Transient Architectural Execution: From Weird Gates to Weird Programs

Ping-Lun Wang
Carnegie Mellon University

Fraser Brown
Carnegie Mellon University

Riccardo Paccagnella
Carnegie Mellon University

Eyal Ronen
Tel Aviv University

Riad S. Wahby
Carnegie Mellon University

Yuval Yarom
Ruhr University Bochum

***Abstract*—An emerging body of work has explored the construction of *weird gates*—code segments computing on microarchitectural state not exposed by the instruction set architecture. Weird gates abstract microarchitectural state (e.g., CPU cache residency) as Boolean values and compute logical functions over these values. Researchers have used weird gates in applications like side-channel amplification and malware obfuscation. Indeed, in principle, the computational model of weird gates—a Boolean circuit of bounded size—can perform (bounded) arbitrary computation. In practice, however, this model is less efficient (both asymptotically and concretely) than the standard processor model, which supports conditional execution, indexed memory, and richer data types.**

In this paper, we show how to build weird computation in the processor model rather than the circuit model. The primitive that makes this possible is *transient architectural execution*: transiently loading microarchitectural state into registers, computing on it, and storing the results back into microarchitectural state—without exposing any state architecturally. Transient architectural execution, a generalization of prior weird gates and transient execution attacks, allows us to wield the full computational capability of the processor to operate on microarchitectural state. We also show how to use the state of the branch predictor to emulate wide variables of up to 16 bits. As a result, our *weird programs* are over two orders of magnitude faster than weird gates and can compute functions that are impractical using prior approaches.

1. Introduction

Multiple lines of work have investigated unintended behavior resulting from microarchitectural CPU optimizations. These works have demonstrated alarming behaviors that can, for example, leak sensitive program data [1]–[62], bypass architectural access control mechanisms [63]–[95], and even run essentially arbitrary computations that are (in theory) invisible to the processor's architectural state [96]–[103].

The root cause behind many of these behaviors is *transient execution*, which occurs when a CPU makes a wrong prediction about the results of future instructions, speculatively executes code based on that prediction, and squashes (i.e., undoes) executed instructions after realizing the prediction was wrong [104]. Past works explore several models of emergent behavior enabled by transient execution, as summarized in Figure 1. Transient execution attacks use transient execution to access sensitive data and encode it into microarchitectural state [63]–[95]. ExSpectre enhances this model by supporting transient computation that is hidden from the nominal control flow of the program [96].

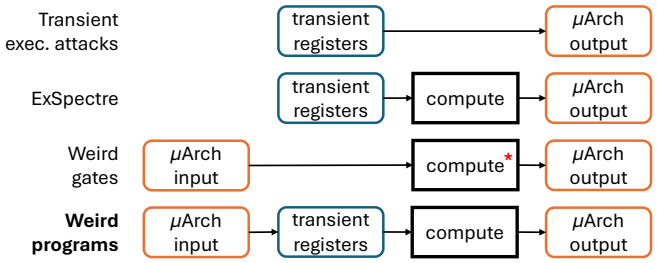

Figure 1. Comparison of weird programs to prior work. Schemes that take input from microarchitectural state can transfer state between transient windows. * indicates limited computational capabilities.

An emerging body of work has used transient execution to construct *weird gates*, which are code gadgets enabling stealthy computation on microarchitectural state [97]–[103]. Weird gates are crafted so that their executions' effect on microarchitectural state depends on their inputs—which are themselves encoded in the microarchitectural state. Consider, for example, a cache-based weird `NOT` gate: its input is '1' if a designated input address is resident in the cache, and otherwise is '0'; its output is likewise encoded as the cache residency of a designated output address; and upon execution, the `NOT` gate sets its output bit to the logical inverse of the input bit. Prior work has shown how to create weird gates that compute universal logic families and how to compose weird gates into *weird circuits* that can (in principle) compute arbitrary logic functions. Security-relevant applications of these circuits include obfuscating malware [97], [101], improving microarchitectural attacks [102], [103], and overcoming side-channel mitigations [99], [100].

However, all these prior works are limited by the inefficient computational substrate weird gates expose—that of Boolean circuits. Specifically, weird gates abstract microarchitectural state (e.g., cache residency) as *weird registers*

that store Boolean values. They then compute a sequence of logical operations on the values in the weird registers, storing the results in other weird registers. As stated above, in principle this enables computing any function. In practice, however, it is severely limited compared to the more familiar computational substrate exposed by processors. First, the circuit model does not support control-flow constructs like `if` statements: within a circuit, all branches must be executed and the result of the taken branch conditionally selected. Second, indexed access to memory (i.e., as in a RAM or ROM) is extremely inefficient in Boolean circuits: each such access entails a subcircuit whose size depends exponentially on the number of bits in the address.

Moreover, the low-level implementation details of weird gates impose significant restrictions to prior work's computational capabilities. First, because processors limit the maximum length of transient execution, current weird-gate designs can compute only relatively small functions (so far, with at most four Boolean inputs [101]). Second, these gates operate on individual bits rather than richer data types. Third, weird registers are read-once, as reading from a weird register destroys its value; keeping data available during execution thus requires complex circuits that carefully manage fanout. Worst of all, the performance of weird gates worsens as circuit complexity increases, which severely limits the ability to work around these restrictions—and thus, the range of practically implementable computation.

Our contribution

This paper demonstrates how to build weird computation in the processor model rather than the circuit model. To this end, we propose a new computational primitive called *weird function*. Similar to weird gates, weird functions also operate on microarchitectural state. Unlike weird gates, however, weird functions execute on the computational substrate exposed by the processor—including multi-bit variables, conditional branching, and indexed memory accesses. As a result, weird functions, which serve as the basic computational primitive of *weird programs*, can execute computations orders of magnitude more efficiently than weird gates.

The primitive that makes this possible is *transient architectural execution*, a generalization of weird gates and transient execution attacks which allows us to wield the full computational capabilities of the CPU to operate on microarchitectural state. Transient architectural execution consists of three steps: At the start of transient execution, a weird function converts microarchitectural state to (transient) architectural state; in other words, it reads values from microarchitectural state into transient registers. Next, a weird function executes a sequence of operations—expressed using *the processor's instruction set*—to compute over the state that was loaded in the first step. Finally, before the CPU squashes the results of transient execution, a weird function reads the results of computation from the transient registers and stores these results in microarchitectural state.

We additionally devise a new type of microarchitectural state storage that leverages the branch target buffer (BTB) to implement *multi-bit weird registers*. In particular, we demonstrate efficient techniques for encoding and decoding values up to 16 bits wide in each predicted branch target in the BTB. Our BTB-based weird registers enable weird functions to load and store much larger amounts of data than previous cache-based weird registers. Additionally, unlike the weird registers of prior works [97], [99], [101], our BTB-based weird registers are not destroyed when they are used; this is possible because our design prevents stray changes to the BTB by carefully exploiting the timing difference between the time a branch is predicted (when the BTB is queried) and when it resolves (when the BTB is updated).

We demonstrate that weird programs and multi-bit weird registers enable μWM computations that are two orders of magnitude faster than weird gates and can compute functions that are impractical using prior approaches. On existing applications like Simon and AES encryption, which are effectively the best case for prior μWMs' Boolean circuit computational model, our weird programs are 1-2 orders of magnitude faster than prior state-of-the-art μWMs [101]. Additionally, our binary-search example (§5.4) highlights the advantage of weird programs' more powerful computational model: the same algorithm would be at least four orders of magnitude slower on existing circuit-based μWMs.

In sum, this work makes the following contributions:

- We propose *transient architectural execution*—a generalization of weird gates and transient execution attacks—enabling weird computation in the processor model as opposed to the circuit model.
- Using transient architectural execution, we construct a new type of μWM, which we call *weird programs*. Weird programs use ISA instructions to compute on microarchitectural state, meaning that they support branches, indexed memory, and rich data types.
- We construct *multi-bit weird registers*, a new type of weird register that uses the BTB to store microarchitectural state. Our multi-bit weird registers allow weird programs to load and store hundreds of bits in one transient window. Compared to prior work, this dramatically expands the range of practically implementable computations by increasing the amount of microarchitectural state that can be accessed and computed upon during a given transient execution.

2. Background and related work

In this section we briefly describe branch target buffers in modern CPUs and give a high-level overview of transient execution. We then discuss the design of microarchitectural weird machines, survey related work, and define relevant terminology that we will use in the rest of the paper.

2.1. Branch target buffer (BTB)

The branch target buffer (BTB) is a component inside the CPU's branch prediction unit, which predicts the targets of indirect branch instructions [105]. While *direct branches*

specify the target address explicitly in the instruction, *indirect branches* specify the branch target via a register or memory location. Resolving the target of an indirect branch sometimes incurs a long delay while the branch target is being computed or if the target refers to an uncached memory location. To prevent the CPU's pipeline from stalling in such cases, Intel CPUs employ both a BTB and an indirect branch predictor (IBP) to predict target addresses [105]. The BTB performs *static predictions* that predict the target based on the *source address* (the address of the branch instruction), while the IBP performs *dynamic predictions* using both source address and the *branching history*, which contains the information of the source and target addresses of recent branches [90]. During execution, the CPU uses both of these mechanisms to predict the targets of indirect branches.

Both the BTB and the IBP have a set-associative structure that records the target of indirect branches. Recent studies [56], [90], [105] have reverse-engineered these structures and showed that the BTB of modern Intel CPUs has more than four thousand entries split between so-called short and long branches, with at least two thousand entries of each type. Short entries store the least significant 10 or 12 bits, depending on the processor model, whereas long entries store the least significant 32 bits. In both cases, the remaining most significant bits are provided by the memory address of the branch. This means that the BTB can in principle store several kilobytes of data. In Section 4 we use this storage to construct multi-bit weird registers.

