTRACT

Over the last two decades, microarchitectural side channels have been the focus of a large body of research on the development of new attack techniques, exploiting them to attack various classes of targets and designing mitigations. One line of work focuses on increasing the speed of the attacks, achieving higher levels of temporal resolution that can allow attackers to learn finer-grained information. The most recent addition to this line of work is Prime+Scope [CCS '21], which only requires a single access to the L1 cache to confirm the absence of victim activity in a cache set. While significantly faster than prior attacks, Prime+Scope is still an order of magnitude slower than cache access. In this work, we set out to close this gap.

We draw on techniques from research into microarchitectural weird gates, software constructs that exploit transient execution to perform arbitrary computation on cache state. We design the Spec-o-Scope gate, a new weird gate that performs 10 cache probes in quick succession, which forms the basis for our eponymous attack. Our Spec-o-Scope attack achieves an order of magnitude improvement in temporal resolution compared to the previous state-of-the-art of Prime+Scope, reducing the measurement time from ≈ 70 cycles to only 5 — only one cycle more than an L1 cache access. We experimentally verify that our attack can detect timing differences in a 5 cycle resolution. Finally, using our Spec-o-Scope attack, we are able to show the first microarchitectural side-channel attack on an unmodified AES S-box-based implementation, which uses generic CPU features and does not require manipulation of the operating system's scheduler.

1 INTRODUCTION

Sharing computer hardware between multiple workloads is a paradigm deployed over a wide range of computing devices, from personal devices, such as mobile phones to cloud servers. Workload isolation, enforced by the operating system in collaboration with the underlying hardware platform, is a key enabler of the paradigm. It enables processing secret or sensitive information on a device that also executes untrusted workloads that may be malicious.

Since their introduction over 20 years ago [37, 38], cache timing attacks, which exploit the timing difference between cache hits and misses, are a prominent threat to isolation. Cache attacks can bypass many security boundaries, including between processes [31, 32], containers [45], virtual machines [16, 43, 44], browser sessions [30, 36], and trusted execution environments [4, 28]. They can be deployed remotely [21], in a cloud [14, 27], or from the browser [22, 23, 36] and leak encryption keys [26, 31, 32, 37, 43], user interface actions [12, 23, 24], and more [4, 35, 41].

Much research has focused on cache attack resolution, i.e., the frequency at which the cache can be probed. Cache timing attacks typically execute code that measures the time it takes to execute a code sequence and infer the cache state from this timing. The length of time it takes to execute the code limits the speed at which the attack can repeat. Moreover, the accuracy of some attacks deteriorates significantly when the attack is repeated too frequently [2]. Low attack resolution prevents the attacker from distinguishing events that occur within a short time interval and from determining the exact time at which victim events occur. To improve attacks, some works slow the victim down, through competition on microarchitectural resources [1, 2], operating system scheduling [13], or by exploiting operating system control [28, 39]. Finally, some attacks focus on improving the attack resolution by devising faster attacks [4, 11, 34].

To the best of our knowledge, the fastest cache probing attack is Prime+Scope [34]. The attack relies on the interaction between cache levels, where eviction from the last level cache (LLC) or the cache directory [20, 42] causes eviction from the L1 cache. By carefully arranging data in the caches, Prime+Scope ensures that a victim access to a location that fits in a monitored LLC set would result in an eviction of a specific cache line from the L1 of the attacker. Thus, probing the cache in the Prime+Scope attacks boils down to *scoping*, or repeatedly measuring the access time to a memory location that is cached in the L1 cache, achieving a reported probing rate of once per approximately 70 cycles.

While the probing speed of Prime+Scope attack is impressive, we note that it is more than an order of magnitude slower than the 4 or 5 cycles reported for accessing cached data [9]. Thus, in this paper, we ask the following question: *Can we perform a cache probing attack at a rate commensurate with cache speed?*

Our Contribution

We answer the question in the affirmative. We present the Spec-o-Scope attack, which builds on Prime+Scope but achieves a rate of one probe per five cycles for a small number of probes, and an average sustained speed of 10 cycles for longer sequences.

To achieve this rate, we use weird gates [8, 18] as a cache probing mechanism. Weird gates use a race condition between speculatively executed instructions to perform logical operations on cache state, i.e., whether specific memory locations are cached or not. Katzman et al. [18] show how using weird gates to transfer cache state between locations can decouple the cache probing from measuring the cache state. Following their approach, we split the Prime+Scope attack into two phases. The first phase uses repeated activation of a weird gate to scope the monitored location and store the results as the cache state of other memory locations. The second phase uses time measurements to observe the stored state. We experiment with multiple mechanisms to cause speculative execution in the weird gate, and demonstrate that by using the gate construction of Kaplan [17], we can reduce the probing rate at the scoping phase to once per 54 cycles.

We then turn our attention to fundamental aspects of weird gates. We observe that the operation of weird gates can be represented in terms of *instruction chains* – subsequences of the instruction stream whose last instruction has a data dependency on all preceding instructions. We propose a new type of instruction chain and a new way of composing chains, allowing us to construct a gate that computes multiple functions of its inputs in a single invocation.

Finally, we build the Spec-o-Scope gate, which instead of computing a function of different inputs, computes a function of repeated probes of the same input. Specifically, our Spec-o-Scope gate construction can perform up to 10 repeated probes of an input location. The gate produces 10 outputs that identify a probe that resulted in a cache miss, if such a cache miss happens during the gate execution. The total latency of the gate is about 100 cycles. Thus, repeated invocation of the gate achieve an average probe speed of 10 cycles. Moreover, the core of the gate is a sequence of 10 probe operations that happen within five cycles of each other.

To demonstrate the utility of our Spec-o-Scope gates, we use them against two implementations of AES. Similar to Prime+Scope, we show an efficient attack against a T-tables-based implementation, requiring ≈ 7000 traces for a full-key recovery, which can be collected in less than one second. More significantly, we present an efficient full-key recovery attack on an S-box implementation of AES, which is considered much more resilient to cache attacks [3]. Our attack is based on the one presented by Cheng et al. [7] but without their requirement for modifying the AES code by adding arguably artificial attack gadgets. We also do not require interrupting the run of the encryption code by exploiting non-trivial control of the operating system as in other previous attacks [3, 28] or the availability of the now deprecated Intel TSX [5]. Our attack requires $\approx 10\,000$ traces, which can be collected in less than 3 seconds.

In summary, in this work we make the following contributions:

- We analyze the Prime+Scope attack and identify that measuring the time limits the attack rate (Section 3).
- We show that a naive use of weird gates can improve the scope rate of Prime+Scope, albeit not by a large margin (Section 4).
- We investigate the construction of weird gates, defining abstractions to represent gate structures, and identifying new constructions (Section 5).
- Building on our new gate constructions, we design the Spec-o-Scope gate, which achieves an order of magnitude improvement over Prime+Scope (Section 6).
- We demonstrate the utility of Spec-o-Scope by attacking both T-Table and S-Box-based implementations of AES (Appendix A and Section 7). To the best of our knowledge, our attack is the first successful Prime+Probe-based attack on the S-box implementation that does not require non-trivial control of the operating system [3, 5, 28] or modification of the original code [7].
- Finally, we open-source the code for our experiments and attacks.[1]

Ethical Disclosure

We have disclosed this new attack technique to Intel. As we do not identify new leakage sources, and current published countermeasures and best practices are still effective against the attack, no restrictions on public disclosure are required.

[1]https://github.com/eyalr0/Spec-o-Scope

2 BACKGROUND AND RELATED WORK

2.1 Cache Attacks

Memory caches. To bridge the gap between slower memory and faster CPUs, modern processors employ caches — small banks of fast memory that store recently and frequently used memory lines. When the processor needs to access memory, it first checks in the cache. In case of a cache hit, data is served from the cache, reducing the access time. In the case of a cache miss, the CPU needs to wait for the data. Data brought in a cache miss is typically stored in the cache. Due to the cache's limited capacity, storing new data may necessitate evicting old data from the cache. Caches typically use a variant of the least recently used (LRU) policy for deciding which memory location to evict. To facilitate management, caches are typically set-associative. That is, the cache and the memory are partitioned into sets, such that a memory location can only be cached in its corresponding set. Moreover, to achieve a balance between size and speed, modern processors employ a hierarchy of caches, ranging from the fast but small L1 cache to the larger but slower last level cache (LLC).

Cache attacks. When the cache is shared between multiple workloads, the state of the cache depends on prior execution of all workloads. Cache attacks measure access time to memory to distinguish cache hits from misses, observing the cache state to leak information on prior execution of co-resident workloads. For example, the Prime+Probe attack [26, 31, 32] first fills a cache set with attacker data and then accesses the data, measuring access time. A slow access time indicates that the accessed data is no longer in the cache, presumably evicted due to a victim's access to a memory address in the same set.

Prime+Scope. Prime+Scope [34] is a variant of Prime+Probe that achieves a high probing rate by combining three observations. First, Prime+Scope observes that instead of checking all of the data inserted by the attacker into the cache set, the attacker only needs to monitor the eviction candidate, i.e., the memory location that will be evicted next from the cache. Second, Prime+Scope relies on the inclusive nature of the LLC, which ensures that the content of the LLC is a superset of the content of the L1. Thus, evicting a memory location from the LLC also evicts it from the L1. Finally, Prime+Scope uses the observation that cache hits in the L1 cache do not affect the replacement policy in the LLC. Exploiting these observations, Prime+Scope first performs a sequence of accesses to memory that ensures that the LLC eviction candidate of a monitored set is cached in the L1. It then repeatedly measures the access time to this eviction candidate, which we denote as scope address. A victim access to a memory location in the monitored set will result in evicting the eviction candidate from the LLC, and consequently also from the L1, allowing the attacker to detect the event. However, as long as the victim does not cause an eviction in the monitored set, the attacker only needs to measure the access time to a memory location that is cached in the L1 cache. Thus, Prime+Scope reduces the time it takes to monitor an LLC cache set from a few thousand cycles [15, 26] to about 70 cycles.

