Bending microarchitectural weird machines towards practicality

Ping-Lun Wang, Riccardo Paccagnella, Riad S. Wahby, and Fraser Brown
Carnegie Mellon University

Abstract

A large body of work has demonstrated attacks that rely on the difference between CPUs' nominal instruction set architectures and their actual (microarchitectural) implementations. Most of these attacks, like Spectre, bypass the CPU's data-protection boundaries. A recent line of work considers a different primitive, called a *microarchitectural weird machine* (μWM), that can execute computations almost entirely using microarchitectural side effects. While μWMs would seem to be an extremely powerful tool, e.g., for obfuscating malware, thus far they have seen very limited application. This is because prior μWMs must be hand-crafted by experts, and even then have trouble reliably executing complex computations.

In this work, we show that μWMs are a practical, near-term threat. First, we design a new μWM architecture, Flexo, that improves performance by 1–2 orders of magnitude and reduces circuit size by 75–87%, dramatically improving the applicability of μWMs to complex computation. Second, we build the first compiler from a high-level language to μWMs, letting experts craft automatic optimizations and non-experts construct state-of-the-art obfuscated computations. Finally, we demonstrate the practicality of our approach by extending the popular UPX packer to encrypt its payload and use a μWM for decryption, frustrating malware analysis.

1 Introduction

Modern processors achieve impressive performance through astonishing complexity. In the last two decades, however, it has become clear that this complexity threatens to completely undermine memory isolation, which is crucial for nearly every aspect of software security. This realization followed the discovery of numerous microarchitectural attacks capable of leaking sensitive information—in the worst case, an entire program's memory—even in the absence of software bugs [3–8, 13, 15, 21, 26, 31, 34, 39–54, 57–60, 64, 66–69, 72, 76–78, 80, 82, 84, 85, 87, 89, 90, 92, 94–100, 104, 105, 109–111, 113, 116–118, 126–128, 130, 133–136, 139–141].

These attacks stem from the difference between the abstraction the processor is *intended* to expose and the abstraction that its implementation *actually* exposes—a gap that is primarily the result of complexity in service of performance. As an example, a recent wave of microarchitectural attacks relies on the fact that the CPU may transiently execute code that, per the semantics of the instruction set architecture (ISA), it should not [13, 21, 52, 57, 66–69, 77, 80, 84, 85, 97–100, 105, 113, 116–118, 127, 128, 139]. While the ISA guarantees that architectural state (e.g., the contents of registers and memory) does not reflect such transient execution, it makes no promises about implicit, microarchitectural state (e.g., the contents of the cache or the internal state of the CPU's branch prediction machinery). Crucially, although this microarchitectural state is not exposed by the ISA, programs can nevertheless modify and observe this state (e.g., through timing measurements that reveal cache residency). The result is that the *actual* processor architecture is a strict superset of the *intended* one (i.e., the ISA)—and that superset can give attackers superpowers.

A processor's superset-of-ISA functionality is an instance of the *weird machine* paradigm [11, 17, 35, 36] which considers, for a given system, a "weird system" that extends the system's intended behavior with its unintended but observable behavior. This paradigm formalizes the notion that an attacker's interaction with a system is bound by its *actual* implementation, not by the designer's intent. For CPUs, the past two decades have demonstrated empirically that securing the ISA is insufficient: it is the entire weird system—the ISA plus observable microarchitecture—that must withstand attacks.

One recent line of work [38, 62, 63, 123, 125] explores *microarchitectural weird machines* (μWMs)—code gadgets that perform computation purely through microarchitectural side effects.[1] μWMs resist analysis, both because they leave little or no trace in the CPU's architectural state and because they are essentially invisible to conventional debuggers (as one

[1] Calling this code a weird machine is arguably an abuse of terminology: strictly speaking, the code executes *on* the computational substrate exposed by the weird superset of the ISA that is inherent in the CPU's microarchitecture. We nevertheless follow colloquial usage and call the code gadgets μWMs.

example: single-stepping through a μWM destroys the computation). At first blush, such stealthy computation seems as if it should lead to attacks at least as powerful as prior work focused on crossing data-protection boundaries (e.g., Spectre [68] and Meltdown [77]); indeed, prior work predicted that μWMs would see widespread use in applications like malware obfuscation. To date, however, their only applications have been clever but somewhat narrow: obfuscating simple computations like 160-bit XOR and 2-block SHA-1 [38], and improving cache side-channel attacks [62, 63].

This motivates our research question: *is there any reason to worry that μWMs can be made practical?*—and thus, is there any reason to invest effort in developing countermeasures? By practical, we mean two specific criteria, neither of which has been satisfied by prior work. First, it should be possible to build μWMs that reliably compute *complex* functions with reasonable performance across a range of microarchitectures. Second, such μWMs should be broadly usable, not limited to a handful of experts who painstakingly hand-craft weird functionality. In this paper we address these challenges, then answer our motivating question in the affirmative by building a malware packer that uses μWMs to hinder analysis (§6).

In prior work, reliability of μWMs on complex computations is a thorny issue: a μWM's reliability tends to fall off precipitously as the size of the computation increases. While prior work has explored techniques for detecting and correcting errors, these tend to dramatically increase a computation's execution time even when no error occurs. Our μWM design (§3), somewhat unintuitively, starts with a more costly *differential* logic encoding (i.e., one in which each bit is represented by the difference between two "wires"). Although this seems like it should make scaling more difficult, it has the effect of making error detection and logical inversion nearly free—and since these were by far the *most costly* primitives in prior work (§2.2), the result is dramatically improved scaling.