2.2. Transient execution

Modern CPUs implement various forms of speculative execution, which make predictions about a program's future behavior. Sometimes, these predictions are wrong and result in a phenomenon called *transient execution*, where the CPU executes *transient instructions* that do not affect the architectural state [104]. These transient instructions can nevertheless affect the *microarchitectural* state of the CPU (for example, the contents of cache or the BTB), and this microarchitectural state can be observed via microarchitectural side channels [1]–[51], [54]–[59]. Prior work has exploited this behavior to mount *transient execution attacks*, which use transient instructions to access sensitive data and encode it into microarchitectural state [63]–[95] and to construct *microarchitectural weird machines*, which use transient instructions for stealthy computation [96]–[103].

Like prior work, we use the term *transient window* to mean the period during which a CPU executes transient instructions (e.g., following a misprediction). The *length* of a transient window is the time required for the CPU to realize that the instructions executed within it should be discarded. *Transient registers* are the registers that transient instructions compute on. Once transient execution ends, values in registers do not reflect the transient computation; in a sense, the transient registers no longer exist.

One way to trigger transient execution is through *return address misprediction* [65], [67], which we now describe (looking ahead, this is how we will trigger transient execution in our weird programs; § 3.3). Modern processors have a component called the *return stack buffer* (RSB), which predicts the address to which a function returns. When the processor executes a function call, the address of the instruction after the function call is pushed onto the RSB. The recorded address is the correct return address as long as the function does not modify its return address on the stack. If the return address on the stack is modified, the RSB will mispredict the return address and return to the original (incorrect) return address, and the processor will execute the instructions at the original return address transiently until it realizes that the return address was modified. By controlling the latency of the instructions that compute the modified return address, one can adjust the length of the transient window resulting from a return address misprediction. Our weird programs, for example, set the return address to values in main memory that are not cached, which creates a long cache miss delay and thus a long transient window.

2.3. Microarchitectural weird machines

Microarchitectural weird machines (μWMs) perform computation using the processor's microarchitectural state, without that computation being apparent in the processor's architectural state. μWMs abstract microarchitectural state into *weird registers* and use microarchitectural side effects to compute on these registers. Existing μWM constructions use cache residency information, i.e., whether a memory location is cached or not, to construct 1-bit weird registers. They then use transient execution and microarchitectural side channels to construct *weird gates*, i.e., simple logical functions that operate on weird registers. Like prior work, we define a *weird circuit* as a sequence of weird gates.

Weird registers. In existing μWMs, each weird register is associated with a memory address and holds a Boolean value based on whether that address is in cache. Specifically, we say that a weird register holds a '1' when its associated memory address is present in the cache, and '0' otherwise. In existing μWMs, weird registers are initialized to '0' (uncached); to set a weird register to '1', the μWM must trigger a memory access that loads the corresponding memory address into the cache. Importantly, each cache-based weird register can only be read once because reading it always sets its content to '1', destroying its initial value.

The main memory content of the cache lines associated to weird registers is irrelevant—the data is stored in the cache residency information, not the value in memory. Conventionally, the value in memory is zero. At the end of a weird circuit execution, a weird register can be converted to an architectural register by timing the access latency to its memory address. When the access latency is short (cache hit), its value is '1'; otherwise, its value is '0'.

Weird gates. We now describe existing weird gates using the terminology from Horowitz et al. [102], who describe a weird gate's execution as a race between a *control chain*

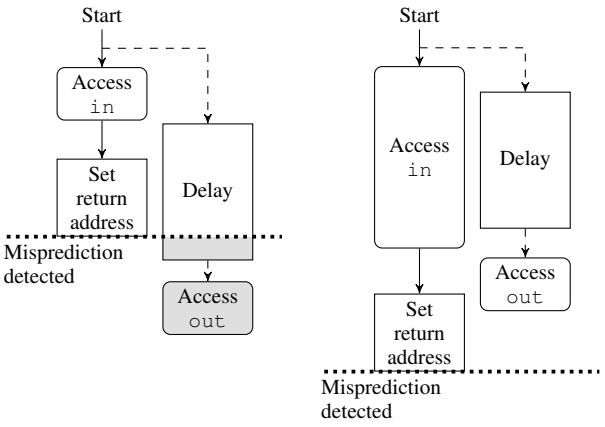

Figure 2. Operation of a `NOT` gate (adapted from [102]), using return address misprediction (§2.2) to trigger transient execution. In the left diagram, the input weird register `*in` is '1' (cached). In the right diagram, it is '0' (uncached). Shaded instructions are never executed. (Not even transiently.).

and one or more *signal chains*. Here, a *chain* is a series of instructions that will create a certain amount of latency during execution; in the usual case, a chain accesses one or more memory locations corresponding to weird registers, creating a variable latency based on whether the memory location is cached (short latency) or not (long latency).

Both the control and signal chains may access memory locations corresponding to weird registers, but they have different purposes. The control chain's latency determines the length of the transient window. In most cases, the last memory access in the signal chain is to the address associated with the weird gate's output register. Thus, if the signal chain runs to completion during the transient window, that memory address will be cached, meaning that the output weird register's value will be '1'. In contrast, if the control chain finishes executing before the signal chain, the output weird register's value will be '0' because transient execution will have ended before the signal chain could set it to '1'.

An example is the `NOT` gate (Figure 2) whose output value is the logical inverse of its input value. In this gate, the control chain accesses the memory location associated with the input weird register; the signal chain accesses the memory location associated with the output weird register. If the input weird register contains a '1', the latency to access the corresponding memory address in the control chain is short, so the signal chain does not run to completion and transient execution ends with the output weird register set to '0'. Otherwise, if the input weird register value is '0', the control chain's latency—and thus the length of the transient window—is long, meaning that the signal chain runs to completion and the output weird register is set to '1'.

Prior work has developed several different types of chains to implement `NOT`, `AND`, `OR`, and other types of common logic gates [97]–[103]. Most recently, Flexo [101] proposed a generalized construction of chains that can implement weird gates with any N-input and 1-output Boolean functions. Flexo's experiments show that many modern processors support any 4-to-1 weird gate, but more than four inputs is not possible due to limited transient window length.

3. From weird circuits to weird programs

In this section, we show how to build weird computation in the processor model rather than the circuit model. We start by discussing the limitations of weird circuits (§3.1). We then introduce *transient architectural execution*, a generalization of transient execution attacks and weird gates that allows us to wield the full computational capabilities of the processor to operate on microarchitectural state (§3.2). Finally, we discuss how transient architectural execution lets us move weird computation from weird circuits to more efficient *weird programs* (§3.3).

3.1. Limitations of weird circuits

While there are many approaches to constructing μWMs, all of them require the programmer to express their program as a sequence of logic gates. More precisely, all prior works expose a computational substrate with the execution model of a Boolean circuit (§2.3). This model is non-uniform—expressing the same computation with different input sizes requires a different circuit—and combinational—the result of the computation depends only on inputs, and not on mutable state. In other words, Boolean circuits don't have indexed memory, branches, dynamically bounded loops, or rich, non-Boolean data types, which makes them an inefficient (and inconvenient) model to program against.

These constraints fundamentally limit scalability of programs in the Boolean circuit model: expressing a high-level program as a circuit requires expensive backflips to avoid constructs like `if` statements, loops with input-dependent bounds, and dynamic array accesses. There are standard methods [106]–[115] to statically convert (bounded) high-level programs into low-level circuits, but this conversion incurs significant execution-time overhead.[1] Branches, for example, require executing *both* the consequent and the alternative and conditionally selecting the result. In other words, the circuit must encode *every possible execution path* through the program. Worse, encoding an indexed memory access entails a conditional load or store to every possible memory location; this means that each memory operation has a time complexity of $\mathcal{O}(2^n)$ for an n-bit address.

For μWMs, this inefficiency snowballs: as circuit size grows, μWM accuracy generally worsens—and the cure for an inaccurate μWM is to re-run at least part of the computation [99], [101], making things *even slower*. Consider a small algorithm that uses conditional branches to dynamically select array elements. Once the programmer has expanded each branch and memory access into conditional

1. Since a wide set of computational substrates—from probabilistic proof systems to multi-party computation protocols to SMT solvers and beyond—expose a circuit-like execution model, compiling high-level programs to circuit-like representations is a well-studied problem.

selects, the small program has become a surprisingly large circuit. The circuit's size is likely to make it slow and inaccurate; if the accuracy is bad enough, error correction will re-run (pieces of) the circuit, slowing things down even more. The computation may take seconds on a μWM where it would be almost instantaneous on a processor (§5).

In contrast to the circuit model, traditional processors expose a much more efficient and expressive computational model, the random-access machine or processor model. The processor model is uniform—a single implementation can support multiple input sizes for the same algorithm—and sequential—the result of the computation depends on both inputs and stored state. A given execution of a program only pays for the branches it actually takes and can dynamically access memory with essentially constant overhead.

In the next sections, we construct μWMs that expose the richer processor computational model and that overcome the limitations of existing circuit-based μWMs.

3.2. Transient architectural execution

We now introduce *transient architectural execution*. We first present a taxonomy to describe existing transient execution-based attack primitives. Then, we use this taxonomy to show how transient architectural execution is a generalization of transient execution attacks and weird gates. Finally, we discuss a technique to copy values from weird registers into transient registers, which allows us to use (transient) ISA instructions to compute on microarchitectural state. Transient architectural execution enables weird computation in the processor model as opposed to the circuit model; we elaborate in Section 3.3.