2.2 Weird Gates

To improve resource utilization, modern processors do not necessarily execute instructions in the order they are specified in the program. Instead, the processor keeps track of the data dependencies between instructions and executes instructions as soon as their inputs are ready. Because in many cases the processor cannot determine the program order before executing previous instructions, processors often speculate on the outcome of instructions. One prominent cause of speculation is the prediction of branch instruction outcomes. However, speculation is not limited to control flow, and can be the result of the assumption that instructions will not cause faults. To handle possible misprediction, the processor retires instructions in order, committing their outcome to the architectural state. This allows the processor to verify that it catches mispredictions before it commits to the outcome of instructions that were speculatively executed. While transient instructions, that execute speculatively as a result of a misprediction, do not change the architectural state of the processor, their execution can leave traces in the microarchitecture. This behavior has been exploited for mounting multiple transient-execution attacks [19, 25].

Recent work on transient execution has demonstrated that it can be used for performing computation on cache state [8, 17, 18, 40]. In a nutshell, these works treat the cache state, i.e., whether a memory location is cached or not, as a Boolean variable. They then implement logical gates, called *weird gates* [8], that operate on this cache state.

Listing 1 shows an example of a weird NOT gate, which uses misprediction of a return instruction to force transient execution [17]. The gate consists of a call to a helper function (Line 3), followed by code that eventually accesses the output address (Lines 4–5) before returning. The helper function changes its return address to skip the code that follows the call. The calculation of the return address uses the value of `*in`, which is known to be zero.

```
1  void NOT_gate(int *out, int *in) {
2    tin = *in;
3    mispredict_ret(real_return + tin);
4    tout = fixed_delay(); // returns 0
5    tout = *(out + tout);
6    lfence();
7  real_return:
8    return;
9  }
10
11 void mispredict_ret(ret) {
12   set_return_address(ret);
13   return;
14 }
```

Listing 1: Return-based NOT gate.

Figure 1 shows the operation of the gate. If `*in` is not cached, as in the right-hand side of the figure, computing the return address takes a long time, and `*out` is accessed speculatively. Conversely, if `*in` is cached (left-hand side of Figure 1), the return address is computed quickly, and speculative execution terminates before executing the access to `*out`. Thus, after executing the gate, the cache state of `*out` will be the logical inverse of the state that `*in` had before the gate.

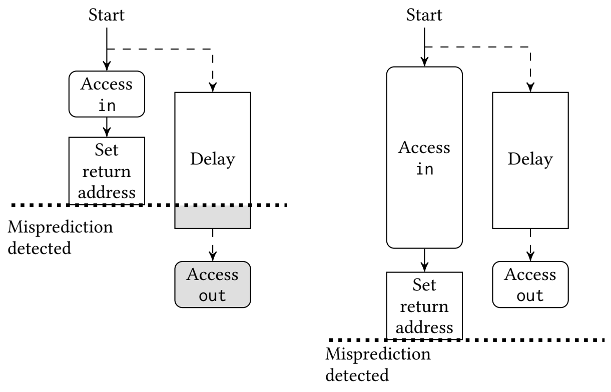

Figure 1: Operation of the NOT gate. Left when `*in` is cached, right when it is not. Shaded instructions are never executed. (Not even transiently.)

This construction of weird gates can be based on other speculation mechanisms. Previously demonstrated gates exploit branch misprediction [8, 18, 40] and return misprediction [17]. Wang et al. [40] also show how to build weird gates based on assuming that instructions do not fail. Last, Evtyushkin et al. [8] demonstrate that intentionally aborting optimistic transactions in Intel Transactional Synchronization Extension (TSX) can also be used for constructing weird gates.

2.3 Cache Attacks on AES

The AES block cipher has been extensively used to demonstrate cache attacks [13, 15, 16, 29, 31, 32] and as such have become a ubiquitous target for cache attacks.

Traditionally, a T-Tables based implementation of AES is attacked. Although specified using a 256-byte S-Box table, AES is instead commonly implemented using four 1024-byte T-Tables which significantly accelerate the encryption by combining multiple operations into a table lookup. Although slower, S-box-based implementations of AES are generally considered less vulnerable to side-channel attacks [3].

There are previous works on attacking AES S-box-based implementations, but they all require a relatively strong threat model. Some attacks are based on the ability of the adversary to interrupt the target's code execution frequently [3], the availability of special hardware features [5] (the now deprecated Intel TSX instruction set extension), or a combination of both [28] (targeting code running inside Intel SGX). The recent Evict+Spec+Time attack by Cheng et al. [7] targets a *modified* version of AES code, to which they added a non-standard secret-independent memory access attack gadget.

2.4 Threat Model

Our work follows the threat model and assumptions made by Purnal et al. [34], Kaplan [17], and Katzman et al. [18]. We assume that an adversary can run *unprivileged* code concretely with the target code on the same physical processor (not necessarily on the same core). We assume the processor has a shared leveled cache structure, and that it supports out-of-order (OoO) and speculative execution. For our attacks on AES, we assume a multi-core system with at least 3 cores, so that our two attack threads and our target code can each run on a separate core without interrupting each other (note that we don't require hyperthreading). Following [33], we also require an inclusive cache hierarchy for our Prime+Scope based LLC attack.

3 SPEC-O-SCOPE OVERVIEW

In this paper, we investigate the temporal resolution of cache attacks. Our starting point is the current state-of-the-art Prime+Scope attack [34]. We observe that it has a temporal resolution of approximately 70 cycles [34]. However, the cache-sampling step of the attack consists of a single access to a memory line that is cached in the L1 cache, whose typical latency is only 4 cycles [10]. Thus, the attack incurs an order of magnitude overhead over the core operation.

```
1 uit32_t scope(char *address) {
2   uint32_t start = rdtscp();
3   char t = *address;
4   uint32_t end = rdtscp();
5   return end - start;
6 }
```

Listing 2: Scope code from the Prime+Scope attack.

To understand the source of this overhead, we look at a typical implementation of the scope step of the Prime+Scope attack, in Listing 2. The code is pretty straightforward. To measure the access time to the scope address `address`, it queries the time stamp counter before the memory access (Line 2) and after it (Line 4). Subtracting the time stamp values yields the access time.

By timing a sequence of 10 000 calls to the RDTSCP instruction and repeating this experiment 50 000 times, we find that on average, RDTSCP takes 32.12 cycles. As each call to `scope` requires two executions of RDTSCP, we observe that measuring the execution time of the memory access makes the bulk of the overhead.

In this work, we aim to reduce the time measurement overhead. Considering the weight of the time measurement in the overhead, the intuitive approach is to replace the use of RDTSCP for time measurement. For that, we first adapt the technique of Katzman et al. [18] to decouple the sampling of the cache state from the time measurement. They use "weird" gates that operate directly on the logical state of the cache, i.e. whether a memory address is cached or not, to divide each iteration of a cache attack into two steps. In the *sampling* step, the attacker uses a NAND gate to probe the target cache set, and store the result of the probe as the logical cache state of another memory location. In the *lifting* step, the attacker measures the time it takes to access the store address, to identify whether it is cached or not. Separating the attack into two steps allows Katzman et al. [18] to overcome the limitation of a slow timer.

We can use a similar approach with the Prime+Scope attack. In each iteration of the *sampling* step, we copy the cache state from the cache line we access (which we denote the scope address) and store it in a dedicated set of cache lines (which we denote the store addresses). After the *sampling* iterations are done, we start the *lifting* step. We iterate over the store addresses and measure the access time to each address, thus learning the original cache states of the scope address in the different scope iterations. The downside of this approach is that, as in the Prime+Scope attack, once the victim has accessed the monitored set, the attacker needs to reset the state of the cache to capture further accesses. Thus, when decoupling the attack from the observation, the attacker can only learn the timing of the first victim access, but nothing about subsequent victim accesses.

We discuss the implementation of this approach in Section 4. However, while it can significantly improve the temporal resolution in the case of slow timers, in our case, a large overhead still remains. This is due to the relatively large overhead incurred by the squashing of the speculative window. To bridge the gap, in Section 6, we develop new techniques (based on the insights described in Section 5) that allow us to perform multiple measurements in a single speculative window, achieving a temporal resolution commensurate with one L1 cache hit.

4 SPECULATIVE TIME MEASUREMENT

In this section, we explore and evaluate multiple options for adapting the techniques of Katzman et al. [18] for use with the Prime+Scope attack.

4.1 Weird Gate for Speculative Measurement

Katzman et al. [18] use a weird NAND gate as part of their implementation of the Prime+Probe attack. Unlike Prime+Probe, the Prime+Scope attack, which we adapt, only accesses a single memory location, the scope address, when monitoring the cache state. Hence, we only need a gate with a single input, i.e., a NOT or a BUFFER gate, for implementing the attack.

The choice of the gate type has a significant impact on our attack. Recall that our goal is to detect with high temporal resolution the first access to the target address. Until this first access, the scope address will be cached at L1. If we use a BUFFER gate, in each scope operation, we will copy the cached state of the scope address to a new store address, i.e., we will access the store address and fetch it into memory. As we will show, our scope operation is much faster than an access to the external memory. This means that the memory fetching request will start to lag after the scope operation. However, the line fill buffer (LFB) that handles such memory requests has only a limited size queue, and when filled, it would block further memory requests. This means that asymptotically, our scope operations rate will be limited by the latency of external memory.

Based on this observation, we use a NOT gate to achieve high temporal resolution. While the scope is cached, a NOT gate does not perform any memory accesses. This allows us to run the scope operations at maximal frequency. After the target address is accessed, the scope address will be evicted, and the NOT gate will start accessing external memory. This first access will be detected with high temporal resolution. Further accesses might fill the LFB's queue and limit the rate, but at this point, we cannot learn new information, so it will not have any effect on the accuracy of the attack.

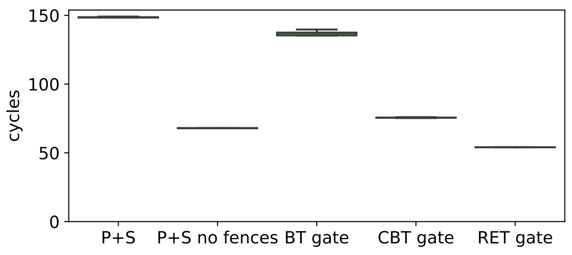

Figure 2: Distribution of execution time for a single execution of a single scope invocation for the original Prime+Scope attack and variants based on the different weird gates types.