This μWM design, which we call Flexo, is also far more generic than prior work: rather than implementing a family of small logic gates, we give a design that can compute *any* N-input boolean function (N is determined by the CPU; our testbed machines can handle 4-input functions; §3.1, §5). This makes our design particularly suitable as a compilation target, a fact we leverage in our first-of-its-kind μWM compiler (§4). Our compilation toolchain, which extends LLVM [73], consumes C/C++ code and emits an assembly Flexo implementation. Beyond making Flexo accessible to non-experts, our compiler makes it possible to create μWM-specific optimization passes; we implement one as a proof of concept.

In summary, we make the following contributions:

- We propose a new circuit design for μWMs that reduces circuit size by 75–87%, resulting in 1–2 orders of magnitude speedup. Our design enables bit-wise error detection and correction—in one case improving a circuit's accuracy from 0.3% to 99.9%—while imposing almost no overhead for circuits with high uncorrected accuracy.
- We design a compiler that can turn high-level C/C++ code into a μWM and use it to compile the SHA-1 hash and the AES and Simon ciphers into μWMs.
- We present a weird machine packer that uses Flexo to obfuscate UPX's [88] unpacking process. Our packer encrypts the packed executable, which a μWM later decrypts at runtime. The packer currently supports AES and Simon decryption modules, which we implement in under six hundred lines of C—but our compiler makes it easy to build different obfuscated encryption schemes.

2 Background and related work

This section discusses transient execution (§2.1), upon which we (and others) build microarchitectural weird machines (§2.2). We discuss related work on circuit compilers in Section 4, and likewise for binary packers in Section 6.

2.1 Transient execution

Transient execution occurs when a processor executes *transient instructions* that are never committed to the architectural state [22]. Even though they do not change architectural state, transient instructions can leave traces in the microarchitectural state (e.g., the cache), and these traces can be observed by an attacker using microarchitectural side channels [3–8, 15, 26, 31, 34, 39–51, 53, 54, 58–60, 64, 72, 76, 78, 82, 87, 89, 90, 92, 94–96, 104, 109–111, 126, 130, 133–136, 140, 141]. This behavior is the foundation of *transient execution attacks* [13, 21, 52, 57, 66–69, 77, 80, 84, 85, 97–100, 105, 113, 116–118, 127, 128, 139], in which attackers exploit transient execution to access and leak program data. We adopt prior work's terminology and define the *transient window* as the period of time during which a processor executes transient instructions (e.g., following a branch misprediction or a fault). The *length* of a transient window depends on the time required for the processor to realize that the instructions executed within it were, in fact, transient and should be discarded.

As a concrete example of transient execution—in fact, the kind that our compiler targets (§4)—consider transient execution due to return address misprediction [69, 80]. The return stack buffer (RSB) is a small per-core structure that predicts the target of `ret` instructions. The RSB stores the address of each instruction that follows a `call` instruction, predicting that the calls will return to the addresses it has stored. The prediction is *usually* correct—except when a function overwrites its return address during execution. In such cases, the processor transiently executes at the (mispredicted) address stored in the RSB until it knows the function's real return address. The length of the transient window, therefore, depends on the time the function takes to compute and load its

actual return address. By computing that return address using different sequences of high-latency instructions, an attacker can tune the length of the transient window.

2.2 Microarchitectural weird machines

Microarchitectural weird machines (μWMs) run computations that are, in theory, invisible to architectural state: they use cache residency information to store values, and transient execution and microarchitectural side channels to compute on those values.[2] The first μWM [38] relied on Intel TSX for transient execution and could obfuscate 160-bit XOR and 2-block SHA-1. However, it was relatively slow (26.5 min to run 2-block SHA-1) and leaked about 40% of its intermediate values to architectural state. Subsequent work improved μWMs' construction and expanded their applications [62, 63, 125]. Wang et al. [125] show that any transient primitive (e.g., exceptions and branch predictors) can create μWMs, and that both x86_64 and ARM CPUs can execute them. Kaplan [62] and Katzman et al. [63] build μWMs that amplify cache side-channel signals, and Katzman et al. [63] also present μWMs that are much faster (923 ms to run 2-block SHA-1) and stealthier (leaking about 1% of intermediate values to architectural state) than prior work.

Computational model μWMs compute on *weird registers*, registers stored in microarchitectural state. Each weird register can store a binary value and is associated with a cache line. The value of a weird register is determined by the *cache residency* of its cache line: if its line is in the cache, the weird register value is '1'; if not, the weird register value is '0'. In other words, weird registers are memory addresses that store values in microarchitectural (i.e., cache) state, rather than using main memory to store values in architectural state.

To set the value of a weird register to '1,' a μWM must bring the register's cache line into the cache by loading or storing its memory address. To set the value of a weird register to '0', a μWM must remove its cache line from the cache by flushing its memory address (e.g., using `clflush`[3]).

