




# The Gates of Time: Improving Cache Attacks with Transient Execution

Daniel Katzman , William Kosasih , Chitchanok Chuengsatiansup , Eyal Ronen , Yuval Yarom 

 *Tel-Aviv University*

 *The University of Adelaide*

 *The University of Melbourne*

## Abstract

For over two decades, cache attacks have been shown to pose a significant risk to the security of computer systems. In particular, a large number of works show that cache attacks provide a stepping stone for implementing transient-execution attacks. However, much less effort has been expended investigating the reverse direction—how transient execution can be exploited for cache attacks. In this work, we answer this question.

We first show that using transient execution, we can perform arbitrary manipulations of the cache state. Specifically, we design versatile logical gates whose inputs and outputs are the caching state of memory addresses. Our gates are generic enough that we can implement them in WebAssembly. Moreover, the gates work on processors from multiple vendors, including Intel, AMD, Apple, and Samsung. We demonstrate that these gates are Turing complete and allow arbitrary computation on cache states, without exposing the logical values to the architectural state of the program.

We then show two use cases for our gates in cache attacks. The first use case is to amplify the cache state, allowing us to create timing differences of over 100 millisecond between the cases that a specific memory address is cached or not. We show how we can use this capability to build eviction sets in WebAssembly, using only a low-resolution (0.1 millisecond) timer. For the second use case, we present the Prime+Store attack, a variant of Prime+Probe that decouples the sampling of cache states from the measurement of said state. Prime+Store is the first timing-based cache attack that can sample the cache state at a rate higher than the clock rate. We show how to use Prime+Store to obtain bits from a concurrently executing modular exponentiation, when the only timing signal is at a resolution of 0.1 millisecond.

## 1 Introduction

Modern processors consist of a large collection of components and algorithms that implement the instruction set architecture (ISA), collectively called the *microarchitecture*. One such

important component is an out-of-order execution engine, which schedules and executes the instructions of the program. Out-of-order execution improves program performance by executing instructions when all their dependencies are satisfied instead of strictly following the program order.

Out-of-order execution is inherently speculative, both because the processor aims to predict the control flow of the program and because it assumes that instructions do not terminate abnormally, e.g., due to traps. Thus, the processor may execute instructions that do not appear in the nominal program execution. Because the results computed by such instructions are dropped and are never committed to the architectural state of the processor, such so-called *transient* instructions were considered an innocuous side effect of out-of-order execution. However, the discovery of transient-execution attacks [9, 40, 43, 92] demonstrated that this is not the case. Specifically, while the results of transient instructions are ignored, the side effects of the computation on microarchitectural components are not reversed. Consequently, an attacker can cause transient execution of instructions that access secret data and transmit it through the state of microarchitectural components, such as caches.

Many microarchitectural components are caches that store the results of recent operations with the aim of accelerating future similar operations. Cache attacks [21, 45, 78], which observe the cache state to leak secret data, have been known for two decades [80]. These attacks target various caches, including data caches [29, 44, 52, 53, 59, 93, 94, 96], instruction and microcode caches [4, 57, 62], address translation [27, 42, 84], and branch prediction [2, 3, 16, 17, 97]. Cache attacks have a devastating impact on the security of cryptographic implementations, including symmetric cryptography [12, 13, 23, 35, 36, 52, 64, 80], public-key systems [7, 19, 54, 55, 65, 96], and even non-cryptographic code [28, 73, 95].

Cache attacks typically need to distinguish cache hits from misses. As the timing difference is very small (less than a hundred nanoseconds), reducing the accuracy of available timers is considered as a line of defense against such at-

tacks [32, 41, 85]. Consequently, to protect against transient-execution attacks that exploit caches, browsers reduce the resolution of the timers they provide [31, 56, 89] and also eliminate some methods of creating artificial timers [70].

Restricting timer resolution also limits the attacker’s ability to find *eviction sets*. These are groups of congruent addresses in the cache, i.e., addresses that map to the same cache set [28, 44, 52, 59, 83, 94], which are used to evict other addresses from the cache. Techniques for finding eviction sets also require the ability to distinguish hits from misses [44, 58, 59, 86]. Hence, low-resolution timers also hamper an essential step for carrying out cache attacks.

Some cache attacks have been designed to use only low-resolution timers [14, 30, 47, 66, 72, 74, 75]. However, to the best of our knowledge, none of these has been adapted for finding eviction sets and the question of finding eviction sets with low-resolution timers remains open. Moreover, in all published attacks, the timer resolution limits the sampling rate. In particular, no high-resolution tracing attacks, e.g., against modular exponentiation [8, 44, 96, 98] have been demonstrated using only low-resolution timers.

A large number of works exploit cache states for implementing transient-execution attacks [5, 6, 10, 11, 30, 39, 42, 43, 46, 60, 61, 66, 67, 68, 69, 71, 76, 81, 82]. However, the reverse question has so far not been investigated. Thus, in this work, we ask the following question:

*Can transient execution improve cache attacks?*

## Our Contribution

In this work we answer the question in the affirmative. We demonstrate that effects of transient execution can improve cache attacks significantly. The core idea is that cache states can affect the length of speculative execution and the way it modifies future state. We develop techniques that allow us to manipulate cache states, amplify it, store information in it, and even compute on it. We then use these techniques to demonstrate how we can perform high-resolution cache attacks with only a low-resolution timer.

The core idea behind our techniques is that cache states can represent information [47]. In our instantiation we use the presence of an address in the last-level cache (LLC) to represent a Boolean state. When a memory address is in the LLC, the value it represents is a logical ‘1’. Conversely, if it is not in the LLC, its value is a logical ‘0’. We note that program execution indirectly affects the logical value associated with an address. For example, reading from or writing to a memory address brings it to the cache, setting its logical value to ‘1’. Conversely, using the `clflush` instruction to evict a memory address from the cache sets its logical value to ‘0’.

We then design gates that operate on the logical value of memory addresses. Each gate has one or more input addresses. The core of the gate consists of an instruction that starts transient execution followed by one or more memory accesses to output addresses. The gate creates an input-dependent race

between the time that the processor detects and squashes the transient execution and the time that the accesses to the output execute. It is designed so that if a logical function of the inputs, such as `NAND`, is `true`, the memory accesses win the race and get executed transiently. Otherwise, if the function is `false`, the transient execution wins the race and the memory accesses do not execute.

We test our gates on multiple processors. All of our gates work with little modifications on Intel and AMD processors. Some of the gates also work on ARM processors, including Apple M1 and Samsung Exynos. Our gates are reminiscent of the “weird gates” of Evtushkin et al. [18]; however, they do not investigate the use of the gates for cache attacks. Moreover, our gates are more generic and more accurate than theirs. See a comparison in [Section 7](#).

To demonstrate the robustness and versatility of our gates, we use them to implement three circuits. To show that our gates allow Turing complete computation, we first implement a four-bit Arithmetic Logic Unit (ALU) [50]. Executing the 250 gates of the circuit produces the correct result in 80% of the cases. Following Evtushkin et al. [18], we then investigate the use of our gates for implementing SHA-1 [49]. Specifically, we implement the round function of SHA-1 using 2 208 gates. The circuit produces a correct 160-bit result in 63% of the cases. Our final example is Conway’s game of life [20]. It consists of 6 464 gates per generation and correctly computes one generation of an  $12 \times 12$  universe in 62.76% of the cases.

We then turn our attention to using the gates for cache attacks. We first show how our gates allow using a low-resolution timer to detect whether a target memory line is in the cache or not. For that, we use gates with a large fan-out to repeatedly replicate the state of the target memory location over multiple memory addresses. We can now access all the addresses with replicated state, amplifying the signal enough to be detected with a low-resolution timer. Overall, we amplify the signal by six orders of magnitude, achieving a timing difference of over 100 millisecond between the cases.

We then use our amplification technique to find eviction sets in the Chrome browser using the built-in JavaScript timer, whose resolution is 0.1 millisecond. Specifically, we use the algorithm of Vila et al. [86] with our amplification. We build an eviction set in approximately 15 seconds on average, showing that reducing clock resolution does not prevent eviction set creation.

Finally, we implement a cryptographic attack with low-resolution timer. We present Prime+Store, a variant of the Prime+Probe attack [36, 44, 52, 53], which is implemented using gates. Prime+Store uses a `NAND` gate to sample the cache state and store the result to a cache state of an unrelated memory address. This decouples the cache sampling from the time measurement, allowing to collect a sequence of samples at a high rate. After collecting multiple samples, we amplify each separately to obtain the probe result. We use the

attack against the implementation of modular exponentiation in GnuPG 1.4.13. We show that although we use a timer with a resolution of 0.1 millisecond, we can sample the cache every 0.33 microsecond—more than two orders of magnitude faster than the clock resolution.

In summary, this paper makes the following contributions:

- We investigate, for the first time, how transient execution can be used for improving cache attacks.
- We show how to use transient execution to build logical gates that manipulate the state of the cache based on whether data is cached or not (Section 3). Our gates are generic and versatile. They can work on multiple architectures and can be implemented in WebAssembly.
- We demonstrate that our gates are robust enough to perform arbitrary calculations in the cache (Section 4).
- We show how to use our gates to amplify a small timing difference and distinguish whether a memory address is cached with a low-resolution timer, and use this amplification to build eviction sets in WebAssembly, using only a low-resolution timer (Section 5).
- We show how to perform high-resolution cache attacks with a low-resolution timer (Section 6).

**Responsible Disclosure.** We disclosed our results and notified affected vendors, namely, Intel, AMD, Apple, Samsung, Arm, Google, and Mozilla. All acknowledged the issue but did not consider that it exposes new threats to their products and did not require an embargo.

## 2 Background

**Cache.** The cache is a small bank of memory located near the processor. By storing recently accessed data, it exploits the temporal and spatial locality of typical programs to bridge the speed gap between the fast processor and the slower memory. Specifically, the memory space is divided into fixed-size *lines*, typically 64 bytes. When the processor accesses memory, it first checks whether the requested line resides in the cache. If that line is found in the cache, or *cache hit*, the memory access is served quickly from the cache. On the other hand, if the requested line is not in the cache, or *cache miss*, the processor is forced to retrieve the line from the main memory, and the memory access is served much slower. The processor then stores the retrieved data in the cache for potential future uses. Since caches have limited capacities, the processor may need to *evict* some lines from the cache to create space for storing the recently retrieved memory lines into the cache.