4.2 Experimental Evaluation

Past works identified three main approaches for generic implementations of a NOT gate, with branch training, without branch training, and return-based. In particular, the no-branch-training variant of Katzman et al. [18] obviate the need for costly training by using an indirect branch with a jump table, instantiated using a `switch` statement, such that at each invocation a different case is taken. This always results in a misprediction, as the processor predicts a repeat of the previous case. We implemented all three approaches as well as two versions of the Prime+Scope attack. The first implementation of Prime+Scope uses the code from the public repository of the attack[2]. It consists of two RDTSCP instructions for measuring the access time to the scope address, with the addition of the MFENCE and LFENCE instructions for synchronizing the instruction stream. The second implementation, presented in Listing 2, is the same Prime+Scope code, but without the synchronization instructions (which can be omitted).

We measured the execution time of all five variants with a cached input (the high resolution case) on an Intel Core i5-8250U. For each variant, we ran 100 000 experiments, each executing 1260 consecutive scope invocations. As the resulting box-plot in Figure 2 demonstrates, the public version of the Prime+Scope attack is the slowest, taking on average about 150 cycles per scope invocation. The version without fences is significantly faster requiring only 70 cycles (as was reported by Purnal et al. [34]). The speculative versions of the attack also vary significantly. The branch training version is the slowest, requiring over 130 cycles per invocation. Conversely, the return-based implementation is the fastest, requiring only 54 cycles for each invocation.

4.3 Scope Overhead

The return-based scope variant's invocation time of 54 cycles is still much higher than the L1 cache typical latency of 4 cycles [10]. We will now attempt to explain the cause of this high overhead. Observing the code of the NOT gate (Listing 1), we see that the critical path through the code consists of two function calls, one memory dereference (`*in`), one addition and one memory write, which is likely forwarded to the RET instruction.

Based on Fog's optimization guides[9, 10], Table 1 summarizes the latency of these operations. The table does not include the cost of the return misprediction and overheads due to the C calling conventions and the loop that executes the gate. We could not find references for the cost of return misprediction. However, Fog [10] estimates branch misprediction costs at 15–20 cycles, and Bryant and O'Hallaron [6] estimates it at 19 cycles. Assuming 19 cycles for the misprediction, and an additional 8 cycles for the calling of the gate, we reach a total of 48 cycles, which account for the majority of the overhead of the NOT gate.

[2]https://github.com/KULeuven-COSIC/PRIME-SCOPE

Table 1: Latency of operations in return-based NOT gate.

Instruction	Count	Latency (cycles)
CALL	2	3
RET	2	2
Read from cache	1	4
ADD	1	1
LEA	1	1
Store forwarding	1	5
Total		21

While it may be possible to remove some of the overhead, it appears that a significant part of it is caused by performing the gate operation. Thus, it would appear that the overhead is essential, and cannot be removed. We therefore require a novel approach.

5 INSTRUCTION CHAINS

To further reduce the overhead, we require a novel type of weird gates. To explain our solution, we use new terminology for describing weird gates, which we will present in this section. The core concept we use is an *instruction chain*, which is a subsequence of the instruction stream that the processor executes, such that the last instruction in the chain has a data dependency on all of the instructions in the chain. For our purposes, an instruction has a direct data dependency on a prior instruction if it uses data produced by the prior instruction. Data dependency between instructions is defined as the transitive closure of direct data dependency, i.e., the data an instruction uses depends on the data produced by the other instruction. We note that in some cases, the data dependency can be implied. For example, RET instructions implicitly depend on the most recent store to the location at the top of the stack, which contains the return address. Similarly, conditional moves and conditional jumps implicitly depend on the most recent instruction that updates the flag.

We start with classifying the chains that exist in the literature based on their function in the gate and their temporal behavior. We then proceed to introduce two new types of chains, which we use later.

Chains in weird gates. Using our definition of chains, we can now see that a typical weird gate builds on a race condition between two types of chains, defined by the purpose of the last instruction of the chain. In *signal* chains, the last instruction, if executed, leaves observable changes in the micro-architectural state of the processor. In all of the examples in this paper, signal chains introduce a memory access to an output address, which depends on the result of the preceding chain. One such example is Line 5 in Listing 1, which uses an access to the address pointed by `out`, with a dependency on the result of the preceding delay. Conversely, a *control* chain typically ends with an instruction whose outcome is mispredicted, resulting in transient execution of (parts of) a signal chain. In all of the code examples in this paper, we use the return misprediction [17], which we abstract as a call to the function `mispredict_ret()`.

When the gate executes, the race between a control and a signal chain determines the gate's output. When the processor encounters the last instruction of the control chain, it mispredicts its outcome and proceeds to transiently execute instructions of the signal chain. At some stage, when the last instruction of the control chain executes, the misprediction is detected. If at this stage the last instruction of the signal chain has already been executed, i.e. the signal chain wins the race, the state of the micro-architecture will be changed. Conversely, if the control chain wins the race, the signal chain is squashed before the last instruction executes, and the state is not changed.

As a concrete example, the chains of a NOT gate are depicted in Figure 1. In each of the parts of the figure, the control chain consists of the operations in the left column of the figure, i.e. accessing `in` and setting the return address. The signal chain, consisting of a delay and an access to `out`, is on the right column of the gate operation diagram.

```
delay_chain1(tmp) {           delay_chain2(tmp) {
  tmp += tmp;                   tmp = sqrt(tmp);
  ...                           ...
  tmp += tmp;                   tmp = sqrt(tmp);
  return tmp;                   return tmp;
}                             }
```

Listing 3: Examples of delay chains.

Delay chains. Chains can also be classified based on their timing behavior. *Delay chains* are designed to take a fixed number of cycles before falling through to the instructions that control their purpose. In these chains, each instruction typically depends on its predecessor in the chain. The delay chain is designed to produce a known output, which is then used as part of the signal or control operation of the chain. In all of the chains that we use, the inputs and output are always zero. Listing 3 shows two examples of delay chains, one using additions and the other using square root operations. The number of repetitions of the operations controls the length of the delay. While we present C-like code, in practice we implement the delay chain in assembly. This allows us better control of the generated code and, in particular, facilitates interleaving of multiple chains.

Probe chains. In contrast with delay chains, probe chains are dependent on variable time operation. They can be used to "measure" the time an operation takes, typically compared to a delay chain. In all of the probe chains we use, the chain depends on one or more memory accesses, such that the chain takes a long time if any of the accessed locations is not in the cache.

Combining chains. Different gates are constructed using different combination of chains. As we described, a NOT gate consists of a fixed delay signal chain and a probe chain for the control. A NAND gate has a similar construction, with the probe chain depending on multiple inputs, rather than just one. Conversely, BUFFER and AND gates have fixed delay chains at their control arm and probe chains at the signal arm.

Some gates include multiple chains of each type. All gates with multiple outputs use multiple chains that share their timing component, but not their signal instructions. Similarly, the NOR and MAJORITY gates of Katzman et al. [18], which consist of multiple control chains, all of which are probe chains, and a single, fixed delay, signal chain. Last, the OR gate of Wang et al. [40], uses multiple signal chains, each probing a different input, but all using the same output.

New Chain types. Utilizing of the flexibility of the terminology, we now introduce two types of chains, which have not been used in the literature so far. These new chains present alternative approaches for implementing prior gates and support new functionality. They also form the basis for our main contribution, the Spec-o-Scope attack.

Multi-probe Chains. Multi-probe chains are chains that probe a sequence of multiple input addresses. Unlike past chains that access multiple addresses in parallel, e.g. the control chain of NAND gates, the accesses to inputs in a multi-probe chain are serialized by introducing dependencies between consecutive chains. Thus, the delay of a multi-probe chain correlates with the *sum* of the delays of the probes, whereas in prior multiple-input chains it correlates with the *maximum* probe delay.

```
m_probe(in1, in2, in3) {       m_input(in1, in2, in3) {
  t1 = *in1;                     t1 = *in1;
  t2 = *(in2 + t1);              t2 = *in2;
  t3 = *(in3 + t2);              t3 = *in3;
  return t3;                     return t1 + t2 + t3;
}                              }
```

Listing 4: Example of a multi-probe chain with three inputs (left), contrasted with a multiple-input chain (right). The code assumes that the memory contents is always zero. Note that in the multiple-input chain, the memory accesses do not depend on each other and can execute in parallel.

Listing 4 shows an example of a multi-probe chain with three inputs on the left side. Note that when loading successive inputs, the accessed address depends not only on the input, but also on the value read from the previous input. Similar to past works, the code assumes that the content read from memory is always zero; hence, adding it to the address creates a dependency without changing the address. In comparison, in the multi-input chain on the right side, the three input loads are independent and will be executed in parallel.

A potential use of multi-probe chains, which is orthogonal to this work, is to implement a majority gate [18]. As an example, a majority-out-of-5 gate uses a multi-probe chain which accesses all five inputs as a signal chain, and a fixed-delay chain with a delay slightly larger than two cache misses as the control chain. The control chain then wins the race if three or more of the inputs are not cached, but loses if at most two inputs are not cached.

```
1  tapped_gate(out1, out2, out3, in1, in2, in3) {
2    t1 = *in1;
3    t2 = *(in2 + t1);
4    t3 = *(in3 + t2);
5    mispredict_ret(real_return + t3);
6    tout1 = *(out1 + delay1());
7    tout2 = *(out2 + dependent_delay2(t1));
8    tout3 = *(out3 + dependent_delay3(t2));
9  real_return:
10   return;
11 }
```

Listing 5: An example of a gate using a tapped multi-probe chain. Each of the three outputs is the result of a race against a different suffix of the control chain in Lines 2–4.