μWMs use transient execution to execute code gadgets that modify the values of weird registers. These gadgets (or "weird gates" or "circuits") implement boolean functions with one or more input registers and a single output register. In the following, we give examples of basic gates that, e.g., set the value of an output weird register when both input registers are '1.' In these examples, the input registers always have an all-zeros architectural value (i.e., the value in main memory), and the output weird register's microarchitectural value is initially '0' (i.e., flushed from cache).

`AND` and `OR` gates Prior work [63, 125] builds `AND` and `OR` gates by exploiting the time difference between (fast) cache hits and (slow) cache misses. Listing 1, for example, shows a weird gate that computes output `Out[0]` as `In1[0] AND In2[0]`—so `Out[0]` ends up in the cache (i.e., as '1') only if `In1[0]` and `In2[0]` were in the cache (i.e., were '1') to begin with. To execute this code, the processor checks the cache for both weird registers, and fetches them if they are not there. It uses the architectural values of these registers to compute the address of `Out[0]`, which it then fetches into cache. Therefore, if the transient window is shorter than the cache miss latency, the output weird register will end up in cache *only if* both input registers are already in cache; otherwise, the processor will not have time to pull the output register into cache (i.e., set its value to '1') before transient execution ends. Building a correct gate requires controlling the length of the transient window; see the prior section for more detail on controlling this length.

```
1  Out[In2[In1[0]]] = 0;
```

```
1  Out[In1[0]] = 0;
2  Out[In2[0]] = 0;
```

Listing 1: The `AND` gate (left) and the `OR` gate (right). The output is initially not in the cache, and the inputs have architectural values set to zero. These gates are executed in transient execution.

`NOT` gate Prior work [62, 63, 125] uses μWMs to implement the `NOT` gate in Listing 2, with input weird register `In[0]` and output register `Out[0]`. This gate controls the length of the transient window so that the processor only fetches the output into the cache when the input *is not* in the cache. The first step (not pictured) is training the branch predictor to take the false branch at line one and jump straight to line two. Then, when the weird gate executes, the branch predictor mispredicts the branch and transiently executes line two. If `In[0]` is in cache ('1'), the processor can resolve the branch target quickly. This *does not* give the processor time to pull `Out[0]` into cache, since `delay` is a function that takes more time than a cache hit (but less than a cache miss) to compute. In contrast, when `In[0]` is not in cache (i.e., '0'), the processor has time to bring `Out[0]` into cache (i.e., make it '1').

```
1  if (In[0] == 0) return; // trained to go to line 2
2  Out[delay(0)] = 0; // cache hit < delay < cache miss
```

Listing 2: The `NOT` gate. The output is initially not in the cache, and the input has the architectural value set to zero.

Executing more complex logic in a single transient window So far we have discussed executing a single, one- or two-input gate per transient window. In practice, it is more efficient to execute as much weird computation as possible within a single transient window. For example, Listing 3 uses an `AND` gate

[2] Note that constructions that do not rely on the cache (as mentioned in [38]) or on transient execution (as shown in [137]) are also possible.

[3] Or e.g., `DC CIVAC + DSB SY` on ARM.

and an OR gate to compute Out[0] = (In1[0] ∧ In2[0]) ∨ In3[0] within one transient execution.

```
1 Out[In1[In2[0]]] = 0;
2 Out[In3[0]] = 0;
```

Listing 3: A composed AND and OR gate. The output is initially not in the cache, and the inputs have architectural values set to zero. This gate is executed in transient execution.

Unfortunately, with the constructions discussed in this section, it is impossible to compose inverting and non-inverting gates (e.g., AND and NOT) within one transient execution. While non-inverting gates use a fixed transient window length that is slightly shorter than the cache miss latency, inverting gates have a window length that depends on their inputs. Therefore, choosing a specific transient window length for the *composition* of both gates would break one gate or the other. As a result, building most complex functionality has in all prior work required executing weird computations in many distinct transient windows of distinct length.

3 New circuit construction

We propose a new weird circuit construction, Flexo, for computing arbitrary N-input, one-output boolean functions within a single transient execution. The number of inputs, N, is architecture-dependent; the x86_64 machines we evaluate in this paper support 4-input functions (§3.1, §5).

Flexo uses a *differential* (or *dual-rail*) encoding [115, Ch. 9], wherein a logical value is represented as the *difference* between two "wire" values. Here, a wire is exactly a weird register, and its value is stored as cache residency information. At first glance, using twice as many wires seems wasteful—and for simple boolean functions like AND and OR, it is. But when expressing more complex functionality, the benefits of our generic, high-level computational primitive outweigh the costs (§5). Specifically, Flexo improves:

- **Performance**. Prior work implements complex functionality by composing a small set of simple gates (e.g., AND, OR, and NOT [38, 62, 63, 125]); in general, such compositions span multiple transient executions. Implementing the same functionality with Flexo uses *fewer*, more complex gates, each taking a single transient execution. Empirically, this yields smaller, faster circuits. For example, all prior work computes 2-input XOR by composing (at least) five transient executions; Flexo computes 4-input XOR in a *single* execution.
- **Accuracy**. Since circuits built from Flexo gates are more concise than existing constructions, they are less susceptible to errors caused by microarchitectural noise. Further, even when errors happen, Flexo's differential encoding makes them easy to detect (and thus correct!): since only two of the four possible configurations for a gate's output value are valid, the other two configurations indicate an error in the gate's execution (§3.2).
- **Flexibility**. Because Flexo can implement *any* N-to-1 boolean function (for our testbeds, $N = 4$; §5), it is an easier compilation target than prior designs: our compiler synthesizes a set of 4-input truth tables, then implements them directly. We discuss further in Section 4.