Taxonomy of attacks that use transient execution. Recall from Section 2.3 that transient execution occurs when a CPU executes instructions that are never exposed to the architectural program state. Starting from Spectre and Meltdown [63], [64], a large body of work has explored the security implications of this phenomenon. Figure 1 shows an abstract view of how these attacks work, focusing on the transient instructions executed within a transient window.

One line of work uses transient execution to mount information disclosure attacks called *transient execution attacks* [63]–[95]. These attacks use transient instructions to load sensitive data into transient registers and later encode this data into microarchitectural state (e.g., cache state, via side channels). This way, an adversary can access data that would normally (i.e., outside of transient execution) be protected by architectural access control mechanisms.

Another line of work uses transient execution to perform stealthy computation. In particular, ExSpectre uses transient instructions to load data into transient registers, perform arbitrary computations on this data, and encode the results into microarchitectural state (using side channels) [96]. Weird gates use *a limited set* of transient instructions to build control and signal chains (§2.3) that compute N-to-1 Boolean functions *directly over microarchitectural state* [97]–[103].

Transient architectural execution. The lines of prior work discussed immediately above and depicted in Figure 1 use, roughly speaking, the same pieces in slightly different ways. It seems natural to ask: is it possible to generalize all three? In particular, can we build a primitive that uses the full power of transient instructions (like ExSpectre) to stealthily compute on purely microarchitectural state (like weird gates)? The answer is yes; transient architectural execution is precisely that primitive.

Transient architectural execution takes place in three steps: (1) copy the computation's inputs, expressed as microarchitectural state, into transient registers; (2) compute on those transient registers using transient instructions; and (3) copy the computation's results from transient registers back to microarchitectural state. This entire process takes place during transient execution, so its effects are not visible architecturally after transient execution ends.

Immediately below, we describe the key building block for step (1), and thus for transient architectural execution.

Copying values from weird registers to transient registers. Performing stealthy computation using transient architectural execution requires copying values between weird registers and transient registers. Weird registers store μWM state and survive across transient windows, but cannot be used as inputs to transient instructions (§2). Thus, to transiently compute on microarchitectural state, it is necessary to move that state from weird registers to transient ones. Transient registers, meanwhile, can serve as operands to transient instructions, but only exist temporarily during a transient window. To save the results of transient architectural execution, it is therefore necessary to copy the computation's outputs from transient registers to weird ones. Encoding data from transient registers into microarchitectural state is a solved problem[2], but the *reverse*—copying data from weird registers into transient ones—is not. In particular, existing techniques to convert microarchitectural state to architectural state rely on accurate time measurements, which are difficult to perform during transient execution [116].

We propose two approaches to copy values from weird registers to transient registers. Conceptually, both our approaches rely on the following idea: using the values of the input weird registers to steer the transient control flow (rather than just influencing the outcome of the race between control and signal chains, as done in weird gates; §2.3). The first approach works with existing cache-based weird registers. However, it does not scale beyond a single input weird register, which motivates our design of a new BTB-based weird register (which we describe in Section 4).

This section describes the simple conversion approach, which copies data from a (cache-based) weird register into a transient one by exploiting branch misprediction. The approach relies on a conditional branch whose transient outcome depends on whether a weird register is '1' or '0'

2. For cache-based weird registers, storing a '1' simply requires performing a transient load to the cache line associated to the weird register (whose value is initially set to '0').

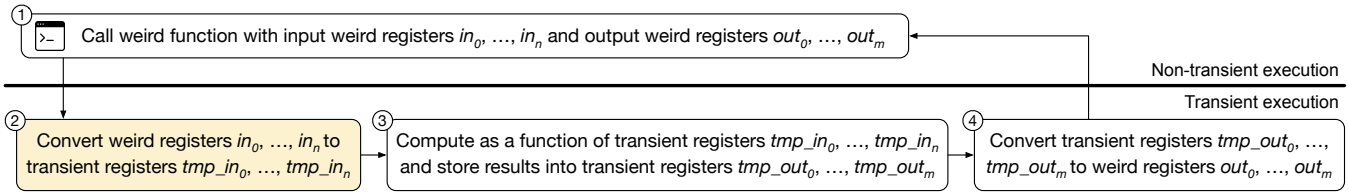

Figure 3. Weird functions use transient architectural execution to compute on weird registers. They copy input values from weird registers to transient ones, compute on these registers, and copy the computation's outputs from transient registers to weird ones. The transient registers used in a weird function only exist while the weird function executes. In contrast, weird registers survive across weird functions. The process of converting values from weird registers to transient registers (shaded in yellow) is a key building block of transient architectural execution and is novel to our work.

```
1  // returns 1 if `addr` is in the cache
2  int cache_to_arch(int *addr) {
3      // addr[0] is set to 0 in the main memory
4      if (addr[0]) {
5          // when `addr` is not in the cache,
6          // BP predicts this direction
7          return 0;
8      } else {
9          // cache hit
10         return 1;
11     }
12 }
```

Listing 1. Example code that copies the value of a cache-based weird register `addr` into a transient register and returns the result. The code executes a conditional branch whose transient outcome depends on whether the input weird register is '0' or '1' (i.e., whether its address is in present the cache). Before transiently executing this function, we train the branch predictor to predict that Line 4's condition evaluates to true.

(i.e., whether a specific cache line is present in the cache). Listing 1 shows a C function that copies data from the weird register associated with `addr`—the register that is '1' when `addr` is in cache and '0' otherwise—into a transient register. Before transiently executing this function, we train the branch predictor to predict that Line 4's condition evaluates to true. However, since the architectural (memory) value stored at address `addr` is 0, the branch always eventually resolves to false. When the function executes transiently:

1) If `addr` is cached (i.e., its associated weird register is '1'), the branch resolves quickly and the CPU speculatively executes the (correct) `else` branch, returning 1.
2) If `addr` is not cached (i.e., its associated weird register is '0'), the branch takes longer to resolve. While waiting for the branch to resolve, the CPU speculatively executes the (predicted) `if` branch, returning 0.

As a result, the transient execution path—and the transient return value of this function—varies depending on the weird register's value. Crucially, this transient return value can be used as input to arbitrary transient instructions.

3.3. Weird programs

We now show how transient architectural execution makes it possible to compute in the processor model rather than the circuit model. To this end, we introduce a new computational primitive called a *weird function*, which uses transient architectural execution to compute on weird registers. We then discuss how weird functions, which serve as the basic computational primitive of *weird programs*, can execute computations more efficiently than weird gates.

```
1  // load the cache state of in0 and in1
2  int tmp_in0 = cache_to_arch(in0);
3  int tmp_in1 = cache_to_arch(in1);
4
5  // fetch out0 if XOR outputs 1
6  if (tmp_in0 ^ tmp_in1) load(out0);
```

Listing 2. Example weird function that performs a 2-input `XOR` operation using transient architectural execution (§3.2). `load` is a simple memory load operation that fetches the output into the cache.

Weird functions. We use transient architectural execution to build a new primitive called a *weird function*. A weird function takes weird registers as input $(in_0, \ldots, in_n)$ and produces weird registers as output $(out_0, \ldots, out_m)$. To do so, it uses the three-step process described above: converting the input weird registers into transient registers $(tmp\_in_0, \ldots, tmp\_in_n)$, performing computations that use these transient registers to produce new output transient registers $(tmp\_out_0, \ldots, tmp\_out_m)$, and finally converting those transient registers into the output weird registers $(out_0, \ldots, out_m)$. This process is illustrated in Figure 3.

Conceptually, the transient registers used in a weird function are like local variables in a regular function—they only exist while the weird function executes. In contrast, weird registers survive across weird functions and can be used to communicate information between weird functions (i.e., between transient windows) and between a weird function and the architectural program (i.e., between transient execution and non-transient execution).

Listing 2 shows the implementation of a weird function that computes a 2-input `XOR` operation. The function first converts the two input weird registers into transient registers, using the function from Listing 1. Then, it computes the `XOR` of these transient registers. Finally, it stores the result of the `XOR` operation into the output weird register.[3]

Weird gates vs. weird functions. Like a weird gate, a weird function takes weird registers as inputs and produces weird registers as outputs; additionally, both weird gates and weird functions execute within a single transient window and are invisible to the processor's architectural state.

3. It does so using a standard load of out_0, which has the side effect of setting the associated weird register to one.

```
// load the cache state of in0, in1, and in2
int tmp_in0 = cache_to_arch(in0);
int tmp_in1 = cache_to_arch(in1);
int tmp_in2 = cache_to_arch(in2);

int result;
// compute either AND or XOR based on tmp_in2
if (tmp_in2) {
    result = tmp_in0 & tmp_in1;
} else {
    result = tmp_in0 ^ tmp_in1;
}

// fetch out0 if result is 1
if (result) load(out0);
```

Listing 3. Example weird function that uses transient architectural execution to compute either an `AND` or a `XOR` operation between in_0 and in_1 based on the value of in_2. Performing this computation using weird gates would require executing both the operations and conditionally selecting the result.

```
// load the cache state of in0 and
// compute the memory address
int *addr = mem + cache_to_arch(in0);

// access a weird register dynamically
int result = cache_to_arch(addr);

// fetch out0 if result is 1
if (result) load(out0);
```

Listing 4. Example weird function that uses transient architectural execution to read data from a dynamically-determined weird register. Performing this computation using weird gates would require reading both weird registers (`addr` and `addr + 1`) and selecting the output using a `MUX` gate.

The key difference between a weird function and a weird gate lies in how they compute. To compute a Boolean operation (e.g., `XOR`), a weird gate cleverly abuses a small set of ISA instructions to create a race condition that sets an output weird register as a function of the input weird registers. In contrast, a weird function computes (on transient registers) using ISA instructions in the normal way: to compute an `XOR`, a weird function uses the `XOR` instruction.