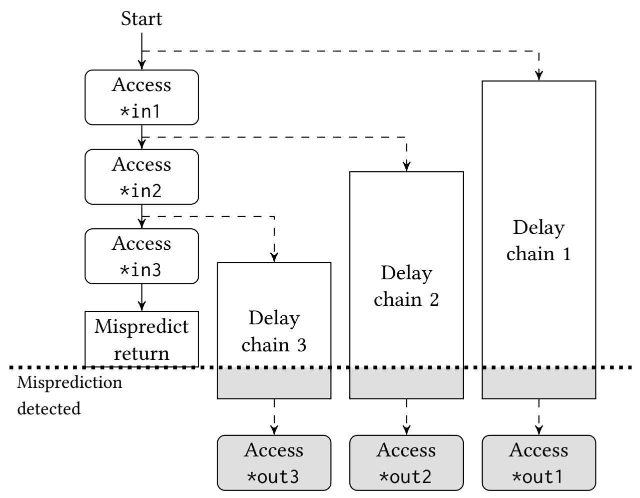

Figure 3: A diagram of a gate which uses a tapped multi-probe chain. Delay chain 1 races against the full multi-probe chain. Delay chain 2 races against the accesses to *in2 and *in3 in the multi-probe chain, and Delay chain 3 races only against the final access in the multi-probe chain.

Tapped Multi-Probe Chains. When a multi-probe chain is used as the control arm of a weird gate, it is possible to attach multiple signal chains, each at a different location in the multi-probe chain. This creates multiple race conditions, between different suffixes of the multi-probe chain and the corresponding signal chains.

An example of a gate that uses a tapped multi-probe chain is shown in Listing 5. The same gate is depicted in Figure 3. The gate includes a multi-probe chain that leads to the control action (Lines 2–4 in Listing 5, left column of Figure 3). Additionally, the gate includes three delay chains, each leading to a separate signal (Lines 6–8, right three columns). Delay chain 1 does not depend on any of the control chain's accesses and therefore races the whole control chain. That is, after executing the gate, `*out1` is accessed and cached if delay chain 1 is faster than the total accesses in the control chain. The second delay chain (Line 7) depends on `t1`, the output of the first access in the multi-probe chain. Consequently, `delay2()` only begins executing after `t1` is available, and thus races only the memory accesses to `*in2` and `*in3`. Finally, delay chain 3 races only the last memory access of the multi-probe chain.

6 MULTIPLE PROBES PER WINDOW

As we show in Section 4, using speculative gates can indeed reduce the time it takes to perform the Prime+Scope attack. However, the reduction is only from 70 to 54 cycles, and most of the gap between scope time and cache access time still remains. As we cannot remove essential operations, our core strategy for improving the temporal resolution of the attack is to perform multiple scope steps within a single speculative window. In this section, we outline the design of our Spec-o-Scope gates that perform multiple probes of the same memory location.

6.1 Spec-o-Scope Gate Design

The core observation behind the Spec-o-Scope gate is that due to the dependency between successive memory accesses in multi-probe chains, the chains can be used to perform multiple accesses to the same memory address. Thus, to realize a Spec-o-Scope gate, we use a tapped multi-probe chain, where all inputs point to the scope address. We then set the signal chains such that they will lose the race if the scope address remains cached throughout the execution of the control chain, but win the race if any of the memory accesses that the signal chain races against misses in the cache.

The resulting timing diagram is shown in Figure 4. In the left-hand side of the figure, we see the case that no victim access is detected, i.e., the scope address remains cached. In such a case, the control arm of the gate executes before any of the signals, and none of the outputs are cached. Conversely, when the victim accesses the target cache set, it causes eviction of the scope address from the cache, forcing a cache miss in the control chain. This cache miss delays the execution of the control chain, allowing *some* of the signal chains to complete (speculatively). Specifically, control chains that are tapped before the access that misses in the cache, i.e., before the victim access, will win the race and access their outputs. Conversely, control chains that tap after the victim access will only start after the miss is handled and will therefore lose the race. This scenario is demonstrated in the right-hand side of Figure 4, where victim access caused the second scope to miss. Consequently, both the first and second delay chains win the race and access their respective outputs. Conversely, the third delay chain, which does not race against the second scope, loses the race and its output is not accessed.

After the gate executes, we can check the outputs to determine whether a victim access has occurred, and if so, when. Specifically, we test whether the output addresses are cached or not. If the first output address is not cached, we know that the victim has not accessed the monitored set during the execution of the Spec-o-Scope gate. Conversely, if the first output address is cached, we need to search for the first non-cached output address in order to determine which of the scope accesses missed. We further recall

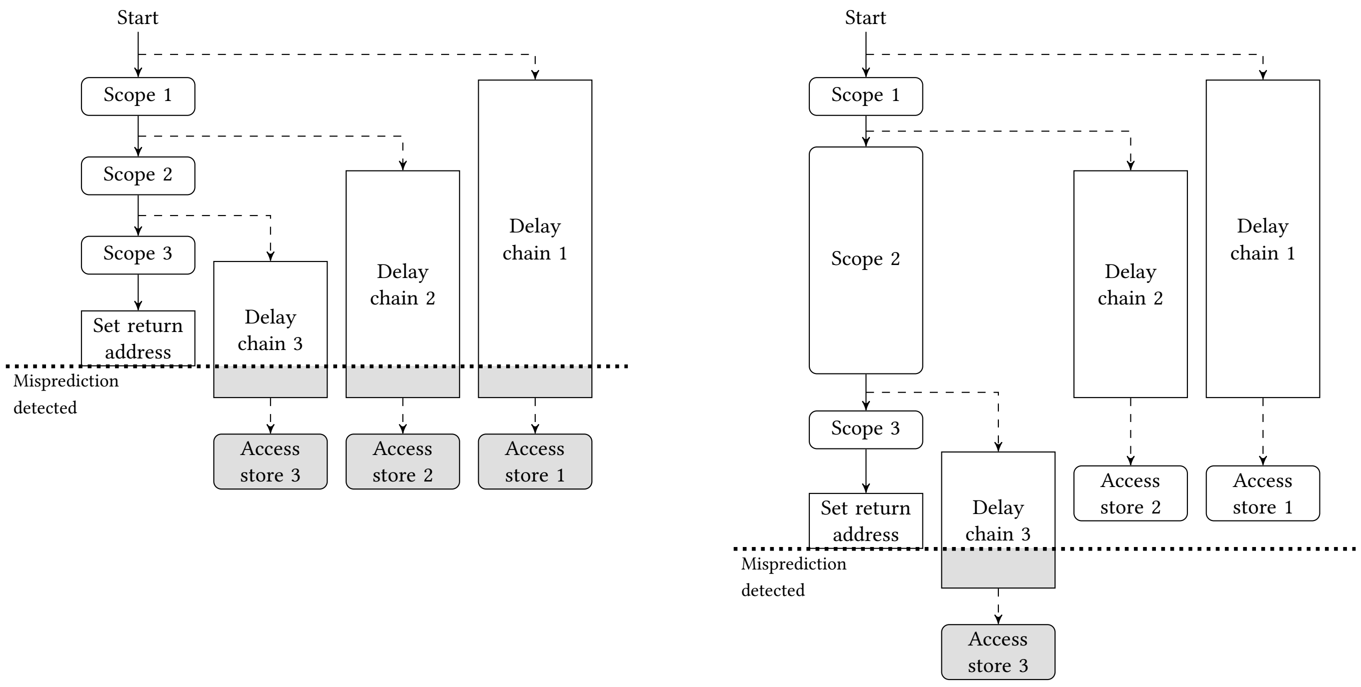

Figure 4: Timing diagram for multiple scopes. Left, without victim access; Right with victim access before the second scope.

that the attacker does not need to test the outputs after each gate invocation, but can wait until the attack is completed.

Implementing the gates, particularly when the number of chains grows, requires some tuning. In particular, we want to reduce the size of the code that executes speculatively to fit within the reorder buffer (ROB). Additionally, we would like to avoid contention on CPU resources, including reservation stations, execution units, and line fill buffers. For that, we select specific instructions to be used in delay chains. To reduce contention, some chains use floating-point instructions (in particular, SQRTSD) and some use arithmetic instructions (in particular, POPCNT). The POPCNT instruction is chosen because it takes up 2 cycles, accepts any register, and is made up of a single micro-op [9]. A concrete implementation of a three-scope Spec-o-Scope gate is presented in Listing 6 in Appendix C. Code for other gate sizes can be found at the repository.

6.2 Spec-o-Scope Gates Evaluation

We now evaluate the temporal resolution that our Spec-o-Scope gates achieve. We first analyze the gates' structure to assess the number of cycles between each scope operation. We then measure the gates' execution time to assess if it matches the analysis results. Last, we complement the measurements by evaluating the sensitivity of gates to event timing.

Gate analysis. The control chain of a gate consists of a sequence of memory accesses. Assuming all hit the cache, as is typical for the Prime+Scope attack, the latency of each such access is 5 cycles [9]. Additionally, the gate includes some overhead, in the form of argument passing, computation, and memory accesses that cause mispredictions, as well as CALL and RET instructions. In Section 4.2 we measured the timing of the NOT gate, which is very similar to our single-scope gate, as about 54 cycles. Consequently, we can extrapolate that an n-scope Spec-o-Scope gate will take $49 + 5n$ cycles when no cache misses are executed.

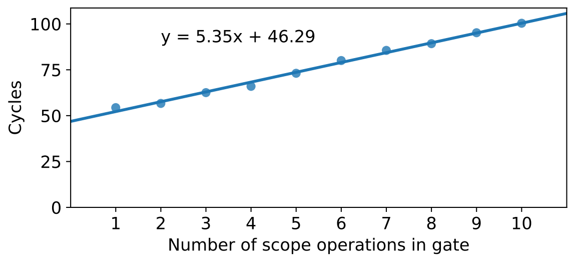

Figure 5: Median execution time of Spec-o-Scope gates.

Gate execution time. To validate the analysis results, we implement ten Spec-o-Scope gates, each with a different number of scope operations and corresponding signal chains. We measure the execution time of 1 000 000 invocations of each gate on an Intel Core i5-8250U, and report the median execution time. Figure 5 shows the results. As we can see, the execution time increases linearly with the number of scope operations. The figure also includes a trend line, showing a slope of 5.35 cycles per scope operation, closely matching the results of our analysis.