Looking ahead, Section 5 shows that Flexo reduces circuit size by 75–87%, contributing to its 17% higher accuracy and 25× faster runtime compared to the prior state of the art [63]. In this section, we define our encoding (§3.1), then outline how this encoding makes error detection simple (§3.2) and how we optimize Flexo gates (§3.3) for accuracy.

3.1 Differential encoding

We use a classical differential encoding, which represents a logical boolean value (i.e., `true` or `false`) as the difference of two wire values (recall that our wire values are stored in weird registers; §2.2). One wire suffices to encode a boolean value: when the wire has a high value (H), we usually say it encodes `true`, and likewise for a low value (L) and `false`. In contrast, a differential encoding represents a boolean value w with two wire values, w_- and w_+, such that $w_- \neq w_+$. To represent $w =$ `false`, we choose $w_-w_+ =$ HL, and likewise $w =$ `true` is represented as $w_-w_+ =$ LH. Since there are four possible states for the combination of two wires (i.e., $w_-w_+ \in \{\mathsf{HH}, \mathsf{HL}, \mathsf{LH}, \mathsf{LL}\}$), w_-w_+ can also represent two *invalid* states (i.e., when $w_- = w_+$).

Representing boolean functions with differential encodings Consider a boolean function $o = f(i_1, \ldots, i_N)$, where $o, i_1, \ldots, i_N \in \{\texttt{true}, \texttt{false}\}$. We can think of a differential gate computing f as a pair of boolean expressions, one that computes each of the differential output wires, i.e., $o_- = f_-(i_{1-}i_{1+}, \ldots, i_{N-}i_{N+})$ and $o_+ = f_+(i_{1-}i_{1+}, \ldots, i_{N-}i_{N+})$, where $f_+ = f$ and $f_- = \neg f$.

Deriving expressions for f_+ and f_- is straightforward: first, expand the logical expressions for f and $\neg f$ to minterm canonical form (i.e., to a sum of products). Next, replace each non-negated variable v with that variable's v_+ wire, and each negated variable w with that variable's w_- wire. The result is a pair of inversion-free expressions that compute o_-o_+. Logical manipulations like these are used extensively by electronic design automation (EDA) toolchains, e.g., tools used to program field-programmable gate arrays (FPGAs) [19, 79, 129]; our compiler (§4) uses Yosys [129] for this procedure.

As a concrete example of differential encoding, consider the AND gate $o = f(i_1, i_2) = i_1 \wedge i_2$. We have that $\neg f(i_1, i_2) = \neg(i_1 \wedge i_2) = \neg i_1 \vee \neg i_2$, where the latter expression is the minterm canonical form, meaning that the following pair of

```
Out_p[In1_p[In2_p[0]]] = 0;
Out_m[In1_m[0]] = 0;
Out_m[In2_m[0]] = 0;
```

Listing 4: A differential μWM for AND. o_+ and o_- correspond to Out_p and Out_m, and likewise for i_1, In1 and i_2, In2.

```
Out_p[In1_p[In2_m[0]]] = 0; // XOR(1, 0) = 1
Out_p[In1_m[In2_p[0]]] = 0; // XOR(0, 1) = 1
Out_m[In1_p[In2_p[0]]] = 0; // XOR(1, 1) = 0
Out_m[In1_m[In2_m[0]]] = 0; // XOR(0, 0) = 0
```

Listing 5: A differential μWM for XOR. o_+ and o_- correspond to Out_p and Out_m, and likewise for i_1, In1 and i_2, In2.

expressions implements the differential gate in terms of the differential inputs: $o_+ = i_{1+} \wedge i_{2+}$ and $o_- = i_{1-} \vee i_{2-}$. Listing 4 shows our implementation of gate as a μWM using the basic AND and OR gadgets from Listing 1 (§2.2).

As a slightly more complex example, consider XOR, i.e., $o = f(i_1, i_2) = (i_1 \vee i_2) \wedge \neg(i_1 \wedge i_2)$. Following the process above, we have $o_+ = (i_{1+} \wedge i_{2-}) \vee (i_{1-} \wedge i_{2+})$, $o_- = (i_{1+} \wedge i_{2+}) \vee (i_{1-} \wedge i_{2-})$. Listing 5 shows our implementation of this gate, which computes XOR in a *single* transient execution—something that no prior work has been able to achieve.

How many inputs can Flexo support? As with all μWMs, the complexity limit for Flexo gates is dictated by the length of the CPU's transient window. Compared to prior work, however, Flexo is much less sensitive to the logical function being calculated. In particular, prior work uses a cascade of gates to implement functionality that inverts inputs or intermediate states (§2.2). In most cases, such compositions span multiple transient windows, reducing speed and accuracy. In contrast, as described above, Flexo's differential encoding lets it compute logical functions directly as a sum of products of the differential input values.

Speaking generally, the cost to evaluate a sum of products depends on the number of terms and the number of variables per term. Every N-to-1 boolean function f has at most 2^N minterms in total between the f_+ and f_- expressions (i.e., at most one term per truth table entry), where each term depends on at most N variables (i.e., an input or its negation).[4] The XOR gate (and its generalization to N inputs, which computes the sum of inputs mod 2) is an example of a maximal-cost gate: for any N, N-input XOR has 2^N minterms in total between f_+ and f_-, with N variables per minterm.