Modern processors typically use multiple caches, with a common hierarchy that consists of L1, L2, and L3 caches. L1 cache is the smallest and fastest, located in each processor core. The L3 cache or last-level cache (LLC) is the slowest but has the largest capacity and is shared among other processor cores.

Modern caches are usually *set-associative* where they are organized into multiple *sets*, each consisting of a number of

*ways*. Each memory line is mapped to a single cache set and can only be stored in one of the ways of the particular set it is mapped to. Memory addresses that map to the same cache set are called *congruent* to each other. A group of congruent memory addresses is called an *eviction set*.

**Prime+Probe.** Prime+Probe [52] is a cache attack technique that exploits the structure of set-associative caches to leak information. The attack consists of three steps. In the first *prime* step, the attacker prepares the cache into a pre-defined state by filling one or more cache sets with attacker’s data, which is usually achieved through repeatedly accessing elements of an eviction set. In the next step, the attacker simply lets the victim execute and possibly evict some of the attacker’s pre-filled data out of the cache. This happens if the victim accesses memory that is mapped to a cache set previously filled with the attacker’s data. In the final *probe* step, the attacker measures the time it takes to access the data it used during the prime phase. If the data can be retrieved quickly or within short access time, this means that this data was still in the cache, i.e., it was not evicted during the victim’s execution. On the other hand, if it takes a long time to access the data, this indicates that the data has been evicted by the victim, implying that the victim has accessed data which is mapped into that specific cache set. The attacker then uses the mapping between cache sets and address bits to learn the memory address that the victim has accessed.

**The Instruction Pipeline.** Program execution in modern processors is shared between two main components, the front end and the execution engine. The front end fetches instructions from memory, decodes them and transfers them to the execution engine for execution.<sup>1</sup> The execution engine then executes the instructions and notifies the front end when completed. The front end then commits the results of the instructions to the architectural state and retires the instructions.

**Out-of-Order Execution.** To exploit instruction-level parallelism, the execution engine does not execute instructions in the order specified by the program. Instead, it executes instructions when all their arguments are ready and a suitable execution unit is available. To implement out-of-order execution, the front end and the execution engine share a data structure called reorder buffer (ROB), which keeps track of in-flight instructions. Specifically, the front end issues instructions to the execution engine by recording them at the tail of the ROB. The execution engine uses a variant of the Tomasulo algorithm [79] to execute the instructions in some arbitrary order, and records execution termination in the ROB. When the instruction at the head of the ROB terminates, the front end removes it from the ROB and retires it. This ensures that instructions retire in program order irrespective of the order they are executed in.

---

<sup>1</sup>Technically, decoding converts instructions to micro-operations ( $\mu$ ops). The exact distinction between instructions and  $\mu$ ops is not relevant for this work and we use ‘instructions’ to refer to both instructions and the  $\mu$ ops they correspond to.

**Branch Prediction.** When the front end decodes a branch, it often cannot determine the next instruction because the branch condition or target address is yet to be computed. Instead of stalling, the front end tries to predict the branch outcome based on recently executed branches. It then continues to issue instructions based on the prediction. For correct predictions, this saves time by allowing younger instructions to execute before the branch executes. However, if the prediction turns out to be incorrect, the execution engine squashes all younger instructions and asks the front end to resume execution from the correct outcome of the branch. The front end ignores squashed instructions and never retires them. Thus, their results are never committed. The same mechanism is used for squashing younger instructions when an older instruction causes a trap, such as division by zero or memory access violation.

Instructions that are executed and eventually squashed are also called *transient*. While the processor does not commit their results to the architectural state, any effects that executing transient instructions has on the microarchitectural state of the processor are not reversed, leading to potential vulnerabilities [9, 40, 43].

### 3 Gates

This section describes the main ideas behind our implementations of logical gates based on cache states and speculative execution. We explain the computational model, assumptions, and design rationale.

#### 3.1 Computational Model

We use logical gates to implement computation on the microarchitectural cache states of memory addresses. We define the logical value of “uncached” addresses as ‘0’, and of “cached” addresses (either in L1, L2, or LLC) as ‘1’. Changing the logical state from ‘0’ to ‘1’ is straightforward: we simply need to read the data stored in an address, and it will be fetched into the cache. However, we do not assume access to low-level instructions (e.g., `clflush`) that can set the cache state to ‘0’ directly. Instead, we assume that the initial value for hitherto unused addresses is ‘0’. Our gates compute a logical function of their inputs and store the result in the output. Because testing an input is done by accessing it, our gate evaluation is destructive. That is, computing over the state of an address brings it to the cache, setting its logical value to ‘1’. The main implication of destructive gates is that we cannot reuse the same address as an input to multiple gates.

#### 3.2 NOT Gate

The main insight that motivates our design is that the length of the speculation window of a mispredicted branch is not fixed.

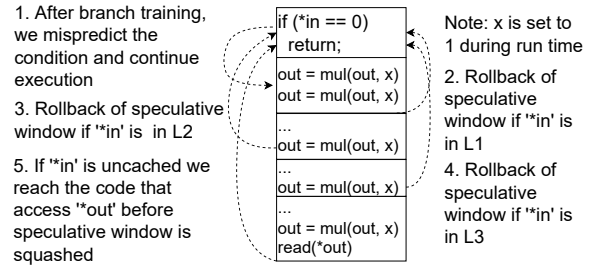


Figure 1: NOT Gate. `read(*out)` is transiently executed only when `*in` is not cached.

Instead, it depends on the time it takes the processor to evaluate the condition of the mispredicted branch. The length of the speculative window determines the number of instructions that are speculatively executed after the branch. Hence, the number of instructions that are executed speculatively varies with the time it takes to resolve the branch condition.

Figure 1 shows how we use these variable-length speculative windows to create a NOT gate. We use a branch conditioned on `*in`. Architecturally, we have `*in=0`. Hence the code should return without executing any of the subsequent instructions. However, the value of `*in` is not immediately available to the CPU as the pointer needs to be dereferenced

Before using the gate, we train the branch predictor to assume `*in` is not zero. Consequently, while waiting for the branch condition to be evaluated, the processor continues to speculatively execute instructions from the predicted branch. The length of this speculative window depends on the time it takes the processor to evaluate the branch condition, which is dominated by the time it takes to retrieve the value of `*in`.

In turn, the time to retrieve the value of `*in` depends on where that value is stored. If the variable `*in` is cached in the L1 data cache, accessing it will be quick ( $\approx 4$  cycles). The time will be longer if `*in` is retrieved from the L2 cache, and even longer if it needs to be retrieved from the LLC. Finally, if the value of `*in` is not cached, it will need to be retrieved from memory, which would take a few hundreds of cycles.

The mispredicted branch contains a sequence of dummy operations (we use `imul` instructions to repeatedly multiply the pointer `out` by 1) followed by a memory access to the output variable `*out`. Tying the value of `out` to the multiplication ensures that the processor does not execute the memory access before all of the dummy instructions complete execution.

The number of dummy operations is carefully chosen such that if `*in` is cached in any of the cache levels, the speculative window will terminate before the memory access to `*out` is issued and `*out` will not be accessed. However, if the value of `*in` is not cached, the speculative window is long enough and the access to `*out` executes speculatively. Eventually, the processor retrieves the value of `*in` and squashes all instructions on the mispredicted branch. However, because memory accesses execute asynchronously, the memory access to `*out`



will complete even if the instruction is squashed.

If `*in` is cached ('1') then `*out` is not accessed and maintains its original logical value. Conversely, if `*in` is not cached, the memory access to `*out` executes transiently, bringing the value of `*out` to the cache, which sets the logical value to '1'. As we assume the initial state of `*out` is uncached ('0'), the end result is that, after executing the gate, the logical value of `*out` is the inverse of the original logical value of `*in`. Hence, the gate computes the logical NOT function.

### 3.3 More Complex Gates

The technique used for implementing the NOT in Section 3.2 can be extended to implement more complex logical gates. We now demonstrate how we can combine inputs to create a NAND gate and add branches to create a NOR.

#### 3.3.1 NAND Gate

To create a NAND gate, we take our NOT gate and replace the if statement `if (*in == 0)` with:

```
if (*in1 + *in2 == 0)
```

Similar to the NOT gate, after we train the branch predictor, a speculative execution window is opened. It continues to run until the values of both `*in1` and `*in2` are made available to the CPU. The CPU processes the two read requests in parallel. Thus, the length of the speculative window is approximately the longer of the two access times.

If either of the input addresses is uncached, the processor needs to wait until the contents is retrieved from memory, resulting in a long speculation window. Consequently, in such a case, speculative execution reaches the `read(*out)` instruction, setting the value of the output to '1'. On the other hand, if both addresses are cached, the length of the speculative window is shorter. Consequently, the misspeculation is squashed before it reaches the read code, and the state of the output address remains '0'. To summarize, if the state of both input addresses is '1', the output value remains '0'. Otherwise, the output is set to '1'. Hence, the code computes the logical NAND function.

#### 3.3.2 NOR Gate

For a NOR gate, we replace the single if statement of the NOT, i.e., `if (*in == 0)`, with two consecutive if statements:

```
if (*in1 == 0) {return;}
if (*in2 == 0) {return;}
```

If either input addresses is cached, the speculative window of the corresponding if statement is short and speculation is squashed before the processor executes the read. This leaves the state of the output address at '0'. However, if both input addresses are not in the cache, the processor needs to retrieve both values from memory before it can squash the speculation of any of the branches. This allows a long speculation window,

1. After branch training, we mispredict the condition and continue execution

2. Rollback of speculative window if `*in` is uncached

3. Only if `*in` is cached we reach the code that access `*out` and only then rollback the speculative window

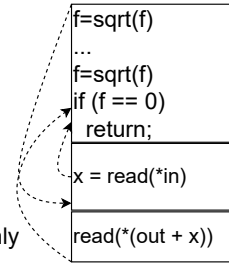


Figure 2: A Buffer Gate with a Fixed Branch Delay.

which would execute the read command, setting the state of the output to '1'. To summarize, only if the state of both inputs values is '0', we get an output value of '1'. Otherwise, the output value is '0'. This is exactly the logical value of a NOR function.