Using ISA instructions as intended allows weird functions to execute in the (efficient) processor model. This means that weird functions can, for example:

1) Use branch instructions. As a result, weird functions do not need to execute every possible program path.
2) Access weird registers dynamically. As a result, weird functions do not need to scan through *the entire address space* every time they want to read a single, dynamically-determined weird register.
3) Leverage complex ISA instructions (e.g., AES-NI, AVX-512) to perform operations that would otherwise require thousands of weird gates.
4) Optionally access read-only architectural memory. This enables, for example, cryptographic applications to access values from lookup tables stored in program memory instead of having to compute these tables for storage in weird registers.[4]

4. Architectural memory is read-only because it cannot be updated transiently. Also, since debuggers can access it directly, architectural memory does not provide the same level of stealthiness as weird registers.

Listing 3 shows an example weird function that computes either an `AND` or a `XOR` operation between in_0 and in_1 based on the value of in_2. The function first converts the input weird registers into transient registers tmp_in_0, tmp_in_1, and tmp_in_2. It then executes a branch that, depending on the value of tmp_in_2, either performs an `AND` or a `XOR` between tmp_in_0 and tmp_in_1. Performing this computation using weird gates would require executing both the `AND` and the `XOR` and conditionally selecting the result.

Listing 4 shows an example weird function that returns the value of one of two possible weird registers depending on the value of the input. When in_0 contains '0', the function returns the value of the weird register associated with address `addr`; when in_0 contains '1', it returns the value of the weird register associated with address `addr + 1`.[5] Finally, the function stores the result into the output weird register out_0. Performing this computation using weird gates would require reading both weird registers (`addr` and `addr + 1`) and selecting the output using a `MUX` gate.

Weird programs. The number of ISA instructions that a weird function can execute depends on how many transient instructions can execute within a transient window. This number is limited by the size of the reorder buffer, which is microarchitecture-dependent. To perform a computation that does not fit within a transient window, weird functions can be used as the basic computational primitive of *weird programs*, which compose several weird functions. In a weird program, the output weird registers of each weird function are directly used as input weird registers to the next weird function. After the execution of a weird program, we read its outputs by converting the values in the output weird registers to architectural values (§2.3). Our Simon implementation (§5) is an example of a weird program: it comprises eight weird functions, each performing 4 encryption rounds.

4. Multi-bit weird registers

In this section we build *multi-bit weird registers*, a core building block for weird programs. We first revisit Section 3.2's approach for copying values from cache-based weird registers to transient registers and show that it does not scale in practice—it can only reliably copy a *single* weird register value per transient window. We then show how to construct multi-bit weird registers that use the state of the BTB for storage. Multi-bit weird registers solve three limitations of cache-based weird registers. First, as the name indicates, each multi-bit weird register can store more than one bit. Second, unlike cache-based weird registers, using the value from a multi-bit weird register does not reset the register's value. Last, we can copy the values of *multiple* multi-bit weird registers into transient registers within a single transient window.

5. Note that this function reads data from a dynamically-determined weird register, *not* from read-only architectural memory.

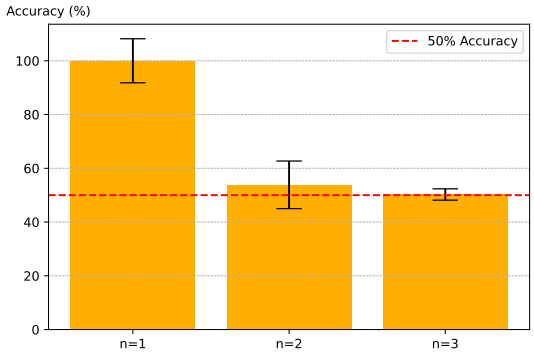

Figure 4. Accuracy of an n-way XOR operation implemented using a weird function and input from cache-based weird registers. While converting a single cache-based weird register gives almost perfect accuracy, the accuracy drops significantly when reading multiple registers.

4.1. Revisiting our approach for converting cache-based weird registers to transient ones

We start by evaluating Listing 1's approach for copying values from cache-based weird registers to transient registers. To this end, we execute the weird function in Listing 2 while varying the number of inputs n. We invoke the function with random 1-bit input values, convert its output weird register into an architectural register, and compare the value of this register with the expected result of the operation. We execute the function one thousand times using random inputs to measure its accuracy, and repeat this process one hundred times to compute the median, mean, and max accuracy. We run the experiment on an Intel Xeon Gold 6430 2.1 GHz (Sapphire Rapids) CPU.

As Figure 4 shows, while converting a single cache-based weird register gives almost perfect accuracy, the accuracy drops significantly when reading multiple registers; at $n = 3$ weird registers, the median accuracy of this approach is about as good as a random guess. We suspect that accuracy drops because we use conditional branches as the register conversion mechanism. In particular, when converting the first weird register from `cache_to_arch`, the outcome of the branch at Line 4 of Listing 1 affects subsequent branch predictions.[6] This means that, for example, the branch prediction on Line 4's branch for the third weird register depends on the values stored in the first two weird registers, making it difficult for us to train that branch.

4.2. BTB-based weird registers

We now demonstrate how to construct weird registers capable of storing sixteen[7] bits using BTB state. Recall from Section 2.1 that each BTB long entry maps a source address to a branch target and stores the least significant 32 bits of the target address. When the processor encounters an indirect branch that matches a BTB entry, the processor predicts that the branch will jump to the stored target address. Eventually, when the branch target resolves, the processor compares the target address to the prediction and reverts the speculative execution in case of a mismatch. Our scheme exploits this branch target prediction mechanism to create weird registers.

```
1  // returns the case that is executed
2  int two_bit_register(int target) {
3      // create an indirect jump using a
4      // switch statement with many cases
5      switch (target) {
6          case 0: return 0;
7          case 1: return 1:
8          case 2: return 2;
9          case 3: return 3;
10     }
11 }
12
13 // set BTB weird register to 3
14 two_bit_register(3);
15
16 // read value from BTB weird register
17 int val = two_bit_register(long_delay());
```

Listing 5. Example code that writes to and reads from a 2-bit BTB-based weird register. Line 14 sets the weird register value to 3 by passing 3 to `two_bit_register`. Line 17 copies the value from the 2-bit weird register to a transient register. Unlike cache-based weird registers, loading the value of a BTB-based weird register does not modify its value, allowing repeated use of the same weird register.

Listing 5 shows how to construct a 2-bit (BTB-based) weird register. The `two_bit_register` procedure (Lines 2–11) implements the weird register. It consists of a switch statement (Line 5) compiled to an indirect branch.[8] The switch statement contains 4 cases, each returning the 2-bit switch case number. This means that the return value of `two_bit_register` corresponds to the transiently executed switch case.

To store a value in the weird register, we use `two_bit_register` with said value as the argument. For example, the code at Line 14 sets the register to 3 by passing 3 to `two_bit_register`. After the indirect branch at Line 5 executes, the CPU sets the BTB prediction for that indirect branch to the matching target (switch case 3). This occurs even if the branch only executes transiently.

To copy the value from the 2-bit weird register to a transient register, we invoke `two_bit_register` with an argument that requires a long delay to resolve—long enough that it cannot be resolved before the end of the transient window. When the CPU reaches Line 5, it consults the BTB and predicts that the branch will jump to switch case 3. Transient execution continues with this prediction until the branch target resolves, which, in our case, never happens. As a result, the transient return value of `two_bit_register` corresponds to the value that was stored on Line 14.

6. Conditional branch prediction depends on the history of taken branches [117].

7. Technically sixteen or more: there's a tradeoff space when choosing how many bits to store. We elaborate later in this section.

8. In practice, compilers only create an indirect branch when there are more than eight switch cases [99]. This example is for illustrative purposes, and we create indirect branches using x86 assembly in our implementation.

In contrast to cache-based weird registers, BTB-based weird registers store values that persist even after being loaded. This is because, during a read operation (e.g., Line 17 of Listing 5), the branch target does not resolve before transient execution ends. As a result, the prediction in the BTB does not change.[9]

Listing 5's construction can be extended to support larger weird registers by increasing the number of switch cases for the indirect branch at Line 5. However, increasing the number of switch cases increases the memory requirements of the weird register. For example, a 16-bit register requires 65,536 switch cases, each taking five bytes[10]—for a total of over 320 KB of memory to implement a single weird register. This limitation makes our approach scale poorly for weird programs that require many weird registers. In the next section, we discuss how we overcome this and other practical limitations.

4.3. Practical considerations

Optimizing BTB-based weird registers. To reduce the overhead of supporting multiple 16-bit weird registers, we separate our BTB-based weird register design into two parts. The first is the *indirect jump* that contains the branch instruction that represents a weird register. The second is the *decode table*, which converts branch targets to transient register values. The decode table is shared between all weird registers, and thus the overhead for each additional register is just one branch instruction.

Listing 6 shows an optimized construction of 16-bit (BTB-based) weird registers. Lines 1–8 are the indirect jumps that define `weird_register_0`, `weird_register_1`, and so on; Lines 10–23 show the decode table, which represents all 16 bit values. To set a weird register to a value, we call the register's indirect branch with the address in the decode table that returns the desired value. For example, Lines 26 and 27 set `weird_register_0` to 2, which is returned from location `target_2` in the decode table. Finally, Lines 31–33 copy a value from `weird_register_1` to a transient register. The code loads a branch target (Line 32) from a memory location that is not in the cache (Line 31), and invokes the indirect jump corresponding to `weird_register_1`. To execute the indirect branch, the processor consults the BTB and jumps speculatively to the stored address—i.e., the value in `weird_register_1`—which is copied to transient register `ax`.