[4] A simple counting argument shows that, as N grows, most N-to-1 functions must be close to the maximum size. This is why we consider generic N-to-1 functions: while it might be possible to squeeze a bit more performance out of Flexo by considering a subset of, say, $(N+1)$-to-1 functions, most of the time we expect that the payoff will be small. We leave detailed investigation and further optimization to future work.

Thus, if a given CPU supports Flexo's XOR on N inputs, Flexo can compute any N-input function. Our experiments show that the eight x86_64 testbed architectures on which we evaluate (§5, Table 1) can correctly evaluate at most a 4-input XOR (and thus, any 4-input boolean function) in a single transient execution. Other microarchitectures may differ.

3.2 Bit-wise error detection and correction

Flexo's differential encoding gives bit-wise error detection "for free," and lets us build accurate error correction that does not impose high overhead. Prior work [38, 63] implements error correction with majority-of-five voting, which requires re-running a circuit five times to produce a final, error-corrected output. In this approach, however, error correction does not scale with the accuracy of the underlying circuit: it imposes unnecessary overhead for already-accurate circuits and may not run enough error-correction rounds to correct inaccurate circuits. Flexo overcomes these limitations by dynamically re-executing a circuit *only* when it detects an error. Section 5 shows that this approach has small overhead ($\approx 1.8\%$) for circuits with good accuracy yet can correct extremely low accuracy—in one case, improving a circuit with 0.3% uncorrected accuracy to 99.9%.

Error detection Since our differential encoding only allows two valid output states (HL and LH), the other two states (HH and LL) signal an error in the circuit's execution. This can happen when a prefetcher fetches both wires in the cache, or when both wires are evicted from the cache. While all invalid states indicate an error, not all errors cause invalid states. Immediately below, we discuss the way we overcome this limitation with a careful application of majority voting.

Error correction When Flexo detects an error in the output bits of a circuit, it needs to re-run that circuit to correct the error. The naive approach is to re-run the circuit until a single run produces a completely valid set of output bits; for a three-output circuit, this means that one run must produce three valid outputs. For better performance, though, we save the results of every valid output bit over the repeated circuit executions—and only re-run the circuit until there *exists* a valid bit for every output. If one run of a three-output circuit produces two valid bits and another produces the third, for example, we can stop re-running the circuit: we have saved three valid output bits over the course of the two runs.

While differential encoding can detect errors leading to invalid outputs HH or LL, it cannot detect errors that lead to valid (but incorrect) outputs. This could happen, for example, if the overall result of a computation should be HL, but some errors cause the computation to incorrectly output LH. To address this problem, we run a majority vote for each output bit in the circuit. On each run, for each valid output bit, we record whether that bit is zero or one. Once each output is either zero or one $> 50\%$ of the time, we return the majority's

choice for each output and stop re-running the circuit.[5] In contrast to prior work, this means that in principle our circuits' runtime is unbounded. In practice, however, a circuit with reasonable accuracy ($> 10\%$) only needs a few—in our experiments, at most seven—re-runs to produce the output, while circuits with much lower accuracy ($< 1\%$) can incur much larger overhead (100–250×).

3.3 Optimizing the memory layout

Prior work shows that microarchitectural features such as the prefetcher affect μWM's accuracy. Katzman et al. [63] report that enabling the prefetcher reduces the accuracy of their circuits by 41–52%. Evtyushkin et al. [38] and Kaplan [62] also discuss the accuracy-related effects of weird registers' placement and alignment in memory.

These observations motivate us to optimize Flexo's memory layout to reduce these microarchitectural effects. For Flexo, the two most important factors for accuracy are (1) the spacing between successive weird registers (i.e., the size of the gaps between two weird registers' memory addresses), and (2) the mapping from circuit wires to cache lines. Our evaluation (§5) demonstrates that Flexo's accuracy remains high *without* disabling the prefetcher—in fact, Flexo gives higher accuracy with the prefetcher enabled than prior state of the art does with the prefetcher disabled [63].[6]

Setting the spacing between weird registers As noted above, the memory layout of weird registers is a key design parameter. If weird registers R1 and R2 have memory addresses that are too close together, the prefetcher might fetch (say) R2 into the cache as a side effect of the weird machine's fetching R1, thereby corrupting the value in R2. Similarly, if registers R1 and R2 both map to the same cache set in a set-associative cache, computing on one register might cause the other's eviction, which also leads to corruption.

Kaplan [62] suggests separating weird registers by 4160 ($4096+64$) bytes: a ≥ 4096-byte offset puts weird registers on different pages, defeating the prefetcher, and the additional 64-byte offset ensures that most weird registers map to a different cache set. But a separation this large is not suitable for large circuits: when the number of active pages exceeds the size of the translation lookaside buffer, the resulting page table thrashing adds more noise to the circuit's execution. In our evaluation (§5), we use 1088-byte ($1024+64$) gaps for the smallest circuit (the 4-bit ALU). For the larger circuits (SHA-1, AES, and Simon), which require more weird registers, we use either 576- or 488-byte (512 ± 64) gaps.