### 3.4 Multiple Inputs and Outputs

Repeating the patterns in Section 3.3 we can increase the number of inputs in the NAND and NOR. For example, for a four input NAND we use the following if statement:

```
if (*in1 + *in2 + *in3 + *in4 == 0)
```

Similarly, we can replicate the output of the gates into multiple output variables by adding read statements to the misspeculated branch. We use the notation  $GATE_{out}^{in}$  for gate  $GATE$  with  $in$  inputs and  $out$  outputs.

The processor uses a structure called *line fill buffer* (LFB) to track memory loads that miss on the L1 cache. Consequently, the number of LFBs limits the number of reads that can be processed concurrently, and the fan-in and fan-out of our gates. Specifically, when the total of the fan-in and fan-out exceeds the number of LFB entries (12 in the processors we use), the gates may fail.

### 3.5 Gates With a Fixed Branch Delay

All of the gates we use operate by creating a race between the time a branch is resolved and the time the outputs are accessed speculatively. In all of the gates we have seen so far, the timing of resolving the branch varies depending on the logical state of the inputs, whereas the timing of the memory access is fixed by the computation of the `imul` instructions. In this section we create additional gate types by swapping the fixed and the variable paths of execution.

Figure 2 shows an example of a BUFFER gate, which copies the logical state of the input to the output. The gate operates by first calculating a sequence of `sqrt` operations, which are set to return the (architectural) value 0. It then branches on the result, returning if the value is indeed 0.

During the computation of the `sqrt` operations, the processor cannot predicate the final result. Consequently, the processor predicts the outcome of the branch, which we exploit by setting the branch predictor to mispredict the branch outcome. During the ensuing speculation window, the processor proceeds to execute the remaining code of the function.

The misspeculated code first reads the input `*in`. It then adds this value to the pointer `out`, and reads from the resulting address. Before using the gate, we ensure that the memory that `in` points to contains the value 0. Thus, the second `read` operation accesses the location pointed by `out`. However, the processor has to read `*in` before it can read `*(out+x)`. Consequently, the timing of the read from `out` depends on whether `*in` is in the cache or not, i.e., whether its state is '0' or '1'. If `*in` is in the cache (state '1'), the read from `out` will be executed before the speculative window is squashed and its state will be set to '1'. However, if `*in` is not the cache (state '0'), the speculative window will be squashed before its value will be made available to the CPU. In that case the second read instruction will not be executed speculatively, and the state of `out` will remain '0'. This is exactly the truth table for a *BUFFER* gate.

As before, by reading more output addresses (e.g., `read(*(out2 + x))`, `read(*(out3 + x))`, etc.) we can increase the fan-out of the gate and extend it to  $BUFFER_Y^1$ .

Similar to the extension of  $NOT_Y^1$  to  $NAND_Y^X$  gate, we can extend  $BUFFER_Y^1$  to  $AND_Y^X$  gate. This is done by making the reading of `*out` dependent on the sum of multiple input addresses instead of a single input address (e.g., `x = read(*in1) + read(*in2)`, `x = read(*in1) + read(*in2) + read(*in3)`, etc.). Only if all input addresses are cached, their sum is available to the CPU, and the read instructions are executed before the speculative window is squashed.

### 3.6 Gates Without Branch Training

In order to open the speculative window, we need to train the branch predictor to mispredict the initial branch we are speculating on. We use the training method introduced in [25]. It starts with two “dry runs” of the gate that train the branch predictor, followed by the actual “wet run”. Moreover, each run of the gate (either “dry” or “wet”) starts with an empty for loop that creates a consistent branch history. As a consequence, the training of the gates is relatively long, increasing the overall run time of our gates.

To reduce training time and allow faster gates, we use a novel approach to cause the branch predictor to reliably mispredict a branch without the need to “retrain” it. The main idea is to replace the single if condition with a large switch statement. The correct case is determined by combining the value of the input address with a counter that is incremented after each evaluation of the gate. As the value of the input address is not available to the CPU, the branch predictor mispredicts by jumping to the previously chosen case based on the counter’s

previous value. According to our experimental evaluation of the gates, switching from the branch training-based approach (“bt”) to our non-branch training-based approach (“nbt”) can significantly reduce the gates’ run time at the cost of a slight reduction in the gates’ accuracy. Thus, the “nbt” gates present a tradeoff between performance and accuracy.

Compilers offer multiple implementations for switch statements. For our technique to work, we need the compiler to use an indirect branch with a jump table. The choice of implementation method depends on the number of cases and their values. In our experiments we find that in both the native and the WebAssembly compiler we use, the compiler chooses a jump-table-based implementation when there are at least eight cases. Once the implementation is chosen, the number of cases has little impact on the gates’ accuracy or performance. Hence we use the required minimum of eight cases for implementing our gates.

The full version of this paper contains code samples for the gate as well as detailed evaluation results.

### 3.7 Gates Evaluation

In our work, we focused our experiments on Intel’s range of processors where we optimized the implementation of our gates to work on those processors. See the full version of this paper for complete experimental results with different gate types including a discussion on the various values for fan-in and fan-out.

Note that many of the gates require tailoring parameters where we perform the tuning on a case-by-case basis (e.g., number of dummy instructions). We can fully optimize many of the gates achieving an accuracy of approximately 99.9% or above. All branch training-based gates achieve an accuracy of over  $\approx 99.5\%$ , while the slightly less accurate non-branch training-based gates still achieve an accuracy of over  $\approx 95.8\%$ . We can also see that our non-branch training-based gates are indeed significantly faster than our branch training-based variants. The non-branch training-based gates are approximately 300 cycles faster. Specifically, they are around twice as fast when the output is '1' and three times faster when the output is '0'.

We could also reproduce similar results on an AMD Ryzen 5 3500U and further implemented some of the gate types on ARM processors. The full version of this paper presents experimental results on the AMD Ryzen 5 3500U and on a Macbook Air laptop with Apple M1 processor and a Galaxy S21 phone with a Samsung Exynos 2100 SoC. We leave the optimization of gates for these platforms to future work.

## 4 Circuits

To demonstrate the versatility of our gates, we now show how they can be used to build complex logical circuits. We

investigate three circuits: the arithmetic logic unit (ALU) from Nisan and Schocken [50], the SHA-1 hash function [49], and finally Conway’s game of life [20].

**Experimental Setup.** We carry out the experiments in this section on an Intel NUC 9 Extreme Kit equipped with an Intel Core i7-9750H CPU running Xubuntu 21.10. Due to the load-sensitive nature of the speculative gates and the cache states, we isolate the core used to run the experiment using the `isolcpus` kernel parameter and enable huge pages.

Our experiments explore the effects of the prefetcher (enable vs. disable) [87] on the accuracy of our circuits. We have also tested the effects of CPU frequency scaling (fixed vs. variable). We find that gates need to be tuned for the processor frequency. However, once tuned, the accuracy of the gates is similar. Moreover, we find that when setting a variable frequency, the processor mostly executes at the highest frequency. Therefore, we only report the results of variable frequency to reflect a more realistic scenario.

## 4.1 ALU

We first demonstrate the viability of our speculative gates through a construction of a four-bit ALU adapted from Nisan and Schocken [50].

The ALU takes two four-bit numbers,  $x$  and  $y$ , along with six control values, as input. Then it produces a four-bit value output. The control values are used to direct the ALU to which operations to perform on each operand. Specifically, by using different combinations of the control values, the ALU performs different operations such as increment, decrement, addition, subtraction, negation, binary AND and binary OR.

To initialize the state of  $x$ ,  $y$ , and the control values, we either read the corresponding address to signify ‘1’ or flush the address to signify ‘0’. We then let the ALU perform its computation. Finally, we read the results by measuring the access time of the output addresses.

We utilize error correction techniques to improve the accuracy of our ALU. Specifically, we run the ALU five times with five copies of the same initial values. We then use the *majority-out-of-5* gate (described in the full version) to compute the final value. Performing such redundant calculation and following a majority vote is a classic error correction approach as used in, for example, redundant coding to detect and correct errors from bit flips [38, 77, 88]. Our ALU is built from 250 logic gates without majority (1 258 logic gates with majority). It consists of 336 intermediate states (1 688 with majority), of which only the four output bits, or 1.19% (0.24% with majority) are exposed architecturally.

We perform 10 000 sets of experiments to measure the performance of our ALU where we focus on the correctness of the four-bit output. Each set of experiments contains 100 runs of the ALU, where the inputs are selected at random. For each run, the accuracy is one if all the four bits are correct. If any of the four bits are incorrect, the accuracy is zero. The

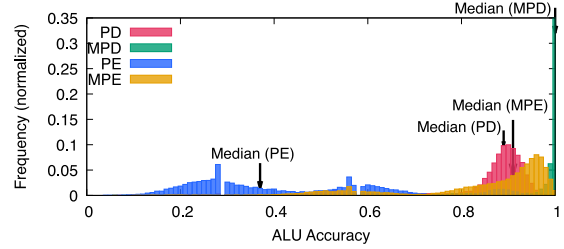


Figure 3: ALU Accuracy. PD denotes accuracy with prefetcher disabled, MPD denotes accuracy with error correction (majority gate) with prefetcher disabled. PE signifies accuracy with prefetcher enabled, while MPE signifies accuracy with majority gate and prefetcher enabled. For visibility, the Y axis is trimmed at a frequency of 0.35.

average of the accuracy of 100 runs represents the accuracy of that set of experiments. Figure 3 illustrates the result of our experiments in histogram. Specifically, it shows the accuracy of the ALU calculation with/without error correction and with/without prefetcher enabled. The histogram clearly highlights a significant increase in the accuracy when using the majority gates, namely, from a median of 89.0% to 100% (82.0% to 95.4% average) when disabling the prefetcher and from a median of 37.0% to 91.0% (43.7% to 84.1% average) when enabling the prefetcher. On average, performing an ALU instruction takes 106 microseconds without error correction and 529 microseconds with error correction.