Gate sensitivity. To test the gate sensitivity, we use it to perform the Prime+Scope attack against an artificial victim. For the test, the victim and attacker run in two synchronized threads. The victim

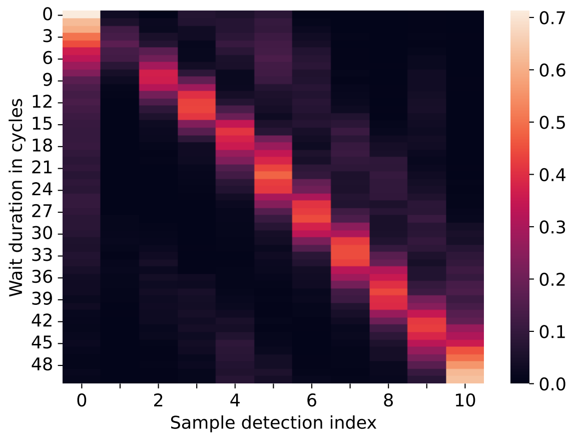

Figure 6: Distribution of the indices of scope operations in which the access is detected for different wait durations (1000 experiments per wait duration).

waits for a small number of cycles (0–50), before accessing a victim address. Concurrently with the victim, the attacker thread executes the Spec-o-Scope gate, noting the index of the scope operation in which the attacker identifies the victim's access. We repeat the experiment 1000 times for each waiting duration and draw the results in Figure 6. For each wait duration, the figure shows the distribution of the indices of scope operation in which the access is detected. That is, the probabilities in each row add up to 1.0.

As the figure shows, the longer the wait time is, the later in the gate the access is detected, advancing roughly one index every 5 cycles. We thus conclude that our Spec-o-Scope gate is sensitive enough to achieve a 5-cycle resolution for the duration of executing the control chain.

6.3 Continuous Attacks

So far we discussed the operation of a single Spec-o-Scope gate. The gate achieves a fine temporal resolution, of a scope operation every 5 cycles. However, it is very limited and can only perform a handful of accesses before the gate terminates. For a continuous attack, the attacker must repeat invoking the gate multiple times.

A 10-scope gate has a latency of about 100 cycles. Thus, on average, we achieve a cache probing resolution of 10 cycles. However, unlike, e.g., Prime+Scope, our scope intervals are not regular. Instead, we perform 10 scope operations at an interval of 5 cycles, followed by a gap of about 50 cycles without scopes.

When a victim's access falls inside this measurement gap, it causes a cache miss in the first attacker scope of the following gate invocation. Consequently, the attacker can detect such accesses. As we will see in Section 7, when the 5 cycles resolution is required, this detection can be used for rejecting traces where the desired resolution cannot be guaranteed.

7 ATTACKING AES S-BOX IMPLEMENTATION

AES S-box-based implementations are generally considered less vulnerable to side-channel attacks [3] compared to T-table-based implementations. This is because the S-box table is much smaller (one S-box table is comprised of 256 bytes or 4 cache lines, compared to four T-Tables, each comprised of 1024 bytes or 16 cache lines). Moreover, it is accessed at a much higher rate (16 accesses per round compared to only 4). Overall, for each round, the probability that a given cache line in the S-box table is *not* accessed is $(\frac{3}{4})^{16} \approx 1\%$ (compared to $(\frac{15}{16})^4 \approx 77\%$ for T-tables).

To witness an event where a given cache line is not accessed on the first round of decryption, we will need $1/(\frac{3}{4})^{16} \approx 100$ traces of decryption on average. However, an event where the cache line is not accessed in the first two rounds of decryption occurs with a probability of $\approx 0.01\%$ and will require $\approx 10\,000$ traces on average to witness it. This means that for a practical attack on S-box-based implementations, an attacker should be able to distinguish if an S-box cache line was accessed or not in a *specific* AES round (i.e., the first round). In other words, we require an oracle O that can reveal the exact round in which a specific part of the S-Box table was first accessed. This requires a side-channel attack with a very high temporal resolution. As each full round of AES takes ≈ 70 cycles, an oracle that can distinguish between two consecutive rounds requires us to sample at a rate faster than 35 cycles.

Our Spec-o-Scope is the first micro-architectural side-channel attack that is able to sample with such high-temporal-resolution. We note that previous attacks on S-box-based implementations overcome this limitation by either interrupting the run of the encryption code by exploiting non-trivial control of the operating system [3, 28], utilize the now deprecated Intel TSX [5], or by modifying the original code [7].

For our attack, we follow the theoretical attack presented by Cheng et al. [7], where we realize the required oracles with our Spec-o-Scope attack. Their attack is made up of two main steps:

(1) Realize an oracle O_1 that can determine if the first cache line of the S-box (cache line 0) was accessed in the first round of decryption or not. O_1 is then used to find "witness" ciphertexts, i.e., ciphertexts that do not access cache line 0 in the first round. Based on these "witness" ciphertexts, we can recover the top two most significant bits (MSBs) of each byte of the first decryption round key (k^0).
(2) Using our knowledge of the two MSBs of each key byte, we can now efficiently generate ciphertexts that are assured not to access cache line 0 in the first round. We then realize an oracle O_2 that can determine if, for a given ciphertext, the first cache line of the S-box was accessed in the second round of decryption. Based on the resulting second round "witness" ciphertexts we found, we can recover the remaining six least significant bits (LSBs) of each byte of k^0 and conclude our attack.

We will now present how we realized oracles O_1 and O_2 and the experimental results of our attack. For more information about the theoretical attack and the algorithms used for key recovery, we refer the reader to a short recap in Appendix B and Cheng et al. [7].

7.1 First-Round Attack

To realize our O_1 oracle, we require a very high accuracy measurement of the time difference between the start of the AES decryption and the first access to a given S-box cache line. However, we only assume coarse-grain synchronization between our attack code and the decryption process, which is not precise enough for our needs.

To overcome this limitation, we employ two attack threads. The naive solution is to target the S-box's cache line with one thread and target some code line at the beginning of the AES decryption code with the second thread. Although this can work, we have experimentally found that this provides relatively noisy results.

Instead, we use a different approach that results in a cleaner signal and also reduces the number of required measurements. We use our Spec-o-Scope attack to measure the timing difference between accesses to two *different* cache lines in the S-box. We run two threads synced using a flag in shared memory, where each thread runs our Spec-o-Scope attack with 10 probes per speculation window. The first thread targets the first 64 bytes of the S-box (cache line 0), and the second one targets the last 64 bytes (cache line 3). Shortly after launching our attack threads, we run the AES decryption code.

Measuring the time difference between accesses to two different lines of the S-box results in two oracles in a single measurement. If the timing difference is very small, we assume both cache lines were accessed in the first round. If cache line 0 was accessed before cache line 3, we assume cache line 3 was not accessed in the first round, and vice versa.

We note that if both the first and last cache lines are not accessed in the first round of decryption, the timing difference will be similar to the case where both of them were accessed in the first round, and our oracle will return an erroneous result. However, the probability that these two S-box cache lines are not accessed on the first round of decryption is $(\frac{2}{4})^{16} \approx 0.0015\%$. As our attack is designed to work with noisy oracles, this negligible addition to the error rate does not affect our attack, and getting two queries per measurement and a much cleaner signal results in a significantly reduced overall complexity.

7.1.1 Experimental Results. We ran our attack on the same CPU configuration used in the previous experiments targeting a standard S-Box-based implementation extracted from OpenSSL 1.1.1i. [3] Similar to Cheng et al. [7] we do not prefetch the S-Box to the cache. Figure 7 shows the distribution of the measured timing difference for "witness" ciphertexts (i.e., didn't access either cache line 0 or cache line 3 in the first round) and "non-witness" ciphertexts that accessed both cache lines in the first round. Note that the X-axis is the difference between the scope index when access was detected in the first attack thread (targeting cache line 0) and the scope index when access was detected in the second attack thread (targeting cache line 3). The difference we show is slightly skewed, as it does not take into account the measurement gap (that occurs every 10 samples) into account. However, this does not seem to affect this part of the attack. Using a threshold of ±20, we get an O_1 oracle (returns true for a witness ciphertext, i.e., if cache line 0 (cache line 3) was not accessed in the first round) with a false positive rate of only 0.01%, and a false negative rate of 21.75%.

Following Cheng et al. [7], we can use the resulting O_1 oracle to test guesses for the 2 MSBs of each first round key byte. Recall that we get two O_1 oracles in each measurement, one for cache line 0 and one for cache line 3. Figure 8 shows the results of targeting key byte 0 whose 2 MSBs are 0x3. We use our O_1 oracle on 10 000 random ciphertexts, clustered by their first byte's MSBs' value. For each

[3]Code provided in additional materials.

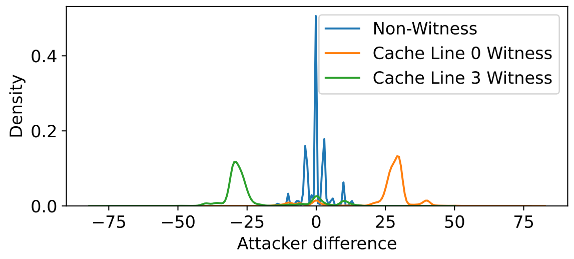

Figure 7: Distribution of timing difference measured by our O_1 oracle attack for witnesses for cache line 0, cache line 3, and when both cache lines were accessed on the first round. Taken over 1000 random keys with 10 000 random ciphertexts for each key.

ciphertext where the MSBs of byte 0 are also 0x3 cache line 0 will be accessed in the first round (the XOR of the MSBs of ciphertext and key at byte 0 is 0x0). In that case, as can be seen in Figure 8, we don't observe any witnesses for cache line 0. Similarly, if the first byte's MSBs' are 0x0, cache line 3 is accessed, and we don't observe any witnesses for it.

In our attack, we recover the key byte's MSBs value by choosing the value that corresponds to the lowest number of witnesses. Although we get some false negatives (witnesses that were not detected), the very low false positive probability still allows us to recover the correct key bytes MSBs with a relatively low number of traces. Moreover, we can reuse the same measurements to find the MSBs of all key bytes.

We ran our attack on 1000 different keys, testing the success rate of our attack for different numbers of traces. Figure 9 shows the success probability of recovering *all* $16 \cdot 2$ MSBs of the first round key as a function of the number of ciphertext traces. 1000 ciphertexts are enough for a 81.5% success rate, while 2000 ciphertexts achieve a 99.5% success rate. We note that measuring 2000 ciphertexts takes, on average, only 0.56 seconds.

7.2 Second-Round Attack

We will now explain the attack on the second round that allows us to recover the rest of the key bits. Recall that the 2 MSBs of the key byes recovered in the first-round attack determine which S-Box cache lines are accessed in the first round. This means that we can use the recovered key bits to generate a set of ciphertexts that are all first-round witnesses, i.e., they will not access cache line 0 on the first round. Using this set, we will now look for second-round witnesses. Our second-round attack has two main parts:

(1) We start by looking for a second-round witness ciphertext (i.e., a ciphertext that doesn't access the first cache line of the S-Box in the first two rounds) out of the set of first-round witnesses we generate.
(2) Next, for each byte of the second-round witness ciphertext we found, we iterate over all possible other 63 values for the 6 LSBs and test if the resulting ciphertext is also a second-round witness or not. We then use this information to recover the remaining 6 LSBs of the key byte.

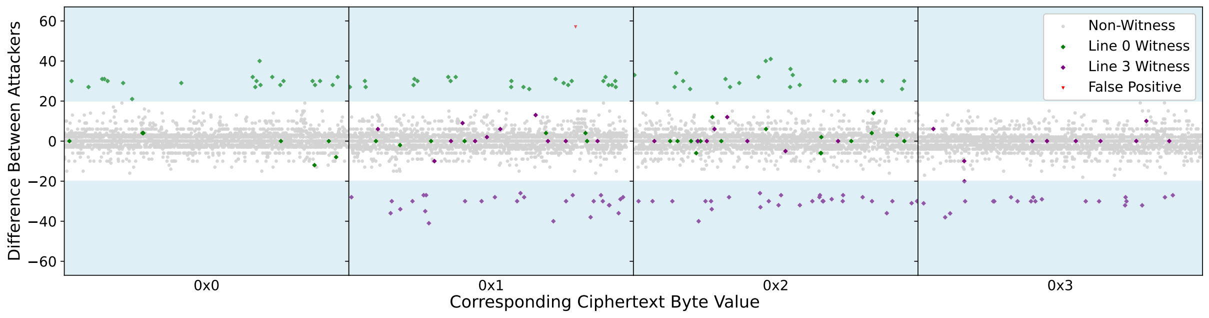

Figure 8: The distribution of the difference measured between attack threads for witness and non-witness ciphertexts as a function of the MSBs' value of byte 0**, measured for** $10\,000$ **randomly generated ciphertexts.**

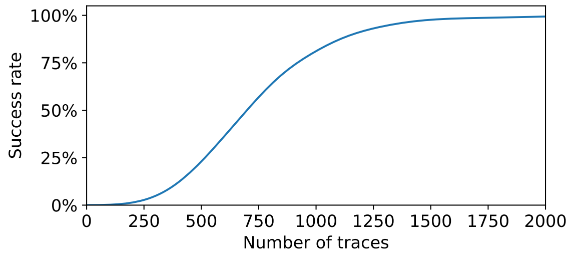

Figure 9: The success rate for full $16 \cdot 2$ **MSBs recovery as a function of the number of traces used.**

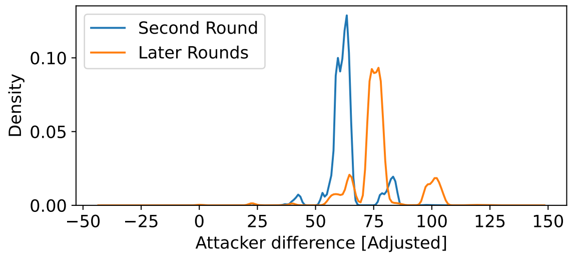

Figure 10: Timing difference distribution measured by our attackers for ciphertexts that access cache line 0 on the second round and ciphertexts that access it on later rounds (i.e., second round witnesses). Taken over 100 **random keys with** 5120 **random first-round witness ciphertexts for each key.**

Assuming the ciphertext chosen in the first part is indeed a second-round witness, for each byte index we get $2^6 = 64$ ciphertexts, where $\approx 34\%$ of them are also second-round witnesses. We enumerate all possible values for the 6 LSBs of the i'th key byte and 8 bits that are a function of first-round state bytes and affect the i'th byte of the second-round state (see Appendix B for details). In total, we need to guess 14 bits. For each guess, we can calculate which of the 64 ciphertexts should be second-round witnesses, and we correlate this guess with the results of our O_2 oracle. To improve the quality of the oracle, we use rejection sampling to discard traces with measurements that might have fallen inside the gate's measurement gap (i.e., detected by the first scope inside a speculative window). In the initial part of the second-round attack, where we find a second round witness, we additionally use majority voting (as detailed in Section 7.2.1), however, for this step of the attack, we do not need to perform majority voting as the correlation-based approach can handle errors.

7.2.1 Finding a Second Round Witness. For the first part of the attack, we require a high-accuracy witness O_2 oracle for a single ciphertext. We improve our accuracy by using rejection sampling to discard traces that have a high probability of being erroneous. We discard any trace with a measurement that might have fallen inside the windows' measurement gap or results in a time difference that is smaller than required for a second-round access. Figure 10 shows the distribution of the measured timing difference (after rejection sampling) for ciphertexts that accessed cache line 0 on the second round and for ciphertexts that accessed cache line 0 on later rounds (usually the third). As before, the timing difference is measured at the difference between the index of the scope operation where access was detected on the two attack threads. However, for this step, we adjusted the difference to take the measurement gap into account, i.e., we increase the index number by 10 after each speculative window.

We tested our O_2 oracle on 100 different keys. For each key, we tested 5120 ciphertexts randomly generated from the set of first-round witnesses (i.e., do not access cache line 0 on the first round). Note that in our experiment, the number of ciphertexts that access cache line 0 on the second round is $\approx$ 100 times larger than the number of ciphertexts that don't (the probability of not accessing is $\approx 1\%$). Our resulting O_2 oracle is noisier than O_1, with a false positive rate of 10.79%, and a false negative rate of 16.76%. The higher error rate is caused by added noise due to longer time differences measured. Moreover, as the number of non-witness ciphertexts is much larger, even a relatively low false positive probability will cause the attack to fail. To prevent false positives, whenever a trace shows a ciphertext is a second-round witness, we repeat the measurement 9 times. Only if a majority of the 9 measurements agree that the ciphertext is indeed a witness will we use it for the second part of the attack. Using this approach, this step of our attack returns a second round witness with a high probability of $\approx 94\%$.

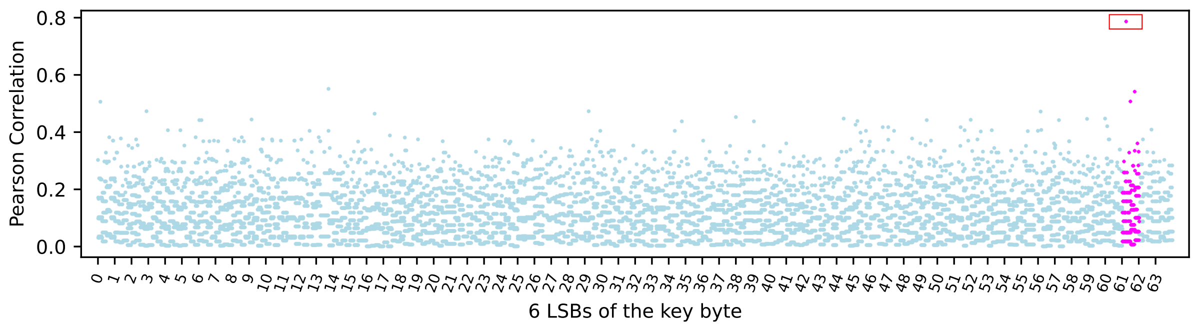

Figure 11: Correlation for each guess of the 14 bits in a single attack on key byte 0. The guesses are clustered by the key byte's 6 LSBs value. The guesses for the correct key bits are highlighted, and the correct guess on all 14 bits is marked.

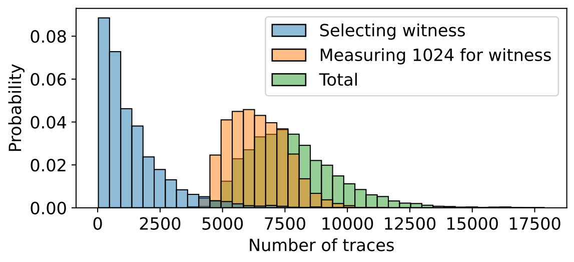

Figure 12: Histogram of the required number of traces to complete the attack. Measured over 1000 random keys and repeated 5 times for each key.

7.3 Recovering the Key Bytes' LSBs

After finding a second-round witness, we start the final step of our attack. For each byte i of the ciphertext, we enumerate and measure all possible values of the lowest 6 LSBs.

Figure 11 shows the Pearson correlation for all possible guesses for the attack on key byte 0. For each guess of the key byte's 6 LSBs, we have $2^8 = 256$ guesses for the other 8 internal state bits. For the correct guess of key bits, we see a single guess for the state bits that has a much higher correlation than all of the other guesses. This is the one guess that is correct on all 14 bits.

We ran the full second round attack on 1000 different keys, re-running the attack for 5 times for each key. Of the 5000 attacks, 48.36% recovered all 128 key bits correctly. Figure 12 shows a histogram of the required number of traces for each step of the second round attack and the total number of traces. The main variance is in the step where we select the witness ciphertext. The number of ciphertexts we sample before finding a witness is a geometric distribution with probability $p = 0.01$. The actual number of traces is larger because of rejection sampling and multiple measurements for majority. The second step requires a total of $64 \cdot 16 = 1024$ traces to recover all key bytes; again, the actual number is larger due to rejection sampling.

7.3.1 Improving Accuracy with Post-processing. While getting a 50% success rate is arguably sufficient (we can just repeat the attack on failure), we can significantly improve our success rate with simple post-processing. If we look at the success rate for each byte separately, across all $5000 \cdot 16$ key byte guesses, we get a much higher success rate of 88%. This means that in a large number of attack attempts, only a small fraction of the byte guesses are incorrect (e.g., up to 4 incorrect bytes). Therefore, if we can identify the incorrect guesses, we can simply brute force them to find the correct values.