[5] Flexo's compiler (§4) makes it easy to tune runtime versus accuracy by specifying the number of votes required when choosing an output value.

[6] Requiring the prefetcher to be disabled makes a μWM less practical: modern CPUs' prefetchers can only be disabled with root or BIOS access, and the prefetcher is generally enabled by default because it boosts performance.

Randomizing weird register memory layout To reduce the effect of the prefetcher for large circuits (which, as discussed immediately above, must use smaller register-to-register gaps), Flexo also randomizes the mapping of weird registers to memory addresses before *each* execution.

To do so, Flexo accesses weird registers via an indirection table $\mathcal{R}$ defining a permutation on $\{0,\ldots,k-1\}$ for k weird registers. When the gap between weird registers is g bytes and the base memory address of the weird register file is b, the address of (zero-indexed) weird register j is given by $b+g\cdot\mathcal{R}(j)$. $\mathcal{R}$ is implemented as a length-k array of integers; before execution, Flexo shuffles $\{0,\ldots,k-1\}$ into the array.

4 A compiler for Flexo

To execute a program using a μWM, a programmer must express that program as a boolean circuit. Transforming a high-level computation into a (well-performing) circuit and implementing that circuit as a μWM is both difficult and error-prone—especially for complex programs like cryptographic computations. To address this, we build the first compiler that generates μWMs. Our compiler consumes C or C++ and targets the differential encoding presented in Section 3; as we discuss below, this is an easier target than prior encodings.

Our compiler makes it easy to create efficient weird machines. For example, the source code for our SHA-1 hash function is 274 lines of code, whereas prior work [63] requires around two thousand lines to implement the same circuit from low-level logic gates. This convenience does not sacrifice performance: our compiler-generated implementation of one SHA-1 round is about 25× faster than prior work (§5). As further evidence that μWM compilers can improve both performance *and* usability, we implement three proof-of-concept optimization passes and configuration hooks:

- Error detection hooks let us build Section 3's error correction mechanism in 21 lines of code.
- The compiler lets us configure the length of the transient window (§2.1), which makes it easy to fine-tune μWMs for a given microarchitecture.
- An optimization pass makes generated circuits more resilient to prefetching and cache eviction.

In this section, we outline prior work that inspired our compiler, and then give an overview of our compilation pipeline. Our compiler is implemented in 2245 lines of C++ code.

Background: circuit compilers and high-level synthesis While prior work suggests building compilers that target μWMs [125], this work, to our knowledge, is the first such implementation. Still, compiling from programming languages to circuits is an important research topic in several distinct fields. High-level synthesis (HLS) creates a hardware description from a programming language (often, C or C++);

see [32] for a survey. In contrast to our compiler, which produces purely combinational circuits (discussed below), the key challenge in HLS is efficiently handling stateful circuits (e.g., flip-flops or latches). HLS techniques may nevertheless prove valuable in future work (§7), particularly with regard to optimization.

Further afield, compilers target the circuit-like representations [91] used by cryptographic primitives like multi-party computation [2, 14, 20, 24, 55, 56, 81], fully homomorphic encryption [121], and probabilistic proof systems [18, 29, 70, 93, 106, 107, 122]. These compilers transform high-level programs into boolean or arithmetic circuits, which, as in μWMs, are almost always purely combinational. Similarly, program verifiers [9, 10, 25, 27, 28, 30, 61, 65, 71, 74, 112, 119, 120, 131] target logical formulas (which are isomorphic to combinational circuits). Both verifiers and cryptographic compilers allow programmers to use control flow (via standard flattening techniques (e.g., as in [25])) and memory operations (via domain-specific representations). Future μWMs may build on such techniques to support a richer input language.

μWM compiler overview Figure 1 shows the process by which our compiler constructs a weird machine from a C or C++ function. First, the programmer provides an input file and annotates the functions that should be compiled to weird machines. Each annotated function is processed by an LLVM pass that translates it into a Verilog [1] description. This pass treats function bodies as the circuit's computation, function arguments as the input and output wires of the circuit, and the function's return value as an error detection bit (§4.1).

The compiler uses Yosys [129], an off-the-shelf EDA tool, to synthesize the Verilog into a set of truth tables with at most 4 inputs. For each such truth table, the compiler generates an assembly implementation of a Flexo gate. Finally, the compiler optimizes the memory layout of the weird machines and splices them into the original LLVM IR file. Appendix A shows the output of each compilation stage.

Input program requirements Since μWMs are ultimately boolean circuits, compiler input programs must not contain:

- *Intermediate loads or stores*: The compiler only permits load operations on input variables and store operations of output variables, with constant load or store addresses.
- *Control flow*: Function bodies cannot contain branches, function calls, or loops.
- *Unsupported operations*: Only arithmetic, comparison, logical, bitwise, assignment, and ternary operators are allowed (e.g., no address-of operator).

Future work can avoid these restrictions using known techniques (see above). Even with these limitations, our compiler supports complex computations like SHA-1 and AES (§5).