## 4.2 SHA-1

Our second example of a circuit is an implementation of a cryptographic hash function SHA-1 [49]. Note that in contrast to Evtvushkin et al. [18] our entire round of SHA-1 calculations are performed in microarchitectural states where we interact with them only for the initial state setting and the final output reading. Generally, SHA-1 consists of a loop with 80 iterations to produce a 160-bit message digest output. In our experiment, we perform one round of SHA-1; Listing 1 shows the pseudocode.

```

1 void sha1_round(A, B, C, D, E, W) {
2   temp = circular_shift(5, A) + ((B & C) |
3     ((~B) & D)) + E + W + 0x5A827999;
4   E = D; D = C;
5   C = circular_shift(30, B);
6   B = A; A = temp;
7 }

```

Listing 1: Pseudocode for the first round of SHA-1

As the listing shows, one round of SHA-1 consists of two circular left shifts, four additions, two binary AND, one binary OR, and one binary NOT, each operating on 32-bit words. The calculation of a round of SHA-1 requires 32-bit adder, AND, OR, and NOR. In total, the circuit consists of 2 208 logic gate

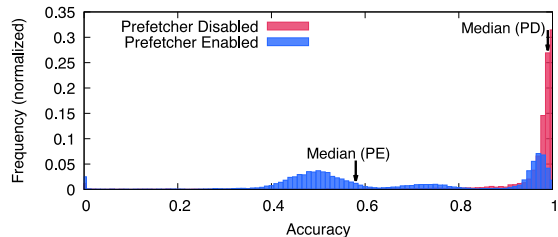


Figure 4: SHA-1 Accuracy

primitives and exposes 1.07% of its 2,976 total microarchitectural intermediate logical states to the architectural state.

We evaluate our implementation by running the SHA-1 and testing the rate of which the full 160-bit result is correctly computed. Specifically, we take measurements of 10 000 experiments, with each experiment performing 100 runs of SHA-1 with random inputs.

Figure 4 shows the distribution of the accuracy for a single round of SHA-1. With prefetcher disabled, we obtain an average and median of 94.95% and 99.00% respectively. When the prefetcher is enabled, we obtain an average and median of 66.55% and 58% respectively. Each round of SHA-1 takes only 969 microseconds to run.

We further evaluate the robustness of our circuit by instrumenting it to compute two blocks of SHA-1, each consisting of 80 rounds. For a complete implementation, we also use four round functions, as specified in the SHA-1 standard [49]. To perform the calculation, we chain consecutive invocation of SHA-1 round circuits. That is, after performing one round, we sample the result and copy it to the architectural state of the processor. We then use the sampled data to set up the cache state for the following round. We further increase the accuracy by computing each round ten times, and using the per-bit majority to determine the output of each round.

Repeating the full computation 1 000 times, we observe a 95.1% probability that the output from our two-block SHA-1 is correct. Each run involves 3 737 600 logic gates and 5 068 800 intermediate values, 1.01% of which is exposed architecturally.

Evtyushkin et al. [18] implement SHA-1 using their “weird gates”. Their implementation relies on Intel Transactional Synchronization Extensions (TSX) [33], a feature of Intel processor but has been mostly disabled due to security issues [34]. Also, their implementation exposes a significant part (41.9%) of the logical state of SHA-1 to the architectural state of the program. In contrast, our implementation uses generic processor features, which are available across multiple architectures, and exposes only 1.01% of the logical states.

### 4.3 Game of Life

As a third example for complex logical circuits based on our gates, we implement Conway’s game of life [20] for a

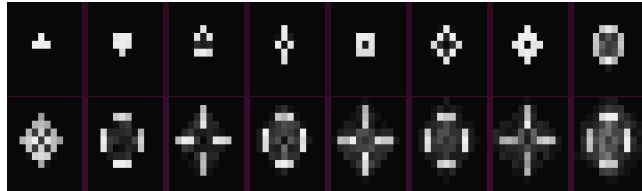


Figure 5: T-tetromino Heatmap (calculated from 300 repetitions, the brighter the cell the higher accuracy)

universe up to size  $12 \times 12$ . Recall that the game is a cellular automaton, consisting of a grid of cells. Each cell has a state which can be either ‘live’ or ‘dead’. Each generation, the state of a cell is updated based on the values of the cell and of its eight neighbors, using the following rules: (1) a live cell that has two or three live neighbors remains live; (2) a dead cell with exactly three live neighbors becomes live; (3) other live cells become dead, and other dead cells remain dead. In our implementation, we denote a live cell by ‘1’ and a dead cell by ‘0’.

According to the rules, calculating the next state of a cell requires evaluating the state of that particular cell and its eight neighbors. Since evaluating a cell changes its value, it becomes crucial to not destroy the state of the cells that are still needed for future evaluations. To tackle this challenge, we microarchitecturally copy the value of a cell into two locations. The first is used to perform an actual calculation while the second is used to restore the original state of the cell.

We implement games over multiple generations with initial states such as T-tetromino [1] and glider. Figure 5 shows 16 generations of the game, starting from a T-tetromino pattern, in a  $12 \times 12$  universe. The brightness of a cell shows the probability that our circuits calculates it as live. Table 1 summarizes the accuracy of a glider across 50 generations for an  $12 \times 12$  universe. We achieve a high accuracy for the first generation; however, due to error propagations onto future generations, the accuracy drops as the game progresses.

Each generation requires 7 808 logic gates to perform its calculation with the total of 11 456 intermediate microarchitectural states. This means that running 10 generations computes 114 560 intermediate microarchitectural states. As we expose only the final output of the circuit to the microarchitectural states, the number of states exposed constantly remains 64 regardless of the number of generations. Specifically, for one generation we expose 0.56% ( $64/11456$ ), whereas for 10 generations we expose 0.06% ( $64/114560$ ). One generation of the game takes 3.19 millisecond to run.

## 5 Probe Amplification

This section demonstrates how we use our gates for side-channel probe amplification. A fundamental ability that most



Generation	Prefetcher Disabled		Prefetcher Enabled	
	Average (percent)	Median (percent)	Average (percent)	Median (percent)
1	59.10	69.00	62.76	73.00
10	48.74	48.00	46.99	48.00
20	22.76	22.00	25.58	25.00
30	15.28	13.00	15.53	11.00
40	17.09	16.00	11.10	9.00
50	10.99	9.00	4.70	3.00

Table 1: Game of Life Glider Accuracy

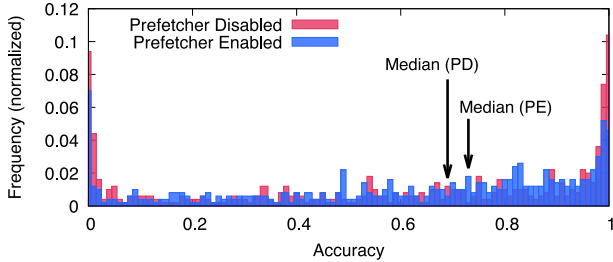


Figure 6: One Generation Game of Life Accuracy

microarchitectural side-channel and speculative execution based attacks require is to determine or “probe” if a specific cache line is in the cache or not. Because the difference in access time is only in the order of approximately 100 clock cycles, this requires access to high-resolution timers. In many settings, e.g., WebAssembly and JavaScript code in modern browsers, access to such timers is actively blocked to prevent this type of attacks. We propose an amplification approach, using our gates, that can amplify the minute timing difference between cache hit and cache miss, allowing the use of timers with arbitrarily low resolution to distinguish the two.

The scheme consists of three amplification steps. In the first step, we use a single gate to achieve a small amplification. In the second step, we create a tree-like structure achieving a theoretical timing difference of up to 4 milliseconds. Finally, in the third step, we combine multiple trees to achieve an arbitrarily long timing difference.

### 5.1 Single-Gate Amplification

The first step in our proposed scheme is using  $NOT_Y^1$ , a NOT gate with a fan-out of  $Y$  (e.g.,  $Y = 4$ ), to gain a small amplification.<sup>2</sup> We denote the access time to an address cached in the LLC by  $t_{in}$ , the access time to an uncached address in the main memory by  $t_{RAM}$ , and their difference by  $\Delta_{cache} = t_{RAM} - t_{in}$ . Assume we want to test if  $addr_{in}$  is cached or not. Instead of directly measuring the access time to  $addr_{in}$ , we use  $addr_{in}$

<sup>2</sup>We use the  $NOT_Y^1$  and not the  $BUFFER_Y^1$  because it is easier to implement for multiple environments (e.g., native and WebAssembly).

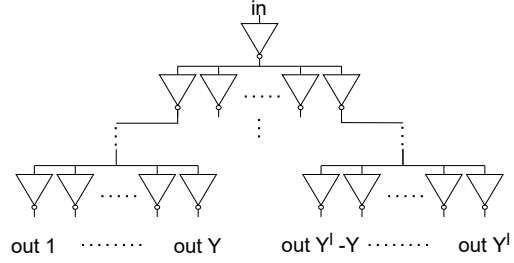


Figure 7: Amplification Tree Based on  $NOT_Y^1$  Gates

as an input for the  $NOT_Y^1$  gate. Then we measure the total time it takes to sequentially access all  $Y$  output addresses. If  $addr_{in}$  was uncached, the  $Y$  output addresses will be cached and the total access time will be  $Y \cdot t_{in}$ . Otherwise, the  $Y$  output addresses will be uncached and the total access time will be  $Y \cdot t_{RAM}$ . This amplifies the timing difference by a factor of  $Y$ , from  $\Delta_{cache}$  to  $\Delta_{gate} = Y \cdot \Delta_{cache}$ .