In the naive attack, we simply choose the key guess with the highest correlation. Instead, we can consider guesses where the maximal correlation is under a threshold (0.6) to be incorrect. Now, for all attempts where at most 4 byte guesses are assumed to be incorrect, we brute force the unknown 24 key bits. Using this approach, we were able to increase our success rate to 73.92% while reusing the same traces as before.

7.4 Number of Required Traces for Full Attack

Full key recovery requires running both the first-round attack, and the second-round attack. We experimentally find in Section 7.1.1 that 2000 decryption traces are sufficient to recover the 32 upper bits of the key with probability 99.5%. We further find in Section 7.3.1 that the second round attack, which on average requires 7830 traces succeeds with a probability of 73.92%. In total, we expect the attack to require 9830 traces on average to recover the full key, and succeed with a probability of $0.995 \cdot 0.7392 \approx 73.55\%$.

8 CONCLUSIONS

This paper presents the Spec-o-Scope attack, a micro-architectural cache contention attack that achieves an order of magnitude improvement over the current state-of-the-art Prime+Scope attack. We evaluate our attack experimentally, showing that it can discern events with 5 cycles of precision. We also show how to use it to efficiently recover the key from T-Table AES. Finally, we demonstrate the first attack on unmodified S-Box AES that does not rely on strong assumptions such as the ability to interrupt the victim frequently or the availability of the now-deprecated Intel TSX.

Our Spec-o-Scope attack is based on advanced transient "weird gates" that exploit complex interactions between different micro-architectural components. We develop new general terminology to describe these interactions and facilitate designing novel weird gates that are based on them. We believe there is further potential for enhancing attacks using such interactions and that future work should investigate the usage of other micro-architectural components and the best way to exploit them.

ACKNOWLEDGMENTS

We thank Antoon Purnal for the advice and assistance in reproducing the Prime+Scope attack.

This research has been supported by: an ARC Discovery Project number DP210102670; the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972; an ISF grant no. 1807/23; the Blavatnik ICRC; the Len Blavatnik and the Blavatnik Family Foundation; and Robert Bosch Technologies Israel Ltd.

REFERENCES

[1] Alejandro Cabrera Aldaya and Billy Bob Brumley. 2022. HyperDegrade: From GHz to MHz Effective CPU Frequencies. In *USENIX Security*. 2801–2818. https://www.usenix.org/conference/usenixsecurity22/presentation/aldaya

[2] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop van de Pol, and Yuval Yarom. 2016. Amplifying side channels through performance degradation. In *ACSAC (ACSAC '16)*. 422–435. https://doi.org/10.1145/2991079.2991084

[3] C Ashokkumar, Bholanath Roy, M Bhargav Sri Venkatesh, and Bernard L. Menezes. 2020. "S-Box" Implementation of AES Is Not Side Channel Resistant. *HASS* 4 (2020), 86–97. https://doi.org/10.1007/s41635-019-00082-w

[4] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *WOOT*. USENIX Association. https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser

[5] Samira Briongos, Ida Bruhns, Pedro Malagón, Thomas Eisenbarth, and José Manuel Moya. 2021. Aim, Wait, Shoot: How the CacheSniper Technique Improves Unprivileged Cache Attacks. In *EuroS&P*. IEEE, 683–700.

[6] Randal E. Bryant and David R. O'Hallaron. 2016. *Computer Systems: A Programmer's Perspection*. Pearson.

[7] Shing Hing William Cheng, Chitchanok Chuengsatiansup, Daniel Genkin, Dallas McNeil, Toby Murray, Yuval Yarom, and Zhiyuan Zhang. 2024. Evict+Spec+Time: Exploiting Out-of-Order Execution to Improve Cache-Timing Attacks. Cryptology ePrint Archive, Paper 2024/149. https://eprint.iacr.org/2024/149

[8] Dmitry Evtyushkin, Thomas Benjamin, Jesse Elwell, Jeffrey A. Eitel, Angelo Sapello, and Abhrajit Ghosh. 2021. Computing with Time: Microarchitectural Weird Machines. In *ASPLOS*. 758–772. https://doi.org/10.1145/3445814.3446729

[9] Agner Fog. 2022. Instruction Tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs. https://www.agner.org/optimize/instruction_tables.pdf.

[10] Agner Fog. 2023. The microarchitecture of Intel, AMD, and VIA CPUs. https://www.agner.org/optimize/microarchitecture.pdf.

[11] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*. 279–299. https://doi.org/10.1007/978-3-319-40667-1_14

[12] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security*. 897–912.

[13] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *2011 IEEE Symposium on Security and Privacy*. 490–505. https://doi.org/10.1109/SP.2011.22

[14] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. Cache Attacks Enable Bulk Key Recovery on the Cloud. In *CHES*. 368–388.

[15] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing – and Its Application to AES. In *IEEE SP*. https://doi.org/10.1109/sp.2015.42

[16] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2014. Wait a Minute! A fast, Cross-VM Attack on AES. In *RAID*. 299–319. https://doi.org/10.1007/978-3-319-11379-1_15

[17] David A. Kaplan. 2023. Optimization and Amplification of Cache Side Channel Signals. `arXiv/2303.00122`. https://doi.org/10.48550/arXiv.2303.00122

[18] Daniel Katzman, William Kosasih, Chitchanok Chuengsatiansup, Eyal Ronen, and Yuval Yarom. 2023. The Gates of Time: Improving Cache Attacks with Transient Execution. In *USENIX Security*. https://www.usenix.org/system/files/usenixsecurity23-katzman.pdf

[19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE SP*. 1–19. https://doi.org/10.1109/SP.2019.00002

[20] Zili Kou, Sharad Sinha, Wenjian He, and Wei Zhang. 2022. Attack Directories on ARM big.LITTLE Processors. In *ICCAD*. 62:1–62:9.

[21] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. NetCat: Practical Cache Attacks from the Network. In *IEEE SP*. 20–38.

[22] Andrew Kwong, Walter Wang, Jason Kim, Jonathan Berger, Daniel Genkin, Eyal Ronen, Hovav Shacham, Riad S. Wahby, and Yuval Yarom. 2023. Checking Passwords on Leaky Computers: A Side Channel Analysis of Chrome's Password Leak Detect Protocol. In *USENIX Security*. 7107–7124.

[23] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. 2017. Practical Keystroke Timing Attacks in Sandboxed JavaScript. In *ESORICS (2)*. 191–209.

[24] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security*. 549–564.

[25] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security*. 973–990.

[26] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE SP*. 605–622.

[27] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. 2017. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS*.

[28] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *CHES*.

[29] Michael Neve and Jean-Pierre Seifert. 2006. Advances on Access-Driven Cache Attacks on AES. In *SAC*. 147–162. https://doi.org/10.1007/978-3-540-74462-7_11

[30] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. 2015. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *CCS*. 1406–1418.

[31] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA*. 1–20.

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

[33] Antoon Purnal, Marton Bognar, Frank Piessens, and Ingrid Verbauwhede. 2023. ShowTime: Amplifying Arbitrary CPU Timing Side Channels. In *AsiaCCS*. 205–217. https://doi.org/10.1145/3579856.3590332

[34] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. 2021. Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In *CCS*. 2906–2920. https://doi.org/10.1145/3460120.3484816

[35] Aria Shahverdi, Mahammad Shirinov, and Dana Dachman-Soled. 2021. Database Reconstruction from Noisy Volumes: A Cache Side-Channel Attack on SQLite. In *USENIX Security*. 1019–1035.

[36] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. 2019. Robust Website Fingerprinting Through the Cache Occupancy Channel. In *USENIX Security*. 639–656.

[37] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. 2003. Cryptanalysis of DES Implemented on Computers with Cache. In *CHES*. 62–76.

[38] Yukiyasu Tsunoo, Etsuko Tsujihara, Kazuhiko Minematsu, and Hiroshi Miyauchi. 2002. Cryptanalysis of Block Ciphers Implemented on Computers with Cache. In *ISITIA*.

[39] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *SysTeX*. https://doi.org/10.1145/3152701.3152706

[40] Ping-Lun Wang, Fraser Brown, and Riad S. Wahby. 2023. The ghost is the machine: Weird machines in transient execution. In *WOOT*. 264–272. https://doi.org/10.1109/SPW59333.2023.00029

[41] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. 2020. Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures. In *USENIX Security*. 2003–2020.

[42] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy H. Campbell, and Josep Torrellas. 2019. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *IEEE SP*. 888–904.

[43] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*. 719–732.

[44] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2012. Cross-VM side channels and their use to extract private keys. In *CCS*. 305–316.

[45] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2014. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *CCS*. 990–1003.

A ATTACKING AES T-TABLES IMPLEMENTATION

We will now show how to use our new Spec-o-Scope attack on AES T-Tables Implementation. T-Tables-based implementations are one of the main targets for micro-architectural attacks since the seminal attack on AES by Osvik et al. [31]. Purnal et al. [34] show that temporal information about T-Table accesses can significantly improve the number of required traces when compared to the traditional attack on the upper nibbles of a 128-bit key. Indeed, because the attack targets first-round accesses to the T-tables, in traditional attacks, information is gained only whenever a cache line is not accessed in the *entire* decryption process (implying that it is not accessed in the first round). However, as a specific cache line is not accessed during the entire decryption process with a probability of only $(\frac{15}{16})^{40} \approx 7\%$, this requires a large number of traces before such an event occurs. In contrast, an attack that can detect the specific round in which an access happened can utilize every trace. As an example use-case for our high temporal resolution attack, we use it to target a 128-bit T-Table-based AES implementation and recover the full key.

A.1 First-Round Attack

We use our Spec-o-Scope attack to measure the time difference between the access to the first cache line of the AES code and the access to the first cache line of each T-Table. We run two threads, synced by a shared memory flag, each monitoring one of the lines, while we decrypt random ciphertexts. To recover the upper nibble of each key byte, we try guessing the correct value by enumerating over each of the 16 options. For each such guess, we can predict whether or not an access to the first line of the table will occur in the first round of decryption for each ciphertext. For each guess, we compute the Pearson correlation between the measured time differences and the predictions over all ciphertexts. We expect the correct nibble to have the highest correlation. Note that each nibble determines only one of four accesses to the table in the first round, and as such, the prediction might be incorrect even with the correct key nibble guess. However, this is expected to occur only on $(1 - (\frac{15}{16})^3) \cdot \frac{15}{16} \approx 17\%$ of the traces for the correct guess, and we still expect to see a positive correlation between the measured times and the prediction.

A.1.1 Experimental Results. We ran our attack on the same CPU configuration used in the previous experiments targeting a T-Table-based implementation extracted from OpenSSL 1.1.1i. [4] Figure 13 shows the results of an attack aimed at distinguishing between accesses to the first cache line of table T1 that occur on the first round, accesses that occur on the second round (required for the second round attack), and accesses that occur at some later round. Note that the X-axis is the difference between the scope index when access was detected in the first attack thread (targeting the T-Table) and the scope index when access was detected by the second attack thread (targeting the first AES code line).

The difference we show is slightly skewed, as it does not take the measurement gap (that occurs every 10 samples) into account. However, this does not seem to affect our correlation attack. Moreover, if

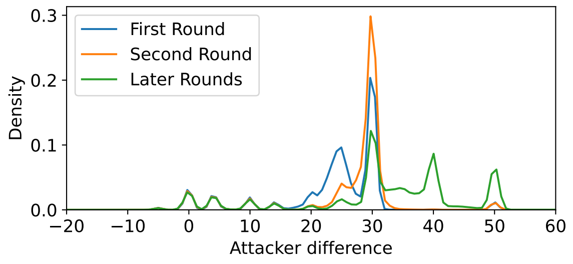

Figure 13: Timing difference distribution measured for ciphertexts accessing the targeted cache line in the first round, the second round, and later rounds. Taken over 1000 random keys with 4000 random ciphertexts for each key.

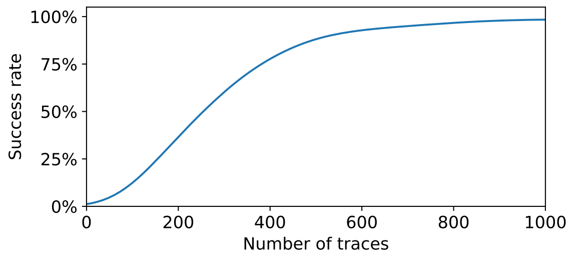

Figure 14: The success rate of recovering 4 upper key nibbles as a function of the number of traces used.

either of the two accesses happens during its attacker thread's measurement gap, it will be detected only in the first scope operation of the next speculation window. This can cause a measurement noise equivalent to up to 10 scope operations. Although we can detect and ignore such measurements, we experimentally discovered that they still have correlative information and are used in our analysis. In contrast, samples with measured differences below a threshold of 20 do not add any information, and we filter them out.

We ran our attack on 1000 different random keys and checked our success rate for different numbers of traces. Figure 14 shows the success rate of correctly recovering the top 4 most significant bits (MSBs) of 4 key bytes (those accessing table T1 in the first round) as a function of the number of traces used in the attack. Using 1000 traces, the attack is able to recover the 4 nibbles with a probability of 98.5%.

A.2 Second-Round Attack

The second-round attack follows a similar methodology to the first-round. Using our new-gained knowledge of the key's upper nibbles, we are able to sample random ciphertexts that do not access the targeted cache line during the first round of decryption. This allows us to focus on distinguishing between second-round accesses and accesses on later rounds.

To recover the remaining 64 bits of the key we once again compute the Pearson correlation between the measured timing differences and access predictions in the second-round. To predict whether an access in the second round falls within the targeted

[4]Code provided in additional materials.

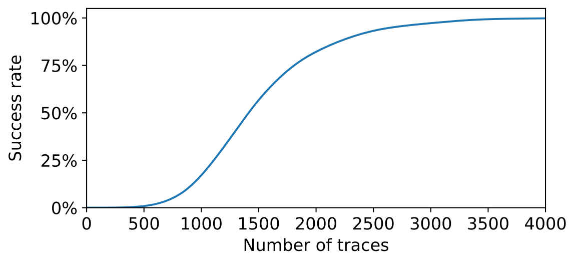

Figure 15: The success rate of recovering 4 lower key nibbles as a function of the number of traces used.

cache line requires knowledge of 4 first-round key bytes, and a single upper nibble in the second round. In total this requires enumerating 20 unknown bits.

We ran our attack on 1000 different keys and checked our success rate for different numbers of traces. Figure 15 shows the success rate of correctly recovering the bottom 4 least significant bits (LSBs) of 4 key bytes as a function of the number of traces used in the attack. Using 3000 traces, the attack is able to recover the 4 nibbles with a probability of 97.6%.

A.3 Number of Required Traces for Full Attack

Full key recovery requires running both the first-round attack and the second-round attack. We experimentally find that 1000 decryption traces are sufficient to recover the 16 upper bits of the key with probability 0.985, so in total, we need $1000 \cdot 4 = 4000$ traces to recover all 64 upper bits with a probability of $0.985^4 \approx 94.1\%$. We further find that the second round attack is able to recover the remaining lower 64 bits using 3000 traces with a probability of $0.976^4 \approx 90.7\%$. In total, we expect the attack to require 7000 traces to recover the full key and succeed with a probability of $0.941 \cdot 0.907 \approx 85.4\%$.

B S-BOX AES THEORETICAL ATTACK RECAP

The following is a short recap of the theoretical attack on S-Box AES, as described by Cheng et al. [7].

We attack an AES decryption algorithm $D_k(c)$, where k is a 16-byte key. We assume access to two oracles, $O_1(c)$, which returns `true` if cache line 0 of the S-box is not accessed by $D_k(c)$ in the first round, and $O_2(c)$, which returns `true` if cache line 0 of the S-box is not accessed by $D_k(c)$ during the first two rounds. Recall that a cache line consists of 64 bytes, and thus the first 64 entries of the S-Box fall within cache line 0. Equivalently, because the S-Box is 256 bytes long, an index i will fall within cache line 0 if its 2 MSBs are 0x0.

The attack, which recovers the first-round key k, consists of two steps; We first recover the 2 MSBs of each key byte using O_1, and then recover the remaining 6 LSBs of each key byte using O_2.

First-Round Attack. Recall that the first round of AES decryption accesses the S-Box once for each state byte, the bytes of $c \oplus k$. Therefore, if $O_1(c)$ is `true`, we conclude that for all $0 \leq i < 16$, the 2 MSBs of $c[i] \oplus k[i]$ are not 0x0, or equivalently, the 2 MSBs of $k[i]$ are not the 2 MSBs of $c[i]$. By sampling enough random ciphertexts, we can rule out all but the correct value of the 2 MSBs. Note that once we know the true values of these MSBs, we can sample random ciphertexts that do not access cache line 0 in the first round.

Second-Round Attack. For the second-round attack, we first search for a ciphertext c for which $O_2(c)$ is `true`. We do so by sampling random ciphertexts for which $O_1(c)$ is `true` and querying O_2. We recover each byte of k separately. The following is a description for $k[0]$. The process for the rest of the bytes is similar. To recover the 6 LSBs of $k[0]$, we query O_2 with the 64 ciphertexts c_j for $0 \leq j < 64$, which are the same as c for all bits except for the 6 LSBs of $c[0]$. Specifically, $c_j[0] = c[0] \oplus j$ and $c_j[i] = c[i]$ for $i \neq 0$. Let s_j be the decryption state just before S-Box substitution, e.g. $s_j[i]$ are the indices used to access the S-Box. Based on the choice of c_j, it can be shown that $s_j[i] = s_0[i]$ for all $i > 3$, and thus these do not access cache line 0. Therefore, $O_2(c_j)$ is determined by the 2 MSBs of $s_j[0]$, $s_j[1]$, $s_j[2]$, and $s_j[3]$. It can be shown that for some s'_0, s'_1, s'_2, and s'_3 it holds that

$$s_j[i] = s'_i \oplus C_i \cdot SB^{-1}(c_j[0] \oplus k[0])$$

where C_i is the corresponding `InvMixColumns` matrix entry and SB^{-1} is the decryption S-Box. In particular, s'_i is independent of j. Therefore, to know if $O_2(c_j)$ is `true` it is sufficient to know the 2 MSBs of s'_0, s'_1, s'_2, and s'_3 and the 6 LSBs of $k[0]$, 14 bits in total. By enumerating all options and comparing the 64 predicted oracle queries against the ground truth one can recover the 6 LSBs of $k[0]$.

C THREE-SCOPE IMPLEMENTATION

Listing 6 shows the implementation of a three-scope Spec-o-Scope gate. Implementations of other gates can be found at the repository.

```
; rsi = bank address
; rdi = trash

jmp start

ret_misprediction:
lea rax, [rip+end]
add rax, rdi
mov [rsp], rax
ret

start:
; Setup, make sure bank is cached
xorpd xmm0, xmm0
mov rax, qword ptr [rsi]
xor rdi, rax

; Sample chain
xor rcx, rcx
cmp rdi, 0xBADF00D
sete cl
mov r8, qword ptr [rax + rcx]
mov r9, qword ptr [rax + r8]
mov r10, qword ptr [rax + r9]

; Mispredict
mov rdi, r10
call ret_misprediction
; --- Transient ---

; First delay chain
mov rcx, qword ptr [rsi + 128]
cvtsi2sd xmm1, rdx
sqrtsd xmm1, xmm1
sqrtsd xmm1, xmm1
sqrtsd xmm1, xmm1
xor edx, edx
ucomisd xmm0, xmm1
sete dl
mov rcx, qword ptr [rcx + rdx]

; Second delay chain
mov rcx, qword ptr [rsi + 136]
cvtsi2sd xmm1, r8
sqrtsd xmm1, xmm1
sqrtsd xmm1, xmm1
sqrtsd xmm1, xmm1
xor edx, edx
ucomisd xmm0, xmm1
sete dl
mov rcx, qword ptr [rcx + rdx]

; Third delay chain
mov rcx, qword ptr [rsi + 144]
cvtsi2sd xmm1, r9
sqrtsd xmm1, xmm1
sqrtsd xmm1, xmm1
sqrtsd xmm1, xmm1
xor edx, edx
ucomisd xmm0, xmm1
sete dl
mov rcx, qword ptr [rcx + rdx]

end:
```

Listing 6: Implementation of a 3-scope Spec-o-Scope gate.