The ghost *is* the machine: Weird machines in transient execution

Ping-Lun Wang
Carnegie Mellon University
pinglunw@andrew.cmu.edu

Fraser Brown
Carnegie Mellon University
fraserb@andrew.cmu.edu

Riad S. Wahby
Carnegie Mellon University
riad@cmu.edu

***Abstract*—Microarchitectural attacks typically exploit some form of transient execution to steal sensitive data. More recently, though, a new class of attacks has used transient execution *to (covertly) compute*: Wampler et al. use Spectre primitives to obfuscate control flow, and Evtyushkin et al. construct "weird" logic gates that use Intel's TSX to compute entirely using *microarchitectural* side effects (i.e., in a cache side channel). This paper generalizes weird gate constructions beyond TSX and shows how to build such gates using any transient execution primitive. We build logic gates using exceptions, the branch predictor, and the branch target buffer, and we design a `NOT` gate that appears to perform roughly one order of magnitude[1] better than the prior state of the art. These constructions work on AMD, Intel, and ARM machines with $\approx$95-99% accuracy; a million `AND` gate executions take from half a second (when built with TSX) to four and a half seconds (when built with the branch target buffer). Our results indicate that weird gates are more generally applicable than previously known and may become more widely used, e.g., for malware obfuscation.**

I. Introduction

The spiraling complexity of modern CPUs has driven a steady drumbeat of *microarchitectural* vulnerabilities, including Spectre, Meltdown, ZombieLoad, and more [1], [2], [3], [4], [5]. These vulnerabilities stem from a mismatch between the guarantees provided by the architectural abstraction (the instruction set architecture or ISA) and the observable behavior of the microarchitecture. Put another way: modern CPUs' ISAs are leaky abstractions, and the implementation details visible through the cracks can be exploited by attackers.

At a high level, such vulnerabilities take advantage of the fact that modern ISAs distinguish between architectural state (e.g., the contents of registers) and microarchitectural state (e.g., the contents of cache). Importantly, these ISAs make (at best) weak guarantees about the consistency of microarchitectural state with the CPU's correct execution. For example, when a CPU speculatively executes, the state of its cache will often reflect operations that were executed and later rolled back; in contrast, registers and other architectural state are not allowed to reflect rolled-back operations. To date, most microarchitectural vulnerabilities give attackers the power to observe data that, according to the architectural abstraction, should be inaccessible. This is extremely serious because even "single-user" computing devices must enforce separation between protection domains (e.g., user vs. kernel, process boundaries, etc.), which these vulnerabilities can sidestep.

Recently, however, a new class of issues has emerged that leverages microarchitectural vulnerabilities for covert *computation*, not just data exfiltration. ExSpectre [6], for example, makes static analysis of malware difficult by exploiting the mismatch between the ISA's stated behavior, e.g., "untaken branches are not executed," and the CPU's actual behavior, e.g., "untaken branches may be executed transiently depending on the state of the branch predictor." In particular, ExSpectre observes that because transient execution is allowed to modify microarchitectural state, transient execution of an "untaken" branch can have observable (microarchitectural) side effects, and can thus be used to execute malicious code *even if that code should never run according to the ISA specification.*

Another recent example of this approach is the work of Evtyushkin et al. on *microarchitectural weird machines* (μWMs) [7]—code gadgets that perform computation using only microarchitectural side effects. The authors use μWMs to implement logic gates operating entirely on microarchitectural state, without (say) using the processor's ALU. Like ExSpectre, μWMs can be used to thwart software analysis techniques: predicting the result of running a program involving μWMs requires an analysis tool to model the CPU's operation at the microarchitectural level, raising the bar to detection.

Evtyushkin et al. evaluate a proof-of-concept implementation based on Intel's Transaction Synchronization Extensions (TSX), which provide a convenient and easily controlled way to induce transient execution. (The authors also briefly mention that other methods may be possible, but do not explore further.) Unfortunately, relying on TSX to construct μWMs has at least two significant downsides: first, it is trivial to flag programs using this approach simply by scanning for use of TSX. Second, TSX support is both rare and shrinking [8], [9], [10], dramatically limiting the applicability of this approach. This leaves open our animating question:

Do μWMs generalize to x86_64 processors without TSX support, and to other processor architectures?

We answer this question in the affirmative by systematically studying the construction of μWMs based on several known methods of inducing transient execution. We give new μWM

[1] The data in the original paper reports `XOR` execution speed and `XOR` executions per second that do not agree with one another. Taking the execution speed at face value, our construction is two orders of magnitude faster; instead, we calculate a faster execution speed for their reported executions per second, and our approach only yields an order of magnitude improvement.

designs, including a NOT gate primitive that appears to be much higher performance than prior designs: using our NOT gate, we build an XOR gate that is roughly two orders of magnitude faster (in relative terms) than the design evaluated by prior work.[2] Since logical inversion is required to implement universal boolean functionality, our work improves the performance of essentially every μWM computation of interest. In sum, we find that μWMs can be readily and reliably constructed using a variety of trigger mechanisms across a wide range of processor architectures ($\approx$95–99% accuracy on AMD, Intel, and ARM processors). They also execute reasonably quickly: a million AND gate executions take from half a second (when built with TSX) to four and a half seconds (when built with the branch target buffer). These facts combined indicate that μWMs could easily become a widespread technique for camouflaging malware. To encourage the community to further refine these techniques and develop defenses, all of our code is available under an open-source license.[3]

II. Background and related work

In this section, we give background on transient execution and weird machines, and discuss related work. This sets the stage for our weird gate constructions in Section III.

A. Transient execution and attacks

We use the generic term *transient execution* to mean the CPU's execution of any instructions whose side effects are allowed to modify only microarchitectural state and are not reflected in architectural state (e.g., registers and memory). Most of the time, the microarchitectural state we consider is cache residency information—whether or not a given memory location is currently cached. Observing microarchitectural state is often an involved process. Extracting cache residency information, for example, requires measuring the latency to read a given memory location (low latency means the memory location is in cache, high latency means not) or exploiting processors' vulnerabilities to leak microarchitectural states [11]. We treat accessing microarchitectural state abstractly: in the case of cache residency, observations implicitly involve such timing measurements, but we refer to them as read operations without specifying the mechanism.

Modern CPUs have several mechanisms that induce transient execution. Essentially all of them have been used to exfiltrate private data (e.g., to read across process boundaries or to read kernel memory from user mode). Meltdown [2] relies on out-of-order execution, which transiently executes instructions after an exception. Spectre [1] variant 1 mistrains the branch predictor, causing transient execution of an untaken branch, while variant 2 relies on a similar mechanism involving the branch target buffer. ZombieLoad [5] uses Intel's Transaction Synchronization Extensions (TSX) to induce transient execution by causing a transaction to abort. The high-level blueprint for exfiltrating private data using these mechanisms is to: (1) induce the CPU to transiently execute computations that cause the targeted data to be reflected in microarchitectural state, then (2) read the microarchitectural state once the transient execution finishes [12], [13], [14].

ExSpectre [6] also uses transient execution, but for the purpose of obfuscating a program's execution. At a high level, it works by "hiding" instructions in untaken branches of a program, then causing that code to be transiently executed. Such code reads its inputs from architectural state—but since it runs transiently, its outputs only appear in *micro*architectural state. This is a limitation because assembling complex malware from small transient code blocks requires moving each block's output from microarchitectural to architectural state before the next block can execute. Microarchitectural weird machines, discussed in the next section, address this issue by computing entirely over microarchitectural state.

Our work builds on all of the above techniques, using transient execution to perform covert computations via *transient weird gates*, which we describe in more detail in Section III.

B. Weird Machines

A weird machine (WM) is a model of a system's unintentional behavior, usually in response to adversarial input [15]. In particular, weird machines capture functionality that is not part of a system's *intended* behavior, but *is* part of the system's *actual* behavior. In this model, attacks on a system that cause it to deviate from intended behavior are understood as programs running on a "weird system" that extends the underlying system with functionality resulting from some detail of the system's implementation.

Weird machines have been used to describe attacks [16], formally analyze exploitability [15], and perform obfuscated computation [7], [17]. There are several weird machines that are based on operating systems mechanisms [17], [18] and x86 instructions [19]. Their execution is visible in the CPU's architectural state, meaning they can be detected with techniques like static and dynamic analysis.

In contrast, Evtyushkin et al. [7] construct weird machines based on microarchitectural side effects that are never visible in the machine's architectural state. Using techniques similar to ZombieLoad [5], these weird machines execute after a fault occurs inside a TSX transaction, storing the computation's results as cache residency information. In particular, the authors design TSX-based AND, OR, XOR, and ASSIGN gates, which we collectively refer to as the "TSX WM". In Section III, we show the TSX WM alongside our constructions, which build on different transient execution mechanisms.

We note that Evtyushkin et al. also describe gates that do not use TSX. These non-TSX gates are considerably less useful than the TSX gates, however, because the non-TSX gates use the branch predictor as a microarchitectural state input. As the authors discuss [7, §4], this means that sequentially composing their non-TSX designs requires repeatedly moving data from architectural to microarchitectural state and back.

[2] That work did not present an implementation or results for its NOT gate design, so we cannot compare directly, but it seems likely that the NOT gate is the bottleneck in the XOR design.

[3] https://github.com/joeywang4/Transient-Weird-Machine

This dramatically reduces both performance and obfuscation potential. For this reason, we do not consider these non-TSX designs to be full-fledged μWMs. In the rest of this paper, we restrict our attention to gates that, like the TSX WM, can be cascaded without round-trips to architectural state.

While the TSX WM makes standard dynamic analysis infeasible because its execution is not visible in architectural state, it still suffers from two drawbacks (Section I):

1) **Ease of detection.** Even an extremely simple static analysis can look for uses of (otherwise very unusual) TSX instructions.
2) **Limited applicability.** Only Intel machines support TSX; Intel now defaults to disabling TSX [10], and most operating systems provide options to disable TSX [8], [9] (in order to protect against ZombieLoad [5]).

This paper addresses these issues by constructing weird machines from mode widely available and less readily detected transient execution primitives.

III. Design

We propose the Transient WM, a weird machine that generalizes the TSX WM and is based on transient execution primitives (e.g., raising exceptions). A Transient WM is composed of three ingredients:

1) **A transient execution primitive**: Similar to the TSX WM, our weird machine begins execution when the CPU is in transient execution mode—when the execution results only affect the microarchitectural states but not the architectural states (i.e., during speculative execution or out-of-order execution). The Transient WM generalizes the TSX WM to any transient execution primitive, including raising exceptions [2], mis-training the branch predictor (BP) [1], and mis-training the branch target buffer (BTB) for indirect jumps [1].
2) **A microarchitectural side channel**: Since any architectural state changes will be discarded after the transient execution, our WM needs a microarchitectural side channel to transmit the (transient) computation results. While there exist different side channels [7], we use the cache side channel as it provides larger state storage than others; in other words, using the cache allows the Transient WM to have more variables to compute and store values.
3) **A weird gate** that computes on side channel data using a transient execution primitive and transmits back a result via the side channel.

In this section, we give intuition for how to construct transient weird gates using transient execution primitives and side channels. Gate inputs and outputs are booleans represented by whether or not a given variable is in cache. For example, an input may be one if `X[0]` is in cache and zero if it is not. Our gates compute using transient execution. The trick is to use gate inputs to adjust the time it takes to fetch the output into cache; if the output should be one, for example, the inputs must ensure that the output variable can be fetched into cache before the transient execution window ends.

The next sections walk through our gate constructions in more detail.

A. *Transient weird gate intuition: the `assign` gate*

The simplest weird gate is `assign`, which assigns the value of an input variable to an output variable. Listing 1 shows four different `assign` constructions using different transient execution primitives.

```
INIT:
    X[0] = 0;
    SIGFPE {
        goto EXIT;
    }

clflush(Y);
tmp /= 0;
Y[X[0]] = 0;
EXIT:
    ...
```

(a) Exceptions

```
INIT:
    X[0] = 0;
    ptr[0] = 4096;

clflush(Y);
for (i = 0; i <= training; i++) {
    mask = 0 - (i == training);
    to_X = (X - ptr) & mask;
    if (to_X == 0) {
        Y[ptr[to_X]] = 0;
    }
}
```

(b) Branch predictor

```
INIT:
    X[0] = 0;
    Z[0] = 4096;
    safe(I) {}
    gate(I) { Y[I[0]] = 0; }
    vicitim(I) { asm("CALL ptr"); }

clflush(Y);
ptr = gate;
for (i = 0; i < training; i++)
    vicitim(Z);
ptr = safe;
victim(X);
```

(c) Branch target buffer

```
INIT:
    X[0] = 0;

clflush(Y);
TSX {
    tmp /= 0;
    Y[X[0]] = 0;
}
```

(d) TSX

Listing 1: The `assign` gate with different transient execution primitives.

a) Exceptions: Listing 1a shows the construction of the `assign` gate using exceptions as the transient execution primitive. The gate implements input and output values using the cache side channel (as do all gates in this paper).

The input to `assign` is a boolean value represented by whether the variable `X[0]` is in cache: the input is one when `X[0]` is in cache and zero when `X[0]` is not in cache. Note, however, that the value of `X[0]` is always zero in architectural state, i.e., main memory. We use `Y[0]` as the gate output: this boolean output is one when `Y[0]` is in cache and zero when it is not. Note, also, that `Y[X[0]]` (line 9 of Listing 1a) is equivalent to `Y[0]`, because of the value of `X[0]`.

First, the code in Listing 1a registers a signal handler for fatal arithmetic errors (`SIGFPE`). This signal handler (line three) jumps to the end of the assign gate (the `EXIT` label) when the CPU encounters such an exception. On line eight, the CPU encounters a fatal arithmetic error in the form of a divide-by-zero, and thus follows the signal handler's orders—except that transient execution continues for a while before the signal handler's effect is committed.

In transient execution mode, the processor encounters line nine, the assignment of `Y[X[0]]` (which is equivalent to `Y[0]`, our output variable). If the input variable `X[0]` is in cache, the CPU can quickly resolve the address of the output variable `Y[0]` and fetch it into cache before the divide-by-zero has committed; if `X[0]` is not in cache, however, the CPU only has time to fetch `X[0]`—and `Y[0]` does not end up in cache. Therefore, `assign` assigns: if the input variable `X[0]` is in cache, so is the output `Y[0]`; if the input variable is not in cache at the start of the computation, neither is the output variable at the end of the computation.

b) Branch predictor and branch target buffer: Similar to Spectre [1], we can construct an `assign` gate by mis-training the branch target buffer (Listing 1c) or the branch predictor (Listing 1b). Listing 1b's `assign` gate first mis-trains the branch predictor (line six): a `for` loop repeatedly executes an `if` statement (line nine) and satisfies its condition (`to_X == 0`)—except on the last iteration. On that last iteration, the branch predictor mis-predicts because of its training, and still executes the `if` block in transient execution mode. The transiently executed code in the `if` block reduces to `Y[X[0]] = 0`, which performs the assignment as in the prior gate.

Listing 1c shows how to mis-train the BTB to construct an `assign` gate. The first `for` loop (line ten) trains the BTB to assume that the `victim` function will call the `gate` function. After the training loop, the `victim` function executes again, this time calling the (useless) safe function—but, thanks to the BTB, the CPU still transiently executes the `gate` function. This function uses our standard `Y[X[0]]` operation to perform the assignment. Therefore—just as in the exception-based `assign`, because of the limited size of the transient execution window—`Y[0]` ends up in cache only if `X[0]` is in cache.

c) TSX: Listing 1d shows how to replace the exception or mis-training transient execution primitive with a TSX block; this is how Evtyushkin et al. [7] construct their gates. The input and output variables to the `assign` gate stay the same, represented by whether `X[0]` and `Y[0]`, respectively, are in cache. When the gate in Listing 1d executes, the CPU clearly encounters a divide-by-zero exception (line six), and so eventually exits the TSX transaction. Before the exit caused by the divide-by-zero has fully committed, however, the processor continues to execute in transient execution mode—and encounters line seven, the assignment of `Y[0]`. Just as in the other gates, this line pulls `Y[0]` into cache only if `X[0]` is already in cache.