### 5.2 Probe Time Amplification Tree

As mentioned in Section 3.4, the fan-out is limited by the size of the LFB, allowing only a small constant amplification. To support a larger amplification factor, our next step is to use a tree structure with tree depth  $l$  as shown in Figure 7. Again, we use  $addr_{in}$  as an input to a  $NOT_Y^1$  gate. However, instead of simply accessing the  $Y$  output addresses, we now use each of them as inputs to  $NOT_Y^1$  gates. This gives us a total of  $Y^2$  output addresses. We then continue in the same manner for the full  $l$  layers, resulting in a total of  $Y^l$  output addresses. If the number of layers  $l$  is even, we expect all output addresses to be cached if  $addr_{in}$  was cached, and uncached otherwise. If  $l$  is odd, the cache state of the output addresses is negated. In either case, measuring the total time of sequentially accessing all of the  $Y^l$  output addresses allows us to amplify the timing difference to  $Y^l \cdot \Delta_{cache}$ . Note that the tree is generated in a breadth-first order, i.e., generating each layer before continuing to the next one.

We note that the amplification factor of this tree structure is limited by the cache size as the number of possible output addresses is limited by the number of cache lines. For example, if we assume an LLC of size 8 MB ( $2^{17}$  cache lines) and  $\Delta_{cache} = 100$  clock cycles (0.033 microseconds on a 3 GHz CPU) then  $\Delta_{tree} \leq Y^l \cdot \Delta_{cache} \approx 4$  milliseconds. Moreover, due to practical considerations (e.g., memory prefetcher, risk of self eviction between layers of the tree, etc.), the maximal number of output addresses we can reliably use is much lower.

### 5.3 Amplification Hyper-tree

To overcome the limitations of tree amplification method, we use a hyper-tree structure. We start our hyper-tree amplification by using an  $l$ -layer amplification tree to copy (or negate

if  $l$  is odd) the cache state of  $addr_{in}$  to a bank of  $Y^l$  output addresses. We then continue to iterate over all addresses in the bank. In each iteration, we use the address from the bank as an input to a new tree, then sequentially access all the  $Y^l$  output addresses of the resulting tree. We measure the time it takes to generate the  $Y^l$  sub-trees and sequentially access all the leaves of each sub-tree. Such a two-level hypertree produces a total of  $Y^{2 \cdot l}$  output addresses. However, it only has at most  $2 \cdot Y^l$  “live” memory addresses at each time, a significant improvement over the single tree case. If needed, we can extend this hyper-tree structure to  $d$  levels of sub-trees and a total of  $Y^{d \cdot l}$  output addresses at a space cost of  $d \cdot Y^l$ .

Note that in contrast to the simple tree amplification, we cannot store all of the  $Y^{d \cdot l}$  output addresses in the cache at the same time because the cache might not be large enough. This means that we cannot merely measure the access time to the output addresses but need to measure the entire process of the amplification. This also means that regardless of the cache state of  $addr_{in}$ , we measure the time it takes to access all of the addresses in all the nodes and leaves of the hyper-tree. However, if the addresses in the layer before the last are uncached, the output addresses are accessed from inside the speculative window in parallel. In such a case, it takes  $t_{RAM}$  time to access all  $Y$  addresses inside the speculative window and then  $Y \cdot t_{in}$  time to access them sequentially. However, if the addresses in the layer before the last are cached, they are only accessed sequentially at the end, with a total time of  $Y \cdot t_{RAM}$ . Hence, theoretically we obtain an overall amplified timing difference of approximately

$$\Delta_{hypertree} \approx Y^{2 \cdot l - 1} \cdot ((Y - 1) \cdot t_{RAM} - Y \cdot t_{in}) \approx Y^{2 \cdot l} \cdot \Delta_{cache}.$$

Note that the actual difference is lower due to the access time in the intermediate layers. See the full version for details on the hyper-tree amplification implementation.

## 5.4 Experimental Verification

To demonstrate our amplification hyper-tree scheme, we implement and test it both in native code and in WebAssembly code. We run the experiments on a Dynabook TECRA A50-EC, with an Intel(R) Core(TM) i5-8250U CPU running Ubuntu 20.04.3 LTS. In our experiments, we set the frequency governance to performance. The WebAssembly code was tested under Chromium 99.0.4843.0 (Developer Build). **Figure 8** shows the results of running an amplification hyper-tree in native. The hyper-tree is composed of three tree layers. The topmost layer is an amplification tree from 1 to 16. The bottom two layers are amplification trees from 1 to 512. In total the tree amplifies the access time of a single address to  $16 \cdot 512 \cdot 512 = 4194304$  memory accesses. For each run, we measure the total time it takes to generate the whole tree and access all the leaves. We run the amplification for a total of 1000 times—500 runs with the root address we amplify

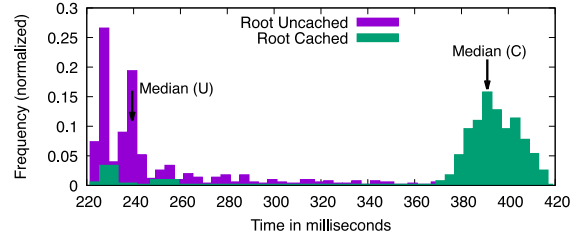


Figure 8: Amplification Hyper-Tree in Native

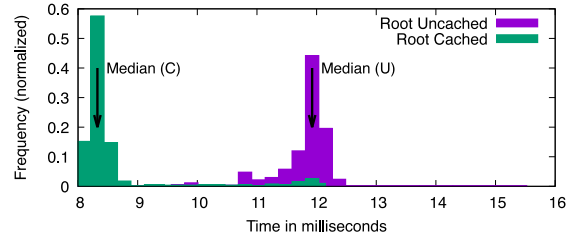


Figure 9: Amplification Hyper-Tree in WebAssembly.

cached, and 500 runs with it uncached. The difference between the median values of the two distributions is more than 100 milliseconds. Our statistical T-test analysis of the two distributions (root cached vs. uncached) yield the p-value (two-tailed) of 0.036, which strongly confirms that we can, indeed, distinguish between the two scenarios.

**Figure 9** shows the results of running an amplification hyper-tree implemented in WebAssembly, after discarding measurements that takes longer than 20 milliseconds as they are too noisy to use ( $\approx 3\%$  of the measurements). The hyper-tree is composed of two tree layers. The topmost layer is an amplification tree from 1 to 192. The second layer is an amplification tree from 1 to 512. In total the tree amplifies the access time of a single address to  $192 \cdot 512 = 98304$  memory accesses. For each run, we measure the total time it takes to generate the whole tree and access all the leaves. Similar to the case of native code, we run 1000 experiments—500 with a cached root and 500 with an uncached root. The difference between the median values of the two distributions is more than 2 milliseconds, which is the current timer resolution provided in the Firefox browser. Similarly, we perform a statistical T-test analysis of the two distributions (root cached vs. uncached) yield the p-value (two-tailed) of 0.010, which, again, strongly confirms that we can, indeed, distinguish between the two scenarios.

## 5.5 Eviction Set Creation

We now show how we use the probe time hyper-tree amplification scheme from **Section 5** to create eviction sets using only low-resolution timers available to JavaScript and WebAssembly code running inside a browser. We implement the eviction set creation algorithm from Vila et al. [86],

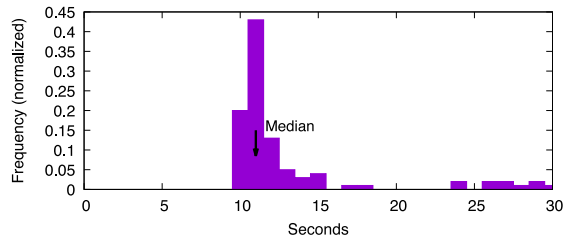


Figure 10: Time to find an eviction set in Chrome using 0.1 millisecond low-resolution timer

while using our probe time amplification to support the low-resolution timers provided by Google Chrome.

We run our experiment on an unmodified Chrome 102.0.5005.61 (Official Build; 64-bit) on the same setup as before. Our WebAssembly code chooses a memory address and tries to find its congruent eviction set that includes 12 addresses. We start with a set of 3000 addresses that, with a very high probability, contain all 128 possible eviction sets with the same page offset as the target address. In 83 out of 100 runs, we are able to find the correct eviction set. Another 8 runs are “close”, meaning that only one address returned is not in the real eviction set. We are not able to find an eviction set for the remaining 9 runs. Failed runs are due to excessive noise. The algorithm detects such failures, and re-running typically finds an eviction set.

Figure 10 shows the measured running time of the algorithm. To summarize, our algorithm is able to find the correct eviction set in 78% of the runs. We only use the 0.1 millisecond resolution timer provided by Chrome, and the median run time of algorithm is approximately 11 seconds.

## 6 Prime+Store: Fast Attacks with Slow Clocks

Slow clocks introduce two problems for microarchitectural side-channel attacks. The first issue is that it is hard to distinguish microarchitectural events with a slow clock; we address this problem in Section 5. The second issue is that the clock limits the rate at which we can measure events; each measurement takes at least one clock tick. This section presents the Prime+Store attack, which overcomes this limitation. We first describe the attack and then demonstrate how we use it against a vulnerable version of ElGamal.

### 6.1 Prime+Store

Our Prime+Store attack is a variant of Prime+Probe. Recall that a Prime+Probe attack consists of two main actions. In the prime step, the attacker accesses all of the members of an eviction set, bringing them to the cache. In the probe step, the attacker accesses the members of the eviction set again to measure the access time and detect if any of the members of the eviction set has been evicted from the cache. Specifically,

the probe step is a function that takes the cache state of the eviction set members and returns `false` if all of them are in the cache and `true` if some of them are not in the cache. We note that under our computational model, if eviction set members are in the cache, they represent the logical value ‘1’. Hence, the probe function effectively calculates the *NAND* of the logical values of the addresses in the eviction set.

Based on this observation, we design our Prime+Store attack using a  $NAND_1^x$  gate, where  $x$  is the associativity of the cache. For the attack, we use an eviction set as the input to the  $NAND_1^x$  gate. This stores the probe result as the cache state of the output of the gate. To perform multiple probes of the same cache set, we repeatedly invoke the  $NAND_1^x$  gate with the eviction set as input, but each invocation we set the output to a different memory address. After we finish sampling, we can then test each of the memory addresses to determine the outputs of the gates. Thus, using this technique, we decouple the cache measurements from the sampling, allowing us to perform repeated samples at a high rate.