How many multi-bit weird registers can we support? Recall from Section 2.1 that the BTB in modern Intel CPUs contains at least 2,048 long entries. Thus, theoretically, a weird program can support 2,048 multi-bit weird registers, one for each BTB entry. Using so many registers puts them at risk: weird programs execute branches and return statements over the course of normal computation, thus updating the BTB—and potentially overwriting multi-bit weird registers. To reduce this risk, we recommend using no more than half the BTB as register storage. We verify that such a configuration works by setting 1,024 multi-bit weird registers to random 16-bit values, reading the contents of one of the registers, and comparing it to the stored value. We retrieve the correct value 99.5% of the time.

```
1  ; indirect jumps
2  weird_register_0:
3      jmp* rbx
4  weird_register_1:
5      jmp* rbx
6  weird_register_2:
7      jmp* rbx
8  ; ... more entries
9
10 ; decode_table
11 target_0:
12     mov 0, ax
13     ret
14 target_1:
15     mov 1, ax
16     ret
17 target_2:
18     mov 2, ax
19     ret
20 ; ... more target_s
21 target_65535:
22     mov 65535, ax
23     ret
24
25 ; write the value 2 to weird_register_0
26 lea target_2, rbx
27 call weird_register_0
28
29 ; load the value from weird_register_1 to ax
30 ; (during transient execution)
31 clflush delay
32 mov [delay], rbx
33 call weird_register_1
```

Listing 6. Example code that writes to and reads from an optimized 16-bit BTB-based weird register. Lines 26–27 set the first weird register value to 2 by making `weird_register_0` jump to `target_2`. Lines 31–33 copy the value from the 16-bit weird register `weird_register_1` to transient register `ax`. Unlike cache-based weird registers, loading the value of a BTB-based weird register does not modify its value, allowing repeated use of the same weird register. Note that there are 65,536 targets in the decode table, and we omit most of the targets to save space.

How large can a multi-bit weird register be? Recall from Section 2.1 that long entries in Intel BTB store the least significant 32 bits of the target address (which is concatenated with the most significant 32 bits of the source address upon prediction). This means that we could use these BTB entries to create weird registers of at most 32 bits. In practice, however, not all potential target addresses can be used. To see why, recall from above that only target addresses corresponding to entries in the decode table result in correct behavior; any other target address would execute garbage code rather than the desired `mov-ret` sequence. Since each entry in the decode table occupies multiple bytes, a constant fraction (at least half) of 32-bit target addresses

9. The BTB state is only updated when a branch resolves—i.e., when the CPU figures out if the prediction was correct. This happens even if the branch only resolves transiently [90]. In our case, the branch is squashed before it can resolve. Hence, the BTB is not updated.

10. To return a 16-bit value in x86, we need a `mov` instruction to move a constant to `ax` and a `ret` instruction, which occupy five bytes.

does not correspond to a decode-table entry. Moreover, the wider the weird register, the larger the decode table must be, since there must be one entry for each possible register value. We resolve this tension by choosing 16-bit registers; each entry in the resulting decode table is 5 bytes, for a total size of 320 KB. With these parameters, we can store up to 1,024 multi-bit weird registers, each of 16 bits, allowing us to use weird programs that operate on up to 2 KB of data.

Converting a multi-bit weird register to an architectural register. The last remaining issue is converting values from multi-bit weird registers to architectural state after the weird program has finished executing. We cannot use the same approach that works for converting cache-based weird registers to architectural values (§2.3), because multi-bit weird registers store values in BTB state. Instead, we use weird functions to convert multi-bit weird registers to transient registers; then, we encode these transient registers in cache state using standard techniques. We divide each 16-bit value into eight 2-bit chunks, and transmit each of these chunks from transient execution to non-transient execution using a Flush+Reload cache covert channel [14].

Improving accuracy by accounting for asynchronous BTB updates. Recent work reports that the BTB in Intel processors is updated asynchronously: a BTB entry update becomes available for use in subsequent branch predictions several hundreds of cycles after the branch that corresponds to this BTB entry is executed [90]. Therefore, we must add a delay after every update to a BTB weird register. This makes weird functions slightly slower, but it improves accuracy.

5. Evaluation

In this section, we demonstrate that weird programs (§3.2) and multi-bit weird registers (§4) provide orders of magnitude speedups compared to state-of-the-art circuit-based μWMs [101]. We use three applications to highlight different sources of speedup:

1) The Simon [118] block cipher, which is the largest weird circuit implemented by prior work (§5.2). We show that, thanks to the computational capabilities of ISA instructions, this weird program achieves an $11\times$ speedup over the state of the art.
2) The AES [119] block cipher (§5.3). We show that weird programs, unlike prior work, can leverage hardware acceleration (e.g., via AES-NI instructions). On AES, this leads to a $543\times$ speedup over the state of the art.
3) The binary search algorithm (§5.4). Here, we use weird programs' support for branches and indexed memory accesses, which yield an estimated four-order-of-magnitude speedup compared to existing μWMs.

5.1. Implementation and experimental setup

We implement our weird programs using a total of 1,507 lines of C/C++ code and x86_64 assembly, and we evaluate these weird programs on a machine with an Intel Xeon Gold 6430 CPU (Sapphire Rapids) running Ubuntu 22.04.

Executing a weird program. To measure a weird program's accuracy and runtime, we compile a binary that generates random values, writes these values into the weird registers used as inputs to the weird program, executes the weird program, converts the output weird registers into architectural state, and finally compares the output from the weird program to the output from a reference program to check if the computation executed correctly. The binary repeats this process one thousand times and records both the number of correct executions and the total execution time. We then execute this binary one hundred times to measure the median and the standard deviation of the accuracy and runtime. This measurement approach is similar to prior work.

Reproducing the state of the art. We compare our results with the state-of-the-art μWMs from Flexo [101] by running the weird circuits from their artifact [120] on our machine. We use the same methodology as weird programs: we execute each circuit one thousand times using random inputs to measure accuracy and runtime, and repeat this process one hundred times to compute the median and the standard deviation. We verify that our reported results are similar to or better than those reported in the original Flexo paper.

Error correction. We also measure the accuracy and runtime of our weird programs with error correction using two-out-of-three majority voting. To this end, we execute the weird program three times and select the majority value for each output bit as the final output. When reproducing Flexo with error correction, we use the differential encoding-based error detection and correction used in their original paper.

5.2. The Simon block cipher

We start by evaluating the accuracy and runtime of a weird program implementing Simon [118]; this is the largest weird circuit implemented by Flexo [101], the state-of-the-art prior μWM. We use the same 32-bit block, 64-bit key, and 32 rounds of encryption that Flexo does.

Although Simon's hardware-oriented design leads to an efficient implementation in Flexo's weird gates, Flexo still needs 4,322 weird gates (and thus 4,322 transient windows) to encrypt a Simon block. This is because, on our testbed machine, each weird gate can compute a logical function with at most four input bits.[11]

In contrast, a single weird function can use tens to hundreds of ISA instructions, each of which can compute on 64 or more bits.[12] On our machine, four rounds of Simon encryption can execute in a single weird function. As a result, our weird program needs eight weird functions (and thus eight transient windows) to encrypt a Simon block. Even though running a weird function is slower than

11. This limit is microarchitecture-dependent. Our machine (and those used in the Flexo paper [101]) can handle at most 4-input Boolean functions.

12. Recall from Section 3.3 that the number of instructions that a weird function can execute depends on the size of the reorder buffer, which is microarchitecture-dependent.

TABLE 1. ACCURACY, RUNTIME, AND BINARY SIZE OF SINGLE BLOCK SIMON ENCRYPTION. WITHOUT ERROR CORRECTION, OUR WEIRD PROGRAM IS AROUND 18× FASTER THAN FLEXO. WITH ERROR CORRECTION, OUR WEIRD PROGRAM IS MORE THAN ONE ORDER OF MAGNITUDE FASTER THAN FLEXO. THIS RESULT SHOWS HOW TRANSIENT ARCHITECTURAL EXECUTION IMPROVES PERFORMANCE BY ALLOWING WEIRD PROGRAMS TO COMPUTE DIRECTLY WITH ISA INSTRUCTIONS. THE BINARY SIZE IS ALSO MORE THAN 62% SMALLER. WE USE "PP" TO DENOTE THE CHANGES IN PERCENTAGE POINTS.

	Flexo [101]	**Weird programs**	Difference
Accuracy	46.80% ±3.15 pp	91.50% ±5.26 pp	+44.70 pp
Runtime (μs)	2115.01 ±427.99	115.93 ±0.95	18.24×
Binary size (KB)	1137.54	424.05	-62.72%
With error correction			
Accuracy	99.90% ±0.04 pp	94.25% ±1.69 pp	-5.65 pp
Runtime (μs)	3835.09 ±404.92	349.31 ±1.91	10.97×
Binary size (KB)	1137.54	428.05	-62.37%

running a weird gate,[13] running eight weird functions is still significantly faster than running 4,322 weird gates.

Table 1 compares the accuracy, runtime, and binary size of our weird program to Flexo. Without error correction, our weird program is around 18× faster and 45 percentage points more accurate than Flexo. With error correction, Flexo (99.9% accuracy) is slightly more accurate than our weird program (94.3% accuracy). This is because Flexo's differential encoding provides, in effect, error detection for free. The weird program, however, is more than an order of magnitude faster than Flexo, which demonstrates that transient architectural execution's use of the ISA dramatically improves performance. In addition to the improvement in runtime, ISA instructions also allow us to reduce the binary size. While Flexo needs to store thousands of weird gates in its binary, our weird programs only need to contain a small amount of weird functions, reducing the binary size by more than 62%.

5.3. The AES block cipher

Next, we compare the accuracy and runtime of our weird program implementing AES encryption [119] against Flexo's [101] implementation. We use the same 128-bit block and 128-bit key that Flexo [101] does.

Flexo implements one round of AES encryption as a weird circuit (using 2,524 weird gates) and composes 19 such weird circuits to compute the ten AES rounds and round keys. When Flexo composes these 19 weird circuits, it converts the output of the each circuit from a weird register into an architectural register, and uses the architectural register as input to the next weird circuit. This process, which is necessary to run large computations in μWMs, makes the computation less stealthy by architecturally exposing the round keys and the output of each round.

13. Recall from Section 4 that weird functions need to wait several hundreds of cycles to wait for their BTB updates to become available for subsequent weird functions.

```
; generate round key
aeskeygenassist RCON, xmm0, xmm1
pshufd 0b11111111, xmm1, xmm1
shufps 0b00010000, xmm0, xmm2
pxor xmm2, xmm0
shufps 0b10001100, xmm0, xmm2
pxor xmm2, xmm0
pxor xmm1, xmm0
; aes encryption for one round
aesenc xmm0, xmm3

; 9 more rounds ...
```

Listing 7. AES-NI instructions we utilize to implement one round of AES encryption. Our weird function requires only 8 ISA instructions to compute the round key and one round of AES encryption. In contrast, Flexo implements one round of AES encryption using 2,524 weird gates.

TABLE 2. ACCURACY, RUNTIME, AND BINARY SIZE OF SINGLE BLOCK AES ENCRYPTION. WITHOUT ERROR CORRECTION, OUR WEIRD PROGRAM IS 109× FASTER (AND 90.7 PERCENTAGE POINTS MORE ACCURATE) THAN FLEXO. WITH ERROR CORRECTION, OUR WEIRD PROGRAM IS 543× FASTER THAN FLEXO. THIS RESULT SHOWS THAT WEIRD PROGRAMS CAN LEVERAGE COMPLEX ISA INSTRUCTIONS TO IMPROVE PERFORMANCE AND STEALTHINESS COMPARED WITH STATE-OF-THE-ART. THE BINARY SIZE IS AROUND 83% SMALLER. WE USE "PP" TO DENOTE THE CHANGES IN PERCENTAGE POINTS.

	Flexo [101]	**Weird programs**	Difference
Accuracy	0.0%	90.70% ±1.63 pp	+90.7 pp
Runtime (μs)	14894.08 ±193.65	137.26 ±2.61	108.51×
Binary size (KB)	2482.77	423.73	-82.93%
With error correction			
Accuracy	99.70% ±0.18 pp	96.05% ±1.21 pp	-3.65 pp
Runtime (μs)	221583.39 ±6176.02	408.19 ±8.49	542.84×
Binary size (KB)	2482.77	423.73	-82.93%

In contrast, our weird program takes advantage of Intel's AES-NI instruction set [121], which can run a full round of AES using a few instructions; in our testbed, these instructions all fit within a single weird function. This means that our program needs just one weird function to encrypt an AES block, and exposes no intermediate state architecturally during the encryption. Listing 7 shows that this weird function requires only eight ISA instructions to compute the round key and one round of AES encryption.

Table 2 compares the accuracy, runtime, and binary size of our weird program and Flexo's composed AES circuits. Without error correction, our weird program computes a full AES encryption in 137.26 μs with 90.7% accuracy. In contrast, despite taking more than 100× longer, Flexo

```
while (left <= right) {
    mid = left + (right - left) / 2;
    if (data[mid] == target)
        break;
    else if (data[mid] < target)
        left = mid + 1;
    else
        right = mid - 1;
}
```

Listing 8. The binary search algorithm implementation using a weird function. `target` is loaded from the multi-bit weird registers, while `data` is a pointer to read-only main memory. Other variables are transient registers that are initialized when the weird function executes.

achieves 0% accuracy.[14] With error correction, both our weird program and Flexo achieve high accuracy (96.05% and 99.70%, respectively). However, our weird program is 543× faster than Flexo. The binary size is also much smaller. While Flexo's circuits are more than 2 MB in size, our weird programs are around 83% smaller, with the size of 423.73 KB. This demonstrates that weird programs can leverage complex ISA instructions to improve performance and stealthiness compared with state-of-the-art.

5.4. Binary search

Finally, we show how the processor model, as exposed by weird programs, makes it practical to execute computations with control flow and memory accesses (§3.1). We do so by implementing a binary search algorithm using a single weird function whose code is shown in Listing 8. We test our implementation on a sorted 2 MB array that contains 512,000 integers, each of 32-bit size. For each run of the algorithm, we sample a random search target from the array and execute the weird program to find the index of the target. The search target is stored in a weird register, while the array is stored in read-only architectural memory. The binary size of this weird program is 419.76 KB.

Without error correction, the program's runtime is 50.82 μs and the accuracy is 94.6%; with error correction, its runtime is 148.22 μs (roughly 3× slower due to the two-out-of-three majority voting) and its accuracy is 99.9%. This shows that weird programs can execute branch instructions, access weird registers dynamically, and access read-only architectural memory with high performance and accuracy.

Since Flexo operates in the circuit model, implementing binary search in Flexo would require transforming control flow and memory accesses into an expensive series of conditional selects. We can provide a lower bound for the execution time of the search, since we know that this search requires a linear scan over the entire array. We find (and prior work reports) a weird gate runtime of roughly 0.4 μs. Since each weird gate supports at most four input bits, scanning an entire 2 MB requires $\frac{2\times 8}{4} = 4$ million weird gates. Multiplying these two figures, the runtime of any circuit-based implementation is at least 1.6 seconds. In comparison, our weird program runs in around 150 μs, a speedup of more than four orders of magnitude.

14. Note that the original Flexo paper does not report the results of the composed AES circuit without error correction.

TABLE 3. THE PROCESSORS AND AWS EC2 INSTANCES WE USE TO VERIFY THE GENERAL APPLICABILITY OF OUR WEIRD PROGRAMS.

Microarchitecture	Instance type	Processor
Sapphire Rapids	m7i.xlarge	Intel Xeon 8488C
Emerald Rapids	i7i.xlarge	Intel Xeon 8559C
Granite Rapids	r8i.xlarge	Intel Xeon 6975P-C

TABLE 4. RESULTS OF RUNNING OUR WEIRD PROGRAMS (OPTIMIZED FOR OUR TESTBED MACHINE) ON THREE AWS EC2 INSTANCES. WE USE "PP" TO DENOTE THE CHANGES IN PERCENTAGE POINTS.

		Sapphire Rapids	**Emerald Rapids**	**Granite Rapids**
Simon	Accuracy	75.95% ±15.32 pp	77.30% ±7.35 pp	35.75% ±4.20 pp
	Runtime (μs)	112.38 ±13.25	116.45 ±8.88	98.53 ±2.86
AES	Accuracy	63.40% ±8.13 pp	70.85% ±3.55 pp	54.95% ±4.70 pp
	Runtime (μs)	207.66 ±11.36	189.72 ±7.86	217.01 ±23.20
Binary search	Accuracy	84.95% ±6.33 pp	91.00% ±3.37 pp	82.00% ±6.63 pp
	Runtime (μs)	48.50 ±1.05	49.16 ±0.97	45.45 ±6.68
With error correction				
Simon	Accuracy	85.00% ±7.09 pp	86.10% ±6.03 pp	34.00% ±3.14 pp
	Runtime (μs)	328.56 ±17.91	342.75 ±14.33	313.45 ±10.89
AES	Accuracy	77.75% ±8.33 pp	85.20% ±6.24 pp	58.65% ±6.83 pp
	Runtime (μs)	617.92 ±32.05	581.37 ±24.29	764.88 ±52.85
Binary search	Accuracy	96.60% ±1.46 pp	98.90% ±1.17 pp	91.80% ±7.06 pp
	Runtime (μs)	140.23 ±2.13	143.38 ±2.22	132.03 ±27.33

5.5. Portability to other microarchitectures

We evaluate the portability of our weird programs on three AWS EC2 instances with Intel Sapphire Rapids, Emerald Rapids, and Granite Rapids CPUs (Table 3). We use weird programs tuned specifically for our testbed machine, featuring a Sapphire Rapids processor. As Table 4 shows, the results of running the binary depend on the machine they execute on. For machines with similar processors, we note a small, albeit noticeable, drop in the accuracy. However, as the processor model diverges, as in moving to the more recent Granite Rapids processor, the accuracy further drops. This highlights the need to tune weird programs to the target

machine. We leave investigating the causes of the drop and developing automatic tuning mechanisms to future work.

6. Discussion and future work

In this section we briefly discuss limitations of the current work and potential future directions for improvement.

Stealth. One notable application of μWMs is to perform *program obfuscation*. Prior work has used μWMs to obfuscate malicious code for malware [97] and to construct a binary packer [101] that uses a weird circuit to decrypt the payload at runtime, frustrating dynamic and static analysis.

From the perspective of architectural state, μWMs are naturally stealthy: they execute transiently and leave only microarchitectural traces. This hinders or prevents dynamic analysis in multiple ways. First, existing debuggers cannot analyze values in weird registers because they have little or no access to the underlying microarchitectural state. Second, single-stepping through the execution of a μWM alters or eliminates the processor's transient execution behavior, with the result that μWMs do not execute correctly. Our weird programs provide the same protections against dynamic analysis, since they also execute transiently and operate on microarchitectural state (including multi-bit weird registers).

An interesting question is whether it would be possible to augment dynamic analysis to detect or de-obfuscate μWMs. Regarding detection, one possible approach might be to monitor for abnormal spikes in the number of instructions that are squashed (e.g., using hardware performance counters [122], [123]). Regarding de-obfuscation, one possible approach might involve μWM emulation. For example, concurrent work by Vanspauwen et al. proposes a tool that emulates the microarchitectural components that cache-based weird circuits use for computation, enabling an analyst to step through transient instructions in (and microarchitectural state changes due to) μWM binaries [124]. However, future work is needed to generalize their approach beyond weird circuits and cache-based weird registers.

When it comes to static analysis, however, our weird programs do not seem quite as stealthy since they include explicit ISA instructions that may be parsed and analyzed by existing binary reverse-engineering tools. Prior Boolean circuits based μWMs may seem stealthier in this regard, because they are instantiated via carefully constructed race conditions (§2.3). This stealth may, however, be illusory: statically reconstructing the Boolean circuit from μWM instruction sequences seems likely to be feasible, at which point an analyst could apply existing tools for reverse engineering digital circuits [125], [126]. Carefully studying the problem of statically analyzing μWMs and weird programs is an interesting future direction.

Applicability to other microarchitectures. The evaluation in this work considers weird programs tailored to a single Intel microarchitecture. As discussed in prior work [101], portability across microarchitectures is important in many applications (e.g., malware obfuscation). In Section 5.5, we show that our weird program implementation is portable to similar Intel microarchitectures, but tuning is needed to achieve high accuracy. Future work is needed to study the portability of our design to other Intel and non-Intel microarchitectures. In particular, such future work will require the reverse engineering of non-Intel processors' BTB (e.g., AMD's BTB), since detailed knowledge of the BTB is an essential prerequisite to constructing BTB weird registers.

Multi-bit weird registers beyond the BTB. It may be possible to use other microarchitectural components to build multi-bit weird registers. However, there are several challenges, such as how to convert microarchitectural states into architectural values during transient execution. Future work is needed to overcome these challenges.

Weird program optimizations. Further optimizations of weird programs seem possible. As one example, consider the case of *batch execution*, i.e., running the same program on many different inputs. In this regime, the cost of BTB updates (§4) could easily be amortized across many executions of the same program by repeatedly executing each weird function on different inputs before modifying the BTB in preparation for executing the next weird function. Put another way, all instances in the batch would be executed in lock-step, one weird function at a time.

Weird programs as a compilation target. Prior work has already demonstrated that compilation to μWMs is possible. While compiling to weird programs is likewise possible, there are several challenges. For example, since weird functions need to finish execution in a transient window with a limited length, one challenge is how to break a computation into weird functions so that each weird function completes execution within a transient window. As different input values will cause the weird function to have different execution times, it is challenging to ensure that the execution is upper bounded by the transient window length.

Acknowledgments

We thank the reviewers for their helpful feedback. This work was funded by an Ann and Martin McGuinn Graduate Fellowship, ARL grant W911NF-25-1-0179, ARC Discovery Project number DP210102670, the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and through project no. 560392681, ISF grant no. 1807/23, Len Blavatnik and the Blavatnik Family Foundation, and the Stellar Development Foundation.

References

[1] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *JCEN*, vol. 8, no. 1, 2018.

[2] C. Percival, "Cache missing for fun and profit," in *BSDCan*, 2005.

[3] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *CT-RSA*, 2006.

[4] M. Neve and J.-P. Seifert, "Advances on access-driven cache attacks on AES," in *SAC*, 2006.

[5] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *ACSAC*, 2006.

[6] O. Acıiçmez and J.-P. Seifert, "Cheap hardware parallelism implies cheap security," in *FDTC*, 2007.

[7] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *CT-RSA*, 2007.

[8] O. Acıiçmez, "Yet another microarchitectural attack: Exploiting I-Cache," in *CSAW*, 2007.

[9] O. Acıiçmez and W. Schindler, "A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL," in *CT-RSA*, 2008.

[10] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games - bringing access-based cache attacks on AES to practice," in *S&P*, 2011.

[11] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *CCS*, 2012.

[12] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: High-speed covert channel attacks in the cloud," in *USENIX Security*, 2012.

[13] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! A fast, cross-VM attack on AES," in *RAID*, 2014.

[14] Y. Yarom and K. Falkner, "Flush+Reload: a high resolution, low noise, L3 cache side-channel attack," in *USENIX Security*, 2014.

[15] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in PaaS clouds," in *CCS*, 2014.

[16] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *S&P*, 2015.

[17] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES," in *S&P*, 2015.

[18] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *USENIX Security*, 2015.

[19] C. Maurice, C. Neumann, O. Heen, and A. Francillon, "C5: Cross-cores cache covert channel," in *DIMVA*, 2015.

[20] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: Practical cache attacks in JavaScript and their implications," in *CCS*, 2015.

[21] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A fast and stealthy cache attack," in *DIMVA*, 2016.

[22] M. Kayaalp, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, "A high-resolution side-channel attack on the last level cache," in *DAC*, 2016.

[23] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Understanding and mitigating covert channels through branch predictors," *TACO*, 2016.

[24] D. Evtyushkin and D. Ponomarev, "Covert channels through random number generator: Mechanisms, capacity estimation and mitigations," in *CCS*, 2016.

[25] R. Guanciale, H. Nemati, C. Baumann, and M. Dam, "Cache storage channels: Alias-driven attacks and verified countermeasures," in *S&P*, 2016.

[26] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over ASLR: Attacking branch predictors to bypass ASLR," in *MICRO*, 2016.

[27] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-CPU attacks," in *USENIX Security*, 2016.

[28] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross processor cache attacks," in *ASIACCS*, 2016.

[29] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+Abort: A timer-free high-precision L3 cache attack using intel TSX," in *USENIX Security*, 2017.

[30] D. Genkin, L. Valenta, and Y. Yarom, "May the fourth be with you: A microarchitectural side channel attack on several real-world applications of Curve25519," in *CCS*, 2017.

[31] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the line: Practical cache attacks on the MMU," in *NDSS*, 2017.

[32] Y. Yarom, D. Genkin, and N. Heninger, "CacheBleed: A timing attack on OpenSSL constant time RSA," *JCEN*, vol. 7, no. 2, 2017.

[33] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," in *S&P*, 2019.

[34] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *USENIX Security*, 2018.

[35] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Robust website fingerprinting through the cache occupancy channel," in *USENIX Security*, 2019.

[36] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *S&P*, 2019.

[37] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: Exploiting speculative execution through port contention," in *CCS*, 2019.

[38] S. Cohney, A. Kwong, S. Paz, D. Genkin, N. Heninger, E. Ronen, and Y. Yarom, "Pseudorandom black swans: Cache attacks on CTR DRBG," in *S&P*, 2020.

[39] D. F. Aranha, F. R. Novaes, A. Takahashi, M. Tibouchi, and Y. Yarom, "LadderLeak: Breaking ECDSA with less than one bit of nonce leakage," in *CCS*, 2020.

[40] B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi, "ABSynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures," in *NDSS*, 2020.

[41] M. Yan, C. W. Fletcher, and J. Torrellas, "Cache telepathy: Leveraging shared resource attacks to learn DNN architectures," in *USENIX Security*, 2020.

[42] M. Kurth, B. Gras, D. Andriesse, C. Giuffrida, H. Bos, and K. Razavi, "NetCAT: Practical cache attacks from the network," in *S&P*, 2020.

[43] M. Lipp, V. Hadžić, M. Schwarz, A. Perais, C. Maurice, and D. Gruss, "Take a way: Exploring the security implications of AMD's cache way predictors," in *ASIACCS*, 2020.

[44] A. Shusterman, A. Agarwal, S. O'Connell, D. Genkin, Y. Oren, and Y. Yarom, "Prime+Probe 1, JavaScript 0: Overcoming browser-based side-channel defenses," in *USENIX Security*, 2021.

[45] R. Paccagnella, L. Luo, and C. W. Fletcher, "Lord of the ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical," in *USENIX Security*, 2021.

[46] F. Sieck, S. Berndt, J. Wichelmann, and T. Eisenbarth, "Util:: Lookup: Exploiting key decoding in cryptographic libraries," in *CCS*, 2021.

[47] A. Purnal, F. Turan, and I. Verbauwhede, "Prime+Scope: Overcoming the observer effect for high-precision cache contention attacks," in *CCS*, 2021.

[48] M. Dai, R. Paccagnella, M. Gomez-Garcia, J. McCalpin, and M. Yan, "Don't mesh around: Side channel attacks and mitigations on mesh interconnects," in *USENIX Security*, 2022.

[49] Y. Guo, A. Zigerelli, Y. Zhang, and J. Yang, "Adversarial prefetch: New cross-core cache side channel attacks," in *S&P*, 2022.

[50] Y. Guo, X. Xin, Y. Zhang, and J. Yang, "Leaky way: A conflict-based cache covert channel bypassing set associativity," in *MICRO*, 2022.

[51] J. R. S. Vicarte, M. Flanders, R. Paccagnella, G. Garrett-Grossman, A. Morrison, C. W. Fletcher, and D. Kohlbrenner, "Augury: Using data memory-dependent prefetchers to leak data at rest," in *S&P*, 2022.

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

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

[54] J. Yu, A. Dutta, T. Jaeger, D. Kohlbrenner, and C. W. Fletcher, "Synchronization storage channels (S2C): Timer-less cache side-channel attacks on the Apple M1 via hardware synchronization instructions," in *USENIX Security*, 2023.

[55] S. Gast, J. Juffinger, M. Schwarzl, G. Saileshwar, A. Kogler, S. Franza, M. Köstl, and D. Gruss, "SQUIP: Exploiting the scheduler queue contention side channel," in *S&P*, 2023.

[56] Z. Zhang, M. Tao, S. O'Connell, C. Chuengsatiansup, D. Genkin, and Y. Yarom, "BunnyHop: Exploiting the instruction prefetcher," in *USENIX Security*, 2023.

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

[58] H. Yavarzadeh, A. Agarwal, M. Christman, C. Garman, D. Genkin, A. Kwong, D. Moghimi, D. Stefan, K. Taram, and D. Tullsen, "Pathfinder: High-resolution control-flow attacks exploiting the conditional branch predictor," in *ASPLOS*, 2024.

[59] F. Rauscher, C. Fiedler, A. Kogler, and D. Gruss, "A systematic evaluation of novel and existing cache side channels," in *NDSS*, 2025.

[60] L. Hetterich, F. Thomas, L. Gerlach, R. Zhang, N. Bernsdorf, E. Ebert, and M. Schwarz, "Shadowload: Injecting state into hardware prefetchers," in *ASPLOS*, 2025.

[61] I. Chun, I. Siu, and R. Paccagnella, "Scheduled disclosure: Turning power into timing without frequency scaling," in *S&P*, 2025.

[62] L. Gerlach, N. Flentje, and M. Schwarz, "Zero-store elimination and its implications on the SIKE cryptosystem," in *uASC*, 2026.

[63] 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.

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

[65] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *WOOT*, 2018.

[66] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," 2018, preprint, arXiv:1807.03757 [cs.CR].

[67] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *CCS*, 2018.

[68] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *USENIX Security*, 2018.

[69] 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.

[70] 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.

[71] 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.

[72] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lippi, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking transient execution through microarchitectural load value injection," in *S&P*, 2020.

[73] D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz, "Medusa: Microarchitectural data leakage via automated attack synthesis," in *USENIX Security*, 2020.

[74] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "Crosstalk: Speculative data leaks across cores are real," in *S&P*, 2021.

[75] H. Ragab, E. Barberis, H. Bos, and C. Giuffrida, "Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks," in *USENIX Security*, 2021.

[76] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "CacheOut: Leaking data on Intel CPUs via cache evictions," in *S&P*, 2021.

[77] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. Zhao, X. Zou, T. Unterluggauer, J. Torrellas, C. Rozas, A. Morrison, F. Mckeen, F. Liu, R. Gabor, C. W. Fletcher, A. Basak, and A. Alameldeen, "Speculative interference attacks: Breaking invisible speculation schemes," in *ASPLOS*, 2021.

[78] O. Kirzner and A. Morrison, "An analysis of speculative type confusion vulnerabilities in the wild," in *USENIX Security*, 2021.

[79] J. Wikner and K. Razavi, "Retbleed: Arbitrary speculative code execution with return instructions," in *USENIX Security*, 2022.

[80] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, "PACMAN: Attacking ARM pointer authentication with speculative execution," in *ISCA*, 2022.

[81] J. Wikner, D. Trujillo, and K. Razavi, "Phantom: Exploiting decoder-detectable mispredictions," in *MICRO*, 2023.

[82] D. Trujillo, J. Wikner, and K. Razavi, "Inception: Exposing new attack surfaces with training in transient execution," in *USENIX Security*, 2023.

[83] R. Zhang, T. Kim, D. Weber, and M. Schwarz, "(M)WAIT for it: Bridging the gap between microarchitectural and architectural side channels," in *USENIX Security*, 2023.

[84] D. Moghimi, "Downfall: Exploiting speculative data gathering," in *USENIX Security*, 2023.

[85] S. Wiebing, A. de Faveri Tron, H. Bos, and C. Giuffrida, "InSpectre Gadget: Inspecting the residual attack surface of cross-privilege Spectre v2," in *USENIX Security*, 2024.

[86] M. Hertogh, S. Wiebing, and C. Giuffrida, "Leaky Address Masking: Exploiting unmasked Spectre gadgets with noncanonical address translation," in *S&P*, 2024.

[87] A. Wang, B. Chen, Y. Wang, C. Fletcher, D. Genkin, D. Kohlbrenner, and R. Paccagnella, "Peek-a-Walk: Leaking secrets via page walk side channels," in *S&P*, 2025.

[88] J. Wikner and K. Razavi, "Breaking the barrier: Post-barrier Spectre attacks," in *S&P*, 2025.

[89] S. Wiebing and C. Giuffrida, "Training Solo: On the limitations of domain isolation against Spectre-v2 attacks," in *S&P*, 2025.

[90] S. Rüegge, J. Wikner, and K. Razavi, "Branch Privilege Injection: Compromising Spectre v2 hardware mitigations by exploiting branch predictor race conditions," in *USENIX Security*, 2025.

[91] J. Kim, D. Genkin, and Y. Yarom, "SLAP: Data speculation attacks via load address prediction on Apple Silicon," in *S&P*, 2025.

[92] J. Kim, J. Chuang, D. Genkin, and Y. Yarom, "FLOP: Breaking the Apple M3 CPU via false load output predictions," in *USENIX Security*, 2025.

[93] Y. Zhu and A. Biondi, "Exploiting inaccurate branch history in side-channel attacks," in *USENIX Security*, 2025.

[94] M. Hertogh, D. Quakkelaar, T. Raymakers, M. Hari Sarma, M. Muench, H. Bos, and E. van der Kouwe, "Rain: Transiently leaking data from public clouds using old vulnerabilities," in *S&P*, 2026.

[95] J.-C. Graf, S. Rüegge, A. Hajiabadi, and K. Razavi, "VMScape: Exposing and exploiting incomplete branch predictor isolation in cloud environments," in *S&P*, 2026.

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

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

[98] P.-L. Wang, F. Brown, and R. S. Wahby, "The ghost is the machine: Weird machines in transient execution," in *WOOT*, 2023.

[99] D. Katzman, W. Kosasih, C. Chuengsatiansup, E. Ronen, and Y. Yarom, "The Gates of Time: Improving cache attacks with transient execution," in *USENIX Security*, 2023.

[100] D. Kaplan, "Optimization and amplification of cache side channel signals," 2023, preprint, arXiv:2303.00122 [cs.CR].

[101] P.-L. Wang, R. Paccagnella, R. S. Wahby, and F. Brown, "Bending microarchitectural weird machines towards practicality," in *USENIX Security*, 2024.

[102] G. Horowitz, E. Ronen, and Y. Yarom, "Spec-o-Scope: Cache probing at cache speed," in *CCS*, 2024.

[103] B. Morgan, G. Horowitz, S. O'Connell, S. van Schaik, C. Chuengsatiansup, D. Genkin, O. Maennel, P. Montague, E. Ronen, and Y. Yarom, "Slice+Slice Baby: Generating last-level cache eviction sets in the blink of an eye," in *S&P*, 2025.

[104] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *USENIX Security*, 2019.

[105] L. Li, H. Yavarzadeh, and D. Tullsen, "Indirector: High-precision branch target injection attacks exploiting the indirect branch predictor," in *USENIX Security*, 2024.

[106] A. Ozdemir, F. Brown, and R. S. Wahby, "CirC: Compiler infrastructure for proof systems, software verification, and more," in *S&P*, 2022.

[107] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *TACAS*, 2004.

[108] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *FMCO*, 2005.

[109] Z. Rakamarić and M. Emmi, "SMACK: Decoupling source language details from verifier implementations," in *CAV*, 2014.

[110] E. Torlak and R. Bodik, "A lightweight symbolic virtual machine for solver-aided host languages," in *PLDI*, 2014.

[111] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, "Fairplay–A secure two-party computation system," in *USENIX Security*, 2004.

[112] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish, "Verifying computations with state," in *SOSP*, 2013.

[113] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *S&P*, 2013.

[114] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur, "Geppetto: Versatile verifiable computation," in *S&P*, 2015.

[115] R. S. Wahby, S. Setty, Z. Ren, A. J. Blumberg, and M. Walfish, "Efficient RAM and control flow in verifiable outsourced computation," in *NDSS*, 2015.

[116] Z. Zhang, G. Barthe, C. Chuengsatiansup, P. Schwabe, and Y. Yarom, "Ultimate SLH: taking speculative load hardening to the next level," in *USENIX Security*, 2023.

[117] H. Yavarzadeh, M. Taram, S. Narayan, D. Stefan, and D. Tullsen, "Half&half: Demystifying Intel's directional branch predictors for fast, secure partitioned execution," in *S&P*, 2023.

[118] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "The SIMON and SPECK lightweight block ciphers," in *DAC*, 2015.

[119] J. Daemen and V. Rijmen, *The Design of Rijndael*. Berlin, Heidelberg: Springer-Verlag, 2002.

[120] P.-L. Wang, "Flexo," https://github.com/joeywang4/Flexo, 2024, accessed on Jun 5, 2025.

[121] Intel Corporation, "Intel Advanced Encryption Standard (AES) Instructions Set," Intel Corporation, Tech. Rep., 2010.

[122] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose, "SoK: The challenges, pitfalls, and perils of using hardware performance counters for security," in *S&P*, 2019.

[123] W. Kosasih, Y. Feng, C. Chuengsatiansup, Y. Yarom, and Z. Zhu, "SoK: Can we really detect cache side-channel attacks by monitoring performance counters?" in *ASIACCS*, 2024.

[124] D. Dries Vanspauwen, L.-A. Daniel, and J. Van Bulck, "WeMu: Effective and scalable emulation of microarchitectural weird machines," in *uASC*, 2026.

[125] M. Fyrbiak, S. Wallat, P. Swierczynski, M. Hoffmann, S. Hoppach, M. Wilhelm, T. Weidlich, R. Tessier, and C. Paar, "HAL- the missing piece of the puzzle for hardware reverse engineering, trojan detection and insertion," *IEEE Transactions on Dependable and Secure Computing*, 2018.

[126] Embedded Security Group, "Hal - the hardware analyzer," https://github.com/emsec/hal, 2019, accessed on Jun 5, 2025.