B. Generalizing from the `assign` gate

To calculate an arbitrary Boolean function, we need more than an `assign` gate. This section discusses how to construct `AND`, `OR`, `NOT`, `XOR`, and `MUX` gates.

a) `AND` and `OR` gates: The `assign` gate only contains one line of code that actually performs the transient computation: `Y[X[0]] = 0`, which pulls `Y[0]` into cache only if `X[0]` is in cache. By repeating this pattern, we can create `OR` gates (Listing 2) and `AND` gates (Listing 3). Both the `OR` gate and the `AND` gate work similarly to an `assign` gate: the `OR` gate assigns the two input variables to the same output variable, while the `AND` gate assigns the value one to the output variable only when both the input variables are in cache (so that the address of the output variable can be quickly resolved). We can even repeat or concatenate the transient computation more than twice to generate a multiple-input `OR` or `AND` gate. For example, repeating line nine of the Listing 1a n times yields an n-input `OR` gate.

```
INIT:
    X[0] = Y[0] = 0;
    SIGFPE {
        goto EXIT;
    }

clflush(Z);
tmp /= 0;
Z[X[0]] = 0;
Z[Y[0]] = 0;
EXIT:
    ...
```

Listing 2: An `OR` gate construction using exceptions.

```
INIT:
    X[0] = Y[0] = 0;
    SIGFPE {
        goto EXIT;
    }

clflush(Z);
tmp /= 0;
Z[Y[X[0]]] = 0;
EXIT:
    ...
```

Listing 3: An `AND` gate construction using exceptions.

b) NOT gate: Building a NOT gate is less straightforward. A NOT gate only fetches the output variable into cache when the input variable is not in cache, which behaves like an *inverse* of an assign gate, *and we cannot simply extend an assign gate to build a NOT gate*. While TSX WM supports a NOT gate, the implementation is not available, and we instead construct a NOT gate independently.

Our NOT gate construction is inspired by the fact that *the complexity of an instruction that leads to transient execution can affect the capacity of transient execution* [6]. In other words, when an instruction causing the CPU to enter the transient execution mode takes more time to execute, then the CPU can execute more instructions in transient execution mode. If we can make an instruction more complex when the input variable is not in cache, then we can keep the CPU in transient execution mode longer—allowing it to fetch the output variable into cache.

Listing 4 shows our NOT gate construction (using exceptions as the transient execution primitive). The input and output variables are X[0] and Z[0], respectively, and Y[0] is an auxiliary variable. By replacing the line from earlier, tmp /= 0, with tmp /= X[0], we can extend the execution time needed to raise a divide by zero exception. In other words, when X[0] is in cache, the CPU can quickly retrieve the value of X[0] (which is set to zero) and raise an exception. Since Y[0] is not in cache, it is not possible to fetch Z[0] into cache before the transient execution ends. Thus, when X[0] is in cache before the gate executes, Z[0] ends up *not* in cache after the gate executes.

On the other hand, when X[0] is not in cache before the gate executes, the CPU needs to wait until the value of X[0] is fetched from memory and until it can raise an exception. This allows the output variable Z[0] to be fetched into cache during the (longer) transient execution window. Thus, if X[0] is not in cache before the gate executes, Z[0] ends up in cache after the gate executes.

```
INIT:
    X[0] = Y[0] = 0;
    SIGFPE {
        goto EXIT;
    }

clflush(Y);
clflush(Z);

tmp /= X[0];
Z[Y[0]] = 0;
EXIT:
    ...
```

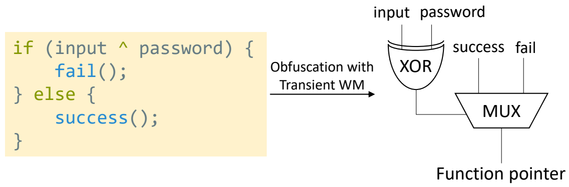
Listing 4: A NOT gate construction using exceptions.

c) XOR and MUX gates: With the AND, OR, and NOT gates in place, we can compose a more complex logic gate. Because they're helpful for program obfuscation, we implement both the XOR gate and the MUX gate. The two gates are constructed as follows:

- **XOR**: AND(OR(X, Y), NOT(AND(X, Y)))
- **MUX**: OR(AND(X, NOT(Z)), AND(Y, Z))

X, Y, and Z are the input variables, while the AND, OR, and NOT are the gates described in previous sections. In Listings 2 and 3, we show how to construct gates by repeating or concatenating existing gates. The XOR and the MUX gates can be constructed using similar methods.[4]

Using XOR and MUX, we can obfuscate control flow and cryptographic operations. Figure 1 shows a simplified example, using a Transient WM to obfuscate password checking with the XOR gate and the MUX gate. By constructing larger circuits using these gates, it is also possible to obfuscate a more complicated cryptographic algorithm; Evtyushkin et al., for example, implement SHA-1 [7].



```
if (input ^ password) {
    fail();
} else {
    success();
}
```

Fig. 1: An example of obfuscating a password checking program with Transient WM.

IV. Evaluation

Our evaluation answers the following questions:

1) How do the speed and accuracy of weird gate designs based on different transient execution mechanisms compare to one another and to prior work?
2) How well do transient weird gate designs generalize across different processor architectures?

In sum, we find that exception-based weird gates enjoy the best accuracy (≈99%), but are slower (≈5×) than TSX-based weird gates. We also measure the speed and accuracy of exception-based weird gates on three different processors and discover that all of them have similar accuracy, while the speed depends strongly on the single-threaded performance of the processor.

A. Implementation and experimental setup

We implement our Transient WM using exceptions, TSX, BP, and BTB as the transient execution primitives (for TSX, BP, and BTB, we use only AND gates in this evaluation). Our implementation is written in 1,379 lines of C/C++ and x86_64 and ARM assembly. Our BP, BTB, and ARM implementations are based on existing Meltdown and Spectre attack implementations [20], [21], [22], and we reuse 198 lines of code from them.

We evaluate on x86_64 machines with CPUs from AMD and Intel, and on an ARM machine. All these machines are available on the AWS EC2 cloud platform. Our testbeds have the following specifications:

[4] https://github.com/joeywang4/Transient-Weird-Machine/blob/main/exceptions/gates/compose.cpp

- AMD EPYC 7R13 Processor (AWS EC2 c6a.large), Ubuntu 22.04
- Intel Xeon CPU E7-8880 v3 @ 2.30GHz (AWS EC2 x1e.xlarge), Ubuntu 22.04
- ARM-based AWS Graviton Processors (AWS EC2 a1.large), Ubuntu 22.04

For the different processors, we calibrate the construction of the XOR and the MUX gates to adjust for different capacities of transient execution; we discuss in Section V.

B. Speed and accuracy of different transient weird gates

To compare speed and accuracy across transient execution primitives, we implement our AND gate using TSX, exceptions, the branch predictor (BP), and the branch target buffer (BTB). We evaluate on our Intel E7-8880 v3 testbed. The branch predictor and branch target buffer versions of our AND gate implementations use 5 and 10 training iterations to mis-train the branch predictor and branch target buffer, respectively. Below, we describe how we calculate speed and accuracy, then discuss results.

a) Measuring speed and accuracy: We run each AND variant one million times with inputs uniformly sampled from $\{0, 1\}$, reading the output from the cache after each iteration by timing the access latency of the output variable. Listing 5 shows our timing code, which is written in x86_64 assembly.

To compute speed, we measure the total execution time of one million iterations. (We note that this is a *pessimistic* estimate of the speed, since it includes the time to set the inputs and read the output in each iteration.) To compute accuracy, we divide the number of correct iterations by the total number of iterations, where an iteration is correct if the AND gate produces the same output as a logical and operation. We repeat this measurement one thousand times to generate the cumulative distribution function of the accuracy, i.e., the portion of the execution that exceeds a certain amount of accuracy, in Figure 2, and we report the median values of the speed and the accuracy in Table I.

```
rdtscp
shl rdx, 32
mov rsi, rdx
or  esi, eax
mov al, [ptr]
rdtscp
shl rdx, 32
or  edx, eax
sub rdx, rsi
mov [clk], rdx
```

Listing 5: The timing function to read from the cache side channel for x86_64 processors. This code calculates the latency to access the ptr variable and output the number of cycles to the clk variable.

b) Comparison of transient execution primitives: Table I shows the speed and accuracy of different implementations of the AND gate on the Intel processor. TSX is the fastest, while the branch target buffer is the slowest. This is because TSX does not rely on any training (which is required for the BP and the BTB) or switching between the kernel and user modes (which is required for handling exceptions).

	TSX	Exception	BP	BTB
Runtime	0.556s	2.628s	2.931s	4.570s
Accuracy	99.56%	99.99%	94.79%	93.52%

TABLE I: The median of the AND gate speed and accuracy when using different transient execution modes (§IV-B). Runtime is the time to execute 1M operations.

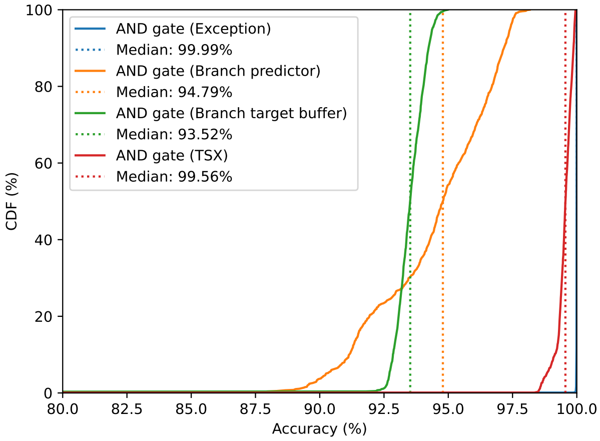


Fig. 2: The cumulative distribution function (CDF) of the accuracy of the AND gate using different transient execution modes.

Exception-based weird gates have the highest accuracy, closely followed by TSX ones; the BP and BTB gates have slightly worse accuracy. According to Figure 2, BP and BTB gates also have higher variation in their accuracy. We believe that the lower and variable accuracy of the BP and the BTB are due to *correct* predictions in spite of our mis-training process, though other effects may also contribute.

C. Portability of exception-based weird gates

This section compares speed and accuracy of our exception-based weird gates across different processor architectures; we also briefly compare with the (non-portable) TSX-based gates given in prior work. We choose exception-based gates because they are portable and because the prior experiment showed that they give good performance; BTB- and BP-based gates are also portable, but we do not evaluate them because they are slower than exception-based gates.

a) Measuring speed and accuracy: We measure speed and accuracy as described in the prior experiment. For the AMD and Intel machines, we use the timing code from Listing 5. Our ARM implementation uses a different timing mechanism because the system register that provides the current CPU cycle is not readable from user space by default. Instead, we use a multi-threaded timer [23]. Listing 6 shows the timing function for the ARM processor.

```
INIT:
    volatile uint64_t counter = 0;
    void* inc_counter(void* a) {
        while(1) {
            counter++;
            asm volatile ("DMB SY");
        }
    }
    pthread_t t;
    pthread_create(
        &t, NULL, inc_counter, NULL);

uint64_t timed_read(uint8_t *addr) {
    uint64_t ns = counter;

    asm volatile (
        "DSB SY\n"
        "LDR X5, [%[ptr]]\n"
        "DSB SY\n"
        : : [ptr] "r" (addr) : "x5"
    );

    return counter - ns;
}
```

Listing 6: The timing function (`timed_read`) to read from the cache side channel for ARM processors.

WG	AMD	Intel	ARM
`AND`	1.801	2.628	32.521
`OR`	1.756 (0.98×)	2.551 (0.97×)	32.506 (1.00×)
`assign`	1.751 (0.97×)	2.637 (1.00×)	32.378 (1.00×)
`NOT`	1.779 (0.99×)	2.741 (1.04×)	32.573 (1.00×)
`XOR`	7.403 (4.11×)	15.648 (5.95×)	130.898 (4.03×)
`MUX`	5.870 (3.26×)	11.975 (4.56×)	131.150 (4.03×)

TABLE II: Weird gate speed across processor architectures (seconds/1M operations) and ratio to `AND` gate speed.

b) Speed comparison across architectures: Table II shows the speed of the Transient WM. When comparing the speed of different gates on the same processor, more complex gates (i.e., `XOR` and `MUX`) are usually three to four times slower than simpler gates (e.g., `AND` and `NOT`). This is because the `XOR` and `MUX` gates are composed of simpler gates like `NOT`. When comparing the speed of the Transient WM across different processors, the AMD CPU has the best performance while the ARM CPU is much slower. This is because the AMD CPU has the best single-threaded performance, while the ARM CPU in our testbed system is optimized for efficiency at the cost of lower performance.

We now briefly compare speed to the prior work's TSX WM, which reports speeds of 0.42 (`assign`), 0.591 (`AND` and `OR`), and 16.6 (`XOR`) seconds per million operations.[5] This comparison is imperfect: we do not have access to that work's gate implementations, nor do we have access to a machine like the one used in that work's evaluation (an Intel i7-6660U running Ubuntu 18.04.4). Moreover, the method used in that work to measure speed is not described in detail.

Nevertheless, we can make some very general observations: first, besides `XOR`, absolute times are very roughly in line with the performance of our portable designs (ignoring `XOR`, TSX WM is ≈3× faster than our AMD implementations). Likewise, the speed of `OR` and `assign` normalized to the speed of `AND` is very close to the same as in our results. The speed of `XOR` is an outlier: whereas our `XOR` implementations are ≈4–5× slower than the corresponding `AND`, the prior `XOR` implementation's performance is ≈28× worse than that work's `AND`. It is unclear why the prior `XOR` gate is so slow; we speculate that this gate includes some error correction mechanism to increase its accuracy at the cost of performance.

WG	Prior Work	AMD	Intel	ARM
`AND`	98.5%	99.96%	99.99%	98.89%
`OR`	97.9%	100.00%	99.99%	99.46%
`assign`	98.5%	99.99%	99.99%	99.42%
`NOT`	-	99.51%	99.99%	99.29%
`XOR`	99.2%	94.36%	95.58%	97.97%
`MUX`	-	98.17%	99.93%	97.64%

TABLE III: Accuracy of weird gates across CPU architectures and compared to prior work (§IV-C). Prior work [7] gives two different accuracy figures for its `XOR` gate: 99.2% (Table 2) and 92.59% (Table 8). The cause for this discrepancy is unclear; we assume the higher accuracy is correct.

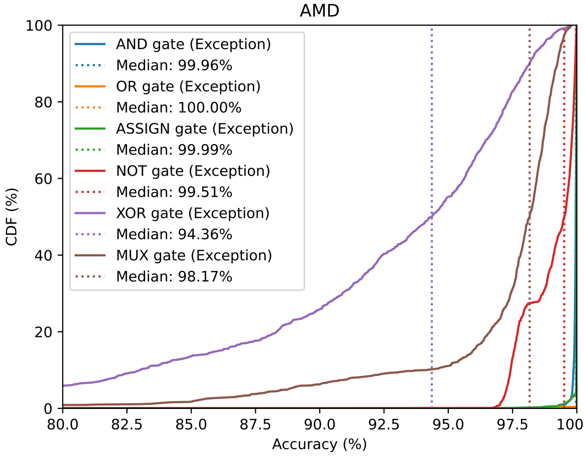


Fig. 3: The CDF of the accuracy of the AMD weird gates.

c) Accuracy comparison results: Table III compares accuracy of the Transient WM across architectures, and with the prior work's TSX WM. Figures 3–5 show the CDFs of the accuracy on the AMD, Intel, and ARM testbeds. For the Transient WM, all the basic gates (`assign`, `AND`, `OR`, `NOT`) have accuracy higher than 99% on all processors. For the `XOR` and `MUX` gates, the accuracy is slightly lower—roughly 95% and 98%, respectively, and their CDFs also have lower slopes and longer tails—indicating that their accuracy is less stable than other gates. This is because these gates combine several other basic gates, and noise in the side channel (e.g., cache collisions generated by other processes) has increasing effect as more gates are combined.

The only TSX weird gate with higher accuracy than an exception weird gate is `XOR`. Unlike our `XOR` gates, which

[5] This is the best-case value for prior work; see Footnote 1.

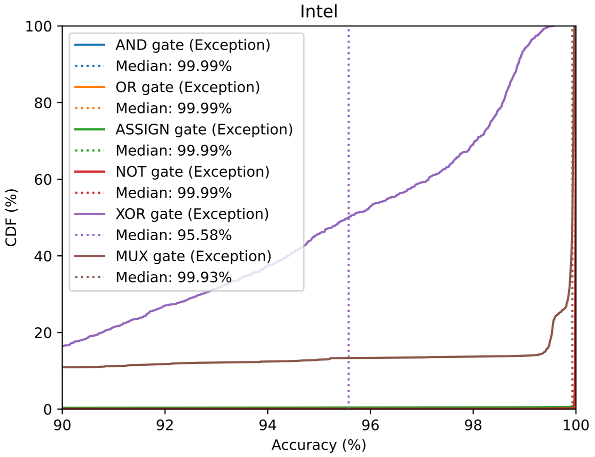


Fig. 4: The CDF of the accuracy of the Intel weird gates.

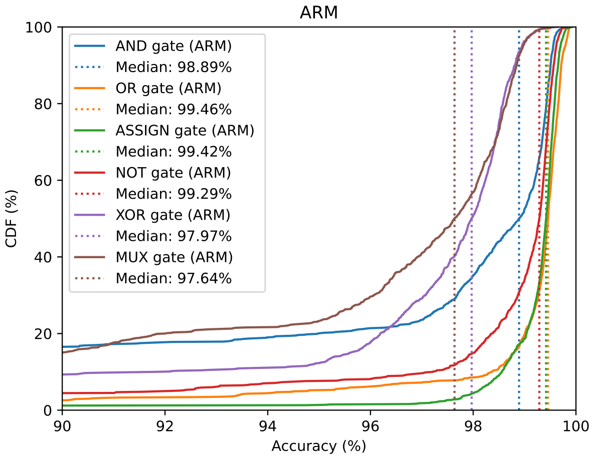


Fig. 5: The CDF of the accuracy of the ARM weird gates.

have *lower* accuracy than basic gates, prior work's XOR gate has *higher* accuracy than its components. As discussed above, this may be because that XOR includes some error correction mechanism that also extends its runtime.

V. Future work and conclusion

In this section, we discuss future research directions for μWM attacks and defenses. Finally, we conclude.

A. Defense against μWMs

While there are no existing defense schemes that focus on detecting or mitigating μWM execution, simple static analysis can detect the existence of a TSX WM, and excessive occurrences of exceptions during dynamic analysis might indicate the use of an exception-based WM. Branch predictor and branch target buffer constructions are more difficult to detect using static and dynamic analysis, but they still require further improvements on their performance and accuracy. Future work can improve these constructions or discover other transient execution primitives to provide better stealthiness, performance, and accuracy.

Protections against microarchitectural attacks may also prevent μWM execution. For example, several hardware-based protections [24], [25] try to block the cache side channel, and there are software-based protections for Spectre and other attacks [26], [27]. Unfortunately, though, hardware changes can be difficult to deploy, and existing software schemes cannot protect programs that use microarchitectural effects on purpose. We believe that μWM detection may be a more feasible short-term research direction. Still, detecting Transient WM is non-trivial, since attackers can mix-and-match several different constructions to try to evade detection (§III).

B. Attack development for μWMs

We find that some of the more complex weird gate constructions can have high variation in their accuracy, especially when the processors are using different microarchitectures (Figure 3–5). To address this, we manually adjust our constructions very slightly to maintain high accuracy across different processors (§IV-A). Avoiding this limitation (e.g., by automatically calibrating weird gates for different execution environments) is an important step towards making μWM-based attacks more widely applicable. More broadly, it would be interesting to understand the applicability of μWM-based constructions to processors that are unlike current mainstream CPU designs (for example, we do not know whether these attacks can be made to work on processors that allow out-of-order execution but are not superscalar).

Another possible research direction is to build a compiler that transforms high-level code into an obfuscated program built from Transient WMs. This compiler could build on existing infrastructure that targets boolean circuits from high-level languages [28], [29], [30], and would reduce the manual effort involved in obfuscation. An important first step is to understand the number of gates that can be executed reliably within the speculation window on a given processor; this will drive the compiler's strategy for chaining gates together.

C. Conclusion

This paper presents Transient Weird Machines, WMs that generalize μWMs to different transient execution primitives. Our Transient WMs apply across different processors, offer greater accuracy than TSX-based WMs, and show better NOT gate performance. Our work suggests that computing with microarchitectural state is a promising and general approach for malware and attack obfuscation.

Acknowledgments

We thank the anonymous reviewers for their helpful feedback. This research was supported by the Ann and Martin McGuinn Graduate Fellowship.

References

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

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, “Meltdown: Reading kernel memory from user space,” in *Usenix Sec*, 2018.

[3] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, “RIDL: Rogue in-flight data load,” in *S&P*, 2019.

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

[5] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, “Zombieload: Cross-privilege-boundary data sampling,” in *CCS*, 2019.

[6] J. Wampler, I. Martiny, and E. Wustrow, “ExSpectre: Hiding malware in speculative execution,” in *NDSS*, 2019.

[7] D. Evtyushkin, T. Benjamin, J. Elwell, J. A. Eitel, A. Sapello, and A. Ghosh, “Computing with time: Microarchitectural weird machines,” in *ASPLOS*, 2021.

[8] Microsoft, “Guidance for disabling Intel® Transactional Synchronization Extensions (Intel® TSX) capability,” 2019. [Online]. Available: https://support.microsoft.com/en-us/topic/guidance-for-disabling-intel-transactional-synchronization-extensions-intel-tsx-capability-0e3a560c-ab73-11d2-12a6-ed316377c99c

[9] P. Gupta and T. Gleixner, “x86/msr: Add the ia32_tsx_ctrl msr,” 2019. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c2955f270a84762343000f103e0640d29c7a96f3

[10] Intel, “Intel® transactional synchronization extensions (Intel® TSX) memory and performance monitoring update for Intel® processors,” 2021. [Online]. Available: https://www.intel.com/content/www/us/en/support/articles/000059422/processors.html

[11] P. Borrello, A. Kogler, M. Schwarzl, M. Lipp, D. Gruss, and M. Schwarz, “ÆPIC Leak: Architecturally leaking uninitialized data from the microarchitecture,” in *Usenix Sec*, 2022.

[12] Y. Yarom and K. Falkner, “FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack,” in *Usenix Sec*, 2014.

[13] D. Gruss, R. Spreitzer, and S. Mangard, “Cache template attacks: Automating attacks on inclusive last-level caches,” in *Usenix Sec*, 2015.

[14] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, “Armageddon: Cache attacks on mobile devices.” in *Usenix Sec*, 2016.

[15] T. Dullien, “Weird machines, exploitability, and provable unexploitability,” *IEEE Transactions on Emerging Topics in Computing*, vol. 8, no. 2, 2017.

[16] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina, “Exploit programming: From buffer overflows to “weird machines” and theory of computation,” *login Usenix Mag.*, vol. 36, 2011.

[17] J. Bangert, S. Bratus, R. Shapiro, and S. W. Smith, “The page-fault weird machine: Lessons in instruction-less computation,” in *WOOT*, 2013.

[18] R. Shapiro, S. Bratus, and S. W. Smith, ““Weird Machines” in ELF: A spotlight on the underappreciated metadata,” in *WOOT*, 2013.

[19] S. Dolan, “mov is Turing-complete,” 2013. [Online]. Available: https://drwho.virtadpt.net/files/mov.pdf

[20] C. Gorgovan, “lgeek/spec_poc_arm: Dump privileged ARM system registers from usermode using variant 3a of meltdown,” 2018. [Online]. Available: https://github.com/lgeek/spec_poc_arm

[21] R. Crosby, Postrediori, E. Rouault, L. Szolnoki, P.-L. Wang, Y. Uchida, W. Hawkins, and huutrung, “crozone/SpectrePoC: Proof of concept code for the Spectre CPU exploit,” 2017. [Online]. Available: https://github.com/crozone/SpectrePoC

[22] R. H. Anton Cao, “Anton-Cao/spectrev2-poc: Proof of concept of Spectre variant 2 vulnerability.” 2020. [Online]. Available: https://github.com/Anton-Cao/spectrev2-poc

[23] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, “Pacman: Attacking ARM pointer authentication with speculative execution,” in *ISCA*, 2022.

[24] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, “InvisiSpec: Making speculative execution invisible in the cache hierarchy,” in *MICRO*, 2018.

[25] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, “Speculative taint tracking (STT): A comprehensive protection for speculatively accessed data,” in *MICRO*, 2019.

[26] T. L. Kernel, “Spectre side channels.” [Online]. Available: https://docs.kernel.org/admin-guide/hw-vuln/spectre.html

[27] Intel, “Retpoline: A branch target injection mitigation,” 2018. [Online]. Available: https://www.intel.com/content/dam/develop/external/us/en/documents/retpoline-a-branch-target-injection-mitigation.pdf

[28] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, “Fairplay–a secure two-party computation system.” in *USENIX Sec*, 2004.

[29] M. Franz, A. Holzer, S. Katzenbeisser, C. Schallhart, and H. Veith, “CBMC-GC: An ANSI C compiler for secure two-party computations,” in *CC*, 2014.

[30] E. Chen, J. Zhu, A. Ozdemir, R. S. Wahby, F. Brown, and W. Zheng, “Silph: A framework for scalable and accurate generation of hybrid MPC protocols,” in *S&P*, 2023.