Recall that the total fan-in and fan-out of our gates is limited by the size of the LFB, but the fan in of the  $NAND$  gate used for the probe operation is the associativity of the LLC. Hence, if the associativity of the LLC is larger than the size of the LFB, the  $NAND$  gate may fail to work. We note, however, that in most cases, the victim only evicts one entry from the cache. Consequently, most of the eviction set remains cached. Accesses to cached memory free the LFB fast, allowing the attack to operate even though the fan-in is larger than the size of the LFB. The attack may still fail when several entries of the eviction set are evicted from the cache. We ignore this case, considering the failure as noise. If such noise is not acceptable, the attacker can use a more complex circuit to compute the *NAND* function using multiple gates.

### 6.2 Attacking ElGamal

To demonstrate the effectiveness of Prime+Store, we use it to recover the private key from a vulnerable implementation of the ElGamal public-key encryption scheme [15]. Specifically, we target the modular exponentiation operation, which raises a base  $b$  to the power  $e$  modulo some modulus  $m$ , i.e., calculating  $b^e \bmod m$ . During ElGamal decryption, the private key is used as the exponent  $e$ . Hence, our attack aims to recover the exponent.

The attack itself consists of three steps. We first collect traces of memory activity that correspond to segments of the modular exponentiation operation. We then process these traces to recover the operations performed during the observed segments. Finally, we “stitch” the segments to recover the private key. In this subsection we describe the attack setup and the victim we target. Following subsections describe the steps of the attack.

**Attack Setup.** We run the experiment on Dynabook TECRA A50-EC, with an Intel Core i5-8250U CPU running Ubuntu

20.04.3 LTS with LFB size 12. The frequency governance is set to performance. We run two processes, a victim and a spy. The victim uses GnuPG 1.4.13 to repeatedly perform ElGamal decryption with a 4096-bit modulus. With this parameter, the GnuPG private key is of length 457 bits. The spy performs the attack, collecting traces as described below in Section 6.3.

**Victim Implementation.** To calculate the modular exponentiation, GnuPG 1.4.13 uses the square-and-multiply algorithm. The algorithm consists of three main operations, square, multiply, and modular reduction. It scans the bits of the exponents from the most to the least significant. For a bit value zero, it performs square followed by a modular reduction. For a bit value one, it performs a sequence of square, modular reduction, multiply, and modular reduction. The algorithm is known to be vulnerable to side-channel attacks and has been attacked multiple times [44, 96, 98]. Specifically, by recovering the sequence of square and multiply operations that the algorithm performs, the attacker can recover the exponent.

### 6.3 Trace Acquisition

In our attack, we find an eviction set for the code of the square operation and use Prime+Store to repeatedly sample cache usage in the cache set. Once we have collected several samples, we use our amplification technique from Section 5 to amplify each sample to allow recovery with our clock.

Because measurement takes a long time, data we collect may decay, e.g., due to spurious cache evictions. To overcome the decay, we use two techniques. First, we observe that decays tend to change values only in one direction ('1' to '0'). We therefore oversample and then coalesce three consecutive samples using a NOR gate before measuring the result. Hence, unless all three samples decay, we do not miss a '1' sample.

However, when measurement time is long, oversampling is not sufficient to avoid decay. Consequently, as a second measure, we limit the number of samples we collect in each trace, so that the trace only correspond to a small segment of the exponentiation operation.

For the attack on ElGamal, we collect a total of 100 000 traces, each consisting of 2 793 samples, at a rate of 0.33 microsecond per sample. We then coalesce groups of three consecutive samples, and amplify for measurement with a 0.1 millisecond clock. Measurement of a trace takes approximately 0.34 seconds, resulting in a trace of 931 coalesced samples, which we further process. Overall, collecting 100 000 traces takes 9 hours and 40 minutes.

The limited number of samples means that we can only observe a single segment of the exponent at a time. As in past works [67, 71, 98], our aim is to collect a large number of segments and then stitch them together.

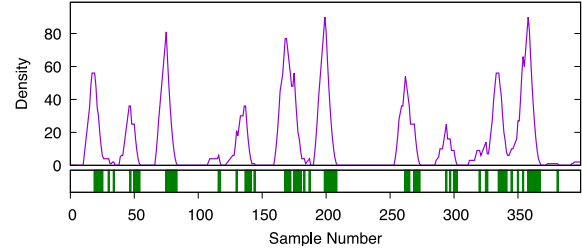


Figure 11: A segment of samples of the square operation in modular exponentiation. The bottom shows samples in which we detect eviction. The top shows the sample density. Peaks with density above 15 correspond to a square operation.

### 6.4 Trace Processing

After collecting the traces, we process them to recover the sequence of square (S) and multiply (M) operations in the segments of the modular exponentiation that they cover. The bottom part of Figure 11 shows an example of a trace. (Trimmed to 400 samples for clarity.) Shaded areas indicate that our Prime+Store attack detected activity in the set in the corresponding coalesced sample. We clearly observe blocks of cache activity that indicate a square operation. However, the samples are noisy, with both gaps during square operations and spurious activity between squares.

To detect the blocks, we measure the density of evictions in an area. For each sample, we count the number of evictions in the subsequent 9 and 15 samples, and multiply the counts to obtain a measure of the density. The top part of Figure 11 shows the density measure for the displayed trace. We then perform peak detection to identify the samples at which a square operation starts. Specifically, we define a peak as a sample that has a higher density than any of the preceding and subsequent eight samples. Peaks with density above a threshold of 12 indicates a start of a square operation.

After recovering the positions of the square operations we use the distance between consecutive square operations. Specifically, we find that when the exponent bit is 0, the distance between consecutive squares is about 30 samples, and when the bit is 1, the distance is around 60 samples. As such, we assume that peaks at a distance of 15–45 samples are consecutive square operations, whereas peaks at a distance of 46–75 samples are a square followed by a multiply. Thus, the sequence of operations in the segment covered in Figure 11 is SSSMSSSSMSSSSM. We ignore peaks that are closer than 15 samples to the previous peak, and treat peak distances of over 75 samples as unknown operations.

### 6.5 Key Recovery

The next step is to “stitch” the segments and recover the key. For that we draw on an algorithm from DNA sequencing [90], adapted to the binary case. Our stitching algorithm relies on



the observation that long enough sequences of square and multiply operations are unlikely to appear more than once within the exponent. Thus, the algorithm iteratively extends a guess of the sequence of square and multiply operations used during exponentiation with the key. For the sake of exposition we first explain a naive algorithm that assumes no errors in the traces.

**Naive Algorithm.** Our naive algorithm starts from the longest segment, which it uses as the current guess, and iteratively extends it to recover the full key. If there are multiple longest segments, it just picks one arbitrarily. To extend the guess, the algorithm searches all captured segment, looking for the matching segment with the largest overlap with the current guess. That is, it looks for a segment that when aligned at some position, has matches on all the positions that overlap with the guess, and out of those it picks the one with the longest overlap. It then merges the segment into the guess, extending the guess in the case that the segment extends beyond the guess. The algorithm stops when running out of segments or when the complete key is recovered.

**Handling Trace Errors.** The main problem with the naive algorithm is that traces do contain errors. To handle errors we modify the algorithm slightly. In the modified algorithm, instead of just associating an operation with a position, we track the likelihood for both operations, guessing the more likely for each position. To generate the initial guess we search for a segment that repeats the largest number of times in the collected traces, and use it for initial guess.

Instead of keeping a single guess for each position, we track the support for a square and a multiply operation. If support for square is larger than support for multiply, we predict that the operation is square. Otherwise, we predict that the operation is multiply.

To extend the guess, the algorithm searches the collected segments for the segment that has the largest match with the prediction in the guess. It then calculates the weight of the segment, which is the number of positions in which the operation in the segment agrees with the operation predicted for the matching location in the guess. Finally, the algorithm updates the prediction by adding the weight of the segment to the guess support in each position.

## 6.6 Evaluation

As discussed in Section 6.3, we collect 100 000 traces over a period of almost 10 hours. Among those, 613 attempts fail, leaving us with 99 387 traces.

We first filter collected segments to remove apparent trace errors. Specifically, we ignore segments that contain 11 or more consecutive square operations. We find that such sequences often appear due to capture errors, and their inclusion confuses our stitching algorithm. We note that with a 457-bit ElGamal private key, we expect about one in five keys to include such a sequence. Omitting these sequences means

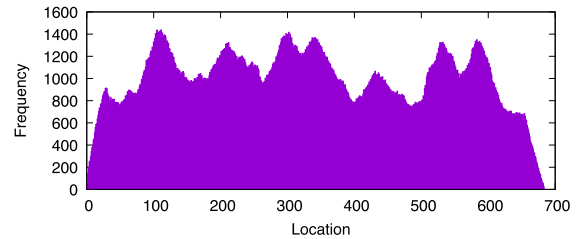


Figure 12: Distribution of Stitched Key in Relation to Ground Truth Location

that for such keys we will need an extra step of adding the missing segments. Overall, there are 12 568 such segments.

To test the coverage of the trace collection, we compare each collected segment with the ground truth. Figure 12 shows the distribution of the positions at which the segments best match the ground truth. Focusing on segments with length of 20 or more operations, we find that 3 572 segments out of the remaining 86 819, completely match the private key. Furthermore, 9 467 long segments match the ground truth with one operation error, and 14 007 match with two errors. Running the stitching algorithm, we find that after merging 22 863 we recover the full key. The process takes 14 minutes and 31 seconds.

## 7 Related Work

**Early Attacks from the Browser.** Genkin et al. [22] demonstrate an LLC attack in WebAssembly, recovering the keys of multiple cryptographic schemes. Oren et al. [51] implement a cache attack in JavaScript, performing website fingerprinting. Gras et al. [26] use a cache attack to break address space layout randomization. All use high-resolution timers that were available to JavaScript and WebAssembly at the time.

**Alternative Timers.** Several works investigate alternative timers for use in web browsers [41, 63, 70]. Following Wray [91], Schwarz et al. [70] propose a timer based on a shared counter, which has been used for Spook.js [5]. Mainstream browsers have eliminated the `SharedArrayBuffer` feature which is used for implementing the shared counter [31, 56, 89]. While some browsers have since partially re-enabled the feature, it is not fully enabled [48].

**Cache Occupancy Attacks.** The cache occupancy attack [74, 75] measures the access time to a cache-size buffer. The timing differences between many and few cache misses can be detected with a low-resolution clock. Moreover, the papers show that by counting the number of times the buffer can be accessed between clock ticks reveals information even when the timer resolution is as low as 100 milliseconds. The downside of the attack is that it has a low spatial resolution, i.e., it provides a proxy of the amount of memory activity, but does not reveal information on which memory addresses are

accessed. Under ideal conditions, the attack can be used for cryptography [24], but it is not clear how this translates to real environments. Moreover, due to the time it takes to access the whole buffer, the temporal resolution of the attack is low.

**Amplifying through Repetition.** When the event that sets the cache state can be repeated, repeating it multiple times can amplify the difference between hits and misses. Such amplification has been used for a Spectre attack, where the attacker can repeat the attack as long as the leaked contents do not change [30, 47, 71]. Theoretically, this approach could also be used to find eviction sets, but we are not familiar with any implementation of such approach and it would seem that a naive implementation will be too noisy to be effective.

**PLRU Attack.** Röttger and Janc [66] propose an L1 cache attack that exploits the Pseudo-LRU replacement algorithm used in the Intel L1 caches. L1 attacks do not apply across cores, but it may be possible, in combination with the Prime+Scope attack [59], to apply cross-core attacks. However, it is not clear if and how the technique can be used for finding eviction sets with low-resolution timers.

**Weird Gates.** Evtvyushkin et al. [18] describe “weird” gates that use transient execution to compute over microarchitectural state. They propose two types of gates. BP gates compute a logical function of the state of the branch predictor and the memory location that contains the code for the gate. They store the result as a state of a location in the data cache. Due to the differences between the types of states used for input and output, BP gates are not composable and it is not clear how to create gates from them.

The second type of gates, TSX gates, use Intel’s Transactional Synchronization Extensions (TSX) [33] to implement the gates. The feature is Intel-specific and is not supported by other processors. Due to security issues [67, 69, 71], Intel disabled the feature by default. Thus, TSX gates cannot work on newer and patched processors. Moreover, JavaScript and WebAssembly do not support TSX, hence TSX gates cannot be implemented in these languages.

While TSX gates are composable like ours, their accuracy (about 99%) is significantly lower than some of our gates’ (over 99.9%). Consequently, Evtvyushkin et al. [18] frequently transfer gates’ output to the architectural state. For example, their SHA-1 implementation exposes 41% of the intermediate values to the architectural state. In contrast, our SHA-1 implementation performs a full round without exposing any intermediate values. Moreover, to increase robustness, the software needs to execute each gate multiple times and perform statistical analysis to decide the likely correct outcome. While our circuits also use redundant computation to increase robustness, we use our majority gates to select the output without exposing the intermediate values to the software.

Evtvyushkin et al. [18] propose to use their gates to implement stealthy computation that is harder to detect and analyze (e.g., a form of program obfuscation to prevent reverse engineering). Similarly, our gates evaluation is also dependent

on the microarchitectural layer, which also stores their state. Thus, the same analysis on stealthiness and program obfuscation apply to our gates.

**Concurrent Work.** In a concurrent and independent work, Kaplan [37] also shows how to use speculative execution to create logical gates. The work mentions the possibility of using branch prediction (which we exploit), but also shows gates based on return address prediction, which require no branch training. It also demonstrates how to combine gates to create logical circuits, to speed up cache attacks, and to amplify cache measurement up to 600 milliseconds.

## 8 Conclusions

We investigate using transient execution to improve cache attacks. We present three types of logical gates that operate on the state of the cache. Our gates are functional enough to perform Turing complete calculations, and are versatile enough to work on a range of environments, including in browsers and across multiple processor architectures. We demonstrate that using our gates we can amplify the timing signal of cache miss vs. hit by six orders of magnitude, achieving a timing difference of 100 milliseconds. Our amplification strategy works well within a browser, and we demonstrate its use for building eviction sets in Chrome, using no timing source other than the JavaScript timer, whose resolution is 0.1 milliseconds. We further present the Prime+Store attack, a variant of Prime+Probe that decouples cache sampling from the timing measurement. We demonstrate the power of Prime+Store by using it to attack the modular exponentiation implementation in a version of GnuPG. We show that we can sample at a rate that is more than 100 times faster than our clock rate, allowing us to obtain (secret) exponent bits from GnuPG. We believe that our gates pave the way for implementing other primitives that manipulate microarchitectural state.

## Acknowledgements

We thank David Kaplan for the useful discussions and for coordinating the publication of his concurrent work.

This project has been supported by an ARC Discovery Early Career Researcher Award DE200101577; an ARC Discovery Project number DP210102670; CSIRO’s Data61; the Blavatnik ICRC at Tel-Aviv University; the Phoenix HPC service at the University of Adelaide; and gifts by Google and Intel.

## References

- [1] Game of life wiki. URL <https://conwaylife.com/wiki/T-tetromino>. 8
- [2] Onur Acıçmez, Shay Gueron, and Jean-Pierre Seifert. New branch prediction vulnerabilities in OpenSSL and

- necessary software countermeasures. In *IMACC*, pages 185–203, 2007. 1
- [3] Onur Aciçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *CT-RSA*, pages 225–242, 2007. 1
- [4] Onur Aciçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *CHES*, pages 110–124, 2010. 1
- [5] Ayush Agarwal, Sioli O’Connell, Jason Kim, Shaked Yehezkel, Daniel Genkin, Eyal Ronen, and Yuval Yarom. Spook.js: Attacking Chrome strict site isolation via speculative execution. In *IEEE SP*, 2022. 2, 13
- [6] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, Frank McKeen, Fangfei Liu, Ron Gabor, Christopher W. Fletcher, Abhishek Basak, and Alaa R. Alameldeen. Speculative interference attacks: breaking invisible speculation schemes. In *ASPLOS*, pages 1046–1060, 2021. doi: 10.1145/3445814.3446708. 2
- [7] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. “Ooh aah... just a little bit”: A small amount of side channel can go a long way. In *CHES*, pages 75–92, 2014. 1
- [8] Daniel J. Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding right into disaster: Left-to-right sliding windows leak. In *CHES*, pages 555–576, 2017. 2
- [9] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtvushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security*, pages 249–266, 2019. 1, 4
- [10] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant CPUs. In *CCS*, pages 769–784, 2019. 2
- [11] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten-Hwang Lai. SgxPectre: Stealing intel secrets from SGX enclaves via speculative execution. In *IEEE EuroS&P*, pages 142–157, 2019. 2
- [12] Chitchanok Chuengsatiansup, Daniel Genkin, Yuval Yarom, and Zhiyuan Zhang. Side-channeling the Kalyna key expansion. In *CT-RSA*, 2022. 1
- [13] Shaanan Cohney, Andrew Kwong, Shahar Paz, Daniel Genkin, Nadia Heninger, Eyal Ronen, and Yuval Yarom. Pseudorandom black swans: Cache attacks on CTR\_DRBG. In *IEEE SP*, pages 1241–1258, 2020. 1
- [14] Patrick Cronin, Xing Gao, Haining Wang, and Chase Cotton. An exploration of ARM system-level cache and GPU side channels. In *ACSAC*, pages 784–795, 2021. 2
- [15] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *CRYPTO*, pages 10–18, 1984. 11
- [16] Dmitry Evtvushkin, Dmitry V. Ponomarev, and Nael B. Abu-Ghazaleh. Jump over ASLR: attacking branch predictors to bypass ASLR. In *MICRO*, pages 40:1–40:13, 2016. 1
- [17] Dmitry Evtvushkin, Ryan Riley, Nael B. Abu-Ghazaleh, and Dmitry Ponomarev. BranchScope: A new side-channel attack on directional branch predictor. In *ASPLOS*, pages 693–707, 2018. 1
- [18] Dmitry Evtvushkin, Thomas Benjamin, Jesse Elwell, Jeffrey A. Eitel, Angelo Sapello, and Abhrajit Ghosh. Computing with time: Microarchitectural weird machines. In *ASPLOS*, pages 758–772, 2021. 2, 7, 8, 14
- [19] Cesar Pereida García and Billy Bob Brumley. Constant-time callees with variable-time callers. In *USENIX Security*, pages 83–98, 2017. 1
- [20] Martin Gardner. Mathematical games: the fantastic combinations of John Conway’s new solitaire game “life”. *Sci. Am.*, 223:120–123, 1970. 2, 7, 8
- [21] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptogr. Eng.*, 8(1):1–27, 2018. 1
- [22] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. In *ACNS*, pages 83–102, 2018. 13
- [23] Daniel Genkin, Romain Poussier, Rui Qi Sim, Yuval Yarom, and Yuanjing Zhao. Cache vs. key-dependency: Side channeling an implementation of Pilsung. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(1):231–255, 2020. 1
- [24] Daniel Genkin, William Kosasih, Fangfei Liu, Anna Trikalinou, Thomas Unterluggauer, and Yuval Yarom. CacheFX: A framework for evaluating cache security. arXiv/2201.11377, 2022. 14
- [25] Google. Spectre. <https://leaky.page>, 2021. 6
- [26] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *NDSS*, 2017. 13
- [27] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating

- cache side-channel protections with TLB attacks. In *USENIX Security*, pages 955–972, 2018. 1
- [28] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security*, pages 897–912, 2015. 1, 2
- [29] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A fast and stealthy cache attack. In *DIMVA*, pages 279–299, 2016. 1
- [30] Noam Hadad and Jonathan Afek. Overcoming (some) Spectre browser mitigations. <https://alephsecurity.com/2018/06/26/spectre-browser-query-cache/>, 2018. Accessed: 2022-01-25. 2, 14
- [31] John Hazen. Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer. <https://blogs.windows.com/msedgedev/2018/01/03/speculative-execution-mitigations-microsoft-edge-internet-explorer/>, 2018. Accessed: 2022-01-25. 2, 13
- [32] Wei-Ming Hu. Reducing timing channels with fuzzy time. In *IEEE SP*, pages 8–20, 1991. 2
- [33] Intel. Intel 64 and IA-32 architectures software developer’s manual volume 1: Basic architecture. <https://cdrdv2.intel.com/v1/dl/getContent/671436>, December 2021. 8, 14
- [34] Intel. Performance monitoring impact of Intel transactional synchronization extension memory ordering issue. <https://www.intel.com/content/dam/support/us/en/documents/processors/Performance-Monitoring-Impact-of-TSX-Memory-Ordering-Issue-604224.pdf>, 2021. 8
- [35] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! a fast, cross-VM attack on AES. In *RAID*, pages 299–319, 2014. 1
- [36] Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. S\$a: A shared cache attack that works across cores and defies VM sandboxing - and its application to AES. In *IEEE SP*, pages 591–604, 2015. 1, 2
- [37] David A. Kaplan. Optimization and amplification of cache side channel signals. arXiv/2303.00122, 2023. 14
- [38] Man Ho Kim, Suk Lee, and Kyung Chang Lee. Kalman predictive redundancy system for fault tolerance of safety-critical systems. *IEEE Transactions on Industrial Informatics*, 6(1):46–53, 2009. 7
- [39] Ofek Kirzner and Adam Morrison. An analysis of speculative type confusion vulnerabilities in the wild. In *USENIX Security*, pages 2399–2416, 2021. 2
- [40] 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. Spectre attacks: Exploiting speculative execution. In *IEEE SP*, pages 1–19, 2019. 1, 4
- [41] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *USENIX Security*, pages 463–480, 2016. 2, 13
- [42] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TagBleed: Breaking KASLR on the isolated kernel address space using tagged TLBs. In *IEEE EuroS&P*, pages 309–321, 2020. 1, 2
- [43] 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. Meltdown: Reading kernel memory from user space. In *USENIX Security*, pages 973–990, 2018. 1, 2, 4
- [44] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE SP*, pages 605–622, 2015. 1, 2, 12
- [45] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. *ACM Comput. Surv.*, 54(6):122:1–122:37, 2021. 1
- [46] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *CCS*, pages 2109–2122, 2018. 2
- [47] Ross McIlroy, Jaroslav Sevcík, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. arXiv/1902.05178, 2019. 2, 14
- [48] MDN Web Docs. Planned changes to shared memory. [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/SharedArrayBuffer/Planned\\_changes](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer/Planned_changes), 2022. Accessed: 2022-01-30. 13
- [49] National Institute of Standards and Technology. FIPS 180-4: Secure hash standard (SHS), 2015. <https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.180-4.pdf>. 2, 7, 8
- [50] Noam Nisan and Shimon Schocken. *The Elements of Computing Systems: Building a Modern Computer from First Principles*. MIT press, second edition, 2021. ISBN 9780262539807. 2, 7
- [51] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *CCS*, pages 1406–1418, 2015. 13



- [52] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA*, pages 1–20, 2006. 1, 2, 3
- [53] Colin Percival. Cache missing for fun and profit. In *Proceedings of BSDCan*, 2005. URL <https://www.daemonology.net/papers/htt.pdf>. 1, 2
- [54] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. “make sure DSA signing exponentiations really are constant-time”. In *CCS*, pages 1639–1650, 2016. 1
- [55] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. To BLISS-B or not to be: Attacking strongSwan’s implementation of post-quantum signatures. In *CCS*, pages 1843–1855, 2017. 1
- [56] Chromium Project. Mitigating side-channel attacks. <https://www.chromium.org/Home/chromium-security/ssca/>. Accessed: 2022-01-25. 2, 13
- [57] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Capkun. Frontal attack: Leaking control-flow in SGX via the CPU frontend. In *USENIX Security*, pages 663–680, 2021. 1
- [58] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. Systematic analysis of randomization-based protected cache architectures. In *IEEE SP*, pages 987–1002, 2021. 2
- [59] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the observer effect for high-precision cache contention attacks. In *CCS*, pages 2906–2920, 2021. 1, 2, 14
- [60] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *USENIX Security*, pages 1451–1468, 2021. 2
- [61] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CrossTalk: Speculative data leaks across cores are real. In *IEEE SP*, pages 1852–1867, 2021. 2
- [62] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M. Tullsen, and Ashish Venkat. I see dead  $\mu$ ops: Leaking secrets via Intel/AMD micro-op caches. In *ISCA*, pages 361–374, 2021. 1
- [63] Thomas Rokicki, Clémentine Maurice, and Pierre Laperdrix. SoK: In search of lost time: A review of JavaScript timers in browsers. In *EuroS&P*, pages 472–486, 2021. doi: 10.1109/EuroSP51992.2021.00039. 13
- [64] Eyal Ronen, Kenneth G. Paterson, and Adi Shamir. Pseudo constant time implementations of TLS are only pseudo secure. In *CCS*, pages 1397–1414, 2018. 1
- [65] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. The 9 lives of Bleichenbacher’s CAT: new cache attacks on TLS implementations. In *IEEE SP*, pages 435–452, 2019. 1
- [66] Stephen Röttger and Artur Janc. A Spectre proof-of-concept for a Spectre-proof web. <https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html>, 2021. 2, 14
- [67] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: rogue in-flight data load. In *IEEE SP*, pages 88–105, 2019. 2, 12, 14
- [68] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. SGAXe: How SGX fails in practice. <https://sgaxeattack.com/>, 2020. 2
- [69] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking data on Intel CPUs via cache evictions. In *IEEE SP*, pages 339–354, 2021. 2, 14
- [70] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript. In *Financial Cryptography*, pages 247–267, 2017. 2, 13
- [71] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*, pages 753–768, 2019. 2, 12, 14
- [72] Martin Schwarzl, Pietro Borrello, Andreas Kogler, Kenton Varda, Thomas Schuster, Daniel Gruss, and Michael Schwarz. Dynamic process isolation. arXiv/2110.04751, 2021. 2
- [73] Aria Shahverdi, Mahammad Shirinov, and Dana Dachman-Soled. Database reconstruction from noisy volumes: A cache side-channel attack on SQLite. In *USENIX Security*, pages 1019–1035, 2021. 1
- [74] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *USENIX Security*, pages 639–656, 2019. 2, 13
- [75] Anatoly Shusterman, Ayush Agarwal, Sioli O’Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. Prime+Probe 1, JavaScript 0: Overcoming browser-based side-channel defenses. In *USENIX Security*, pages 2863–2880, 2021. 2, 13
- [76] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU register state using microarchitectural side-channels. arXiv/1806.07480, 2018. 2
- [77] Charles E Stroud and Ahmed E Barbour. Design for

- testability and test generation for static redundancy system level fault-tolerant circuits. In *Proceedings, 'Meeting the Tests of Time', International Test Conference*, pages 812–818. IEEE, 1989. 7
- [78] Jakub Szefer. Survey of microarchitectural side and covert channels, attacks, and defenses. *J. Hardw. Syst. Secur.*, 3(3):219–234, 2019. 1
- [79] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1):25–33, 1967. 3
- [80] Yukiyasu Tsunoo, Etsuko Tsujihara, Kazuhiko Mine-matsu, and Hiroshi Hiyachui. Cryptanalysis of block ciphers implemented on computers with cache. In *ISITA*, 2002. 1
- [81] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, pages 991–1008, 2018. 2
- [82] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: hijacking transient execution through microarchitectural load value injection. In *IEEE SP*, pages 54–72, 2020. 2
- [83] Stephan van Schaik, Kaveh Razavi, Ben Gras, Herbert Bos, and Cristiano Giuffrida. RevAnC: A framework for reverse engineering hardware page table caches. In *EUROSEC*, pages 3:1–3:6, 2017. 2
- [84] Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious management unit: Why stopping cache attacks in software is harder than you think. In *USENIX Security*, pages 937–954, 2018. 1
- [85] Bhanu Chandra Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in xen. In *CCSW*, pages 41–46, 2011. 2
- [86] Pepe Vila, Boris Köpf, and José F. Morales. Theory and practice of finding eviction sets. In *IEEE SP*, pages 39–54, 2019. 2, 10
- [87] Krishnaswamy Viswanathan. Disclosure of hardware prefetcher control on some intel® processors. <https://www.intel.com/content/www/us/en/develop/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.html>, 2014. 7
- [88] John Von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In *Automata Studies.(AM-34), Volume 34*, pages 43–98. Princeton University Press, 2016. 7
- [89] Luke Wagner. Mitigations landing for new class of timing attack. <https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/>, 2018. Accessed: 2022-01-25. 2, 13
- [90] Mike J. Wilkinson, Claudia Szabo, Caroline S. Ford, Yuval Yarom, Adam E. Croxford, Amanda Camp, and Paul Gooding. Replacing Sanger with Next Generation Sequencing to improve coverage and quality of reference DNA barcodes for plants. *Scientific Reports*, 7(1):46040, 2017. doi: 10.1038/srep46040. 12
- [91] John C. Wray. An analysis of covert timing channels. In *IEEE SP*, pages 2–7, 1991. doi: 10.1109/RISP.1991.130767. 13
- [92] Wenjie Xiong and Jakub Szefer. Survey of transient execution attacks and their mitigations. *ACM Comput. Surv.*, 54(3):54:1–54:36, 2021. 1
- [93] Wenjie Xiong, Stefan Katzenbeisser, and Jakub Szefer. Leaking information through cache LRU states in commercial processors and secure caches. *IEEE Trans. Computers*, 70(4):511–523, 2021. 1
- [94] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy H. Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *IEEE SP*, pages 888–904, 2019. 1, 2
- [95] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *USENIX Security*, pages 2003–2020, 2020. 1
- [96] Yuval Yarom and Katrina Falkner. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, pages 719–732, 2014. 1, 2, 12
- [97] Tao Zhang, Kenneth Koltermann, and Dmitry Evtushkin. Exploring branch predictors for constructing transient execution Trojans. In *ASPLOS*, pages 667–682, 2020. 1
- [98] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *CCS*, pages 305–316, 2012. 2, 12