Finally, our compiler implements special semantics for arguments and return values of the functions it translates. A

```
#define ROL(x, n) (((x) << (n)) | ((x) >> (32 - (n))))

void __weird__sha1_round(
    unsigned* input,
    unsigned* output, unsigned* error_output
) {
    unsigned a = input[0];
    unsigned b = input[1];
    unsigned c = input[2];
    unsigned d = input[3];
    unsigned e = input[4];
    unsigned w = input[5];
    unsigned k = 0x5A827999;

    unsigned f = (b & c) | ((~b) & d);
    unsigned temp = ROL(a, 5) + f + e + w + k;

    output[0] = temp;
    output[1] = a;
    output[2] = ROL(b, 30);
    output[3] = c;
    output[4] = d;
}
```

Listing 6: An example input program to our compiler. This program implements one round of SHA-1.

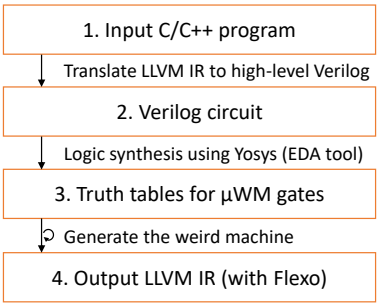


Figure 1: The compiler pipeline. In the first stage, the compiler executes clang's frontend with the `-emit-llvm` flag to emit LLVM IR code.

μWM's input values are function arguments (which can be pointers or integral types), as are its outputs (which must be pointers; see §4.1). Specifying a `bool` function return type directs the compiler to emit extra error detection code: the function returns true if any output wire yields an error.

4.1 The compilation pipeline

Our compiler converts input programs (e.g., Listing 6) into boolean circuits in several pipeline stages, shown in Figure 1.

From LLVM IR to Verilog circuit In this stage, the compiler translates LLVM IR into a high-level Verilog circuit. It treats function arguments as input and output buses; an input is an argument whose value is only ever *read*, and an output is one whose value is only ever *written*. The restric-

tions on input programs discussed above make the LLVM IR to Verilog conversion step straightforward. For example, the compiler converts arithmetic instructions into the corresponding Verilog operators and converts memory instructions into assignments between wires or buses.

In addition to specifying a global boolean return value that is set if an error occurred, the programmer can specify error bits for individual outputs, which can be used to implement custom error correction. To implement the error correction described in Section 3, for example, we write code that checks each output's error flag, saves valid values, and conditionally re-executes the circuit.

From Verilog to truth tables Our compiler guarantees that the Verilog it generates encodes purely combinational logic (i.e., free of memory elements like flip-flops and latches). The Verilog can thus be translated into truth tables using standard logic synthesis techniques [101, Ch. 12]. Our compiler uses Yosys [129], an open-source synthesis tool that targets FPGAs; this is convenient because Flexo's arbitrary 4-input gates (§3) closely resemble an FPGA's lookup tables (LUTs). The output from Yosys is a set of 4-input truth tables, which would normally be used to program an FPGA's LUTs; our compiler instead uses them to construct a μWM.

We note that Flexo's ability to implement any 4-input boolean function makes this step substantially easier while improving the quality of the result. In principle, one could use Yosys to target the set of gates supported by prior work (e.g., `AND`, `OR`, and `NOT` [38, 62, 63, 125]). However, extracting good performance from such a circuit would be challenging because it would require the synthesis process to account for the speed, size, and accuracy of a composition of heterogeneous gates. In contrast, the performance of Flexo's 4-input gates is essentially oblivious to the logical function being computed, and its basic unit of computation is much richer than prior work's. This dramatically simplifies the synthesis process and completely eliminates a (difficult) optimization pass.

From truth tables to Flexo The final pipeline stage converts the prior stage's truth tables into Flexo gates and emits an assembly implementation of the μWM. To do so, the compiler uses the Quine-McCluskey logic minimization algorithm [83] to construct the f_+ and f_- expressions as sums of products, then implements those expressions using basic `AND` and `OR` gadgets (§2.2; Listings 4 and 5, §3.1).

The compiler then generates assembly to prepare the circuit's inputs (i.e., by transferring the values from architectural to microarchitectural state), to read the circuit's outputs into architectural state (i.e., by measuring cache residency information for each output value), and to trigger transient execution for each gate (via return address misprediction; §2.1). The compiler's approach to triggering transient execution gives it precise control over the length of the transient window. This is a parameter that the user can set, giving a convenient knob for tuning the μWM to a particular microarchitecture.

The compiler optimizes the μWM's memory layout to improve accuracy in the face of prefetching and cache evictions (§3.3): it controls the spacing between successive weird registers (to avoid prefetching and correlated evictions), and emits code that randomizes weird registers' memory locations before each execution (to further counteract the prefetcher). Finally, the compiler splices the optimized assembly code into the LLVM IR file and generates a standard executable.

5 Evaluation

We now evaluate Flexo via the following questions:

1. How does Flexo compare to state-of-the-art prior work in speed and accuracy? (§5.1)

2. How large of a Flexo circuit can execute correctly across different microarchitectures? (§5.2)

3. How does Flexo perform on larger computations (i.e., SHA-1 and AES) using composed circuits? (§5.3)

We find that Flexo significantly reduces the circuit size and performs one to two orders of magnitude faster than the state of the art. Furthermore, Flexo is the first μWM that can perform reliably ($\geq$97.8% accuracy) across different microarchitectures even for complex computations like SHA-1 and AES, allowing us to build an end-to-end application: a weird machine–based packer (§6).

Throughout this section, we construct circuits for Flexo using the compiler described in Section 4. These circuits are compiled from at most a few hundred lines of C or C++, and constructing the circuits requires no manual edits. This stands in stark contrast to prior work, in which all circuits must be hand crafted by experts.

Testbeds We evaluate Flexo on eight different AWS EC2 machine types (Table 1), all running Ubuntu 22.04; we use one of these machines (Skylake) to reproduce the results of state-of-the-art prior work (§5.1). Our testbed machines cover several of the latest x86_64 microarchitectures from AMD and Intel. (In the rest of this section we refer to these machines using their microarchitectures, e.g., "Skylake.")

Because we run our experiments in a public cloud environment, other tenants are able to inject noise into the CPU's shared microarchitectural states. While our experiments suggest that this noise has little effect on our results (see Appendix B for details), achieving high accuracy on shared machines increases our confidence that Flexo is suitable for deployment in real-world settings.

Experimental method To measure Flexo's runtime and accuracy on a given circuit, we compile (§4) a binary that evaluates the circuit one thousand times on random inputs and reports the number of correct executions and total time. We then repeatedly execute that binary for forty-three hours

Microarchitecture	Instance Type	Processor
Zen 1	t3a.xlarge	AMD EPYC 7571
Zen 2	c5a.xlarge	AMD EPYC 7R32
Zen 3	c6a.xlarge	AMD EPYC 7R13
Zen 4	m7a.xlarge	AMD EPYC 9R14
Skylake	c5n.xlarge	Intel Xeon 8124M
Cascade Lake	m5n.xlarge	Intel Xeon 8259CL
Icelake	m6in.xlarge	Intel Xeon 8375C
Sapphire Rapids	m7i.xlarge	Intel Xeon 8488C

Table 1: The AWS EC2 instances and processors we use to perform our evaluation.

without error correction and one hundred hours with error correction, and then we report the median from these executions.

This measurement approach is similar to prior work. Running the same circuit repeatedly amortizes away constant measurement offsets. Reporting the median of several executions avoids outlier cases where the μWM fails because of an unlucky combination of microarchitectural and architectural state (e.g., page alignment, cache state, noise from other processes or tenants, etc.).

When preparing a binary for a given microarchitecture, we first run a sweep to determine the transient window length that gives the best accuracy or the shortest runtime. We build circuits with fifty different transient window lengths by adjusting the number of division instructions used when triggering transient execution (§2.1). For circuits without error correction, we choose the transient window length that gives the best accuracy; for circuits with error correction, we choose the one with the shortest runtime among those with an accuracy of at least 95%. This process does not require expert hand-tuning, but as in prior work it means that our binaries are specific to a target microarchitecture. In Section 6, we show that a single μWM-enhanced packer binary works on all the Intel machines from our testbed, meaning that non–microarchitecture-specific binaries can still have good performance; in Section 7, we discuss ideas to further enhance our compiler's ability to create generic, multi-microarchitecture binaries.

5.1 How does Flexo compare to prior work?

We compare the accuracy and execution time of our circuits to those from Gates of Time (GoT) [63], state-of-the-art prior work building μWMs. We find that, across a variety of circuits, Flexo performs approximately 25–49× faster and 4–17% more accurately than GoT with error correction, and approximately 5–10× faster and 18–39% more accurately than GoT without error correction.

Reproducing GoT's results We evaluate both projects on the Skylake machine. Although we do not use exactly the same CPU as GoT (their CPU is unavailable on AWS EC2), our CPU uses the same core microarchitecture as the one used in their evaluation. Further, our reported results for GoT are similar to those reported in GoT. Specifically, Tables 2 and 3 give similar results to [63, §4.1–4.2] with prefetchers enabled.

4-bit ALU

	GoT [63]	Flexo	Improvement
Wires	336	62	-81.55%
Gates	250	32	-87.20%
Accuracy	81.00%	98.90%	+17.90%
Runtime (μs)	134.37	13.58	9.89×
With error correction			
Accuracy	96.00%	100.00%	+4.00%
Runtime (μs)	673.48	13.82	48.73×

Table 2: Comparison of the circuit size, accuracy, and runtime of GoT [63] and Flexo for the 4-bit ALU.

SHA-1 round function

	GoT [63]	Flexo	Improvement
Wires	2976	1061	-64.35%
Gates	2208	544	-75.36%
Accuracy	57.00%	95.80%	+38.80%
Runtime (μs)	1161.91	219.12	5.30×
With error correction			
Accuracy	83.00%	100.00%	+17.00%
Runtime (μs)	5771.13	233.32	24.73×

Table 3: Comparison of the circuit size, accuracy, and runtime of GoT [63] and Flexo for one round of SHA-1.

We evaluate two circuits from GoT: a 4-bit arithmetic logic unit (ALU) and one round of the SHA-1 hash function. We choose these computations because they are of practical interest and cover a range of circuit sizes (SHA-1 is about 10× larger than the 4-bit ALU). To evaluate GoT's performance, we execute the shell script in their artifact,[7] reporting median accuracy and runtime. We compare both error-corrected and non-error-corrected GoT and Flexo circuits.

Results without error correction The top portion of Table 2 compares GoT's 4-bit ALU performance to that of Flexo. Flexo reduces the circuit size by 87.20% compared to GoT, which leads to a 9.89× speedup. At the same time, Flexo improves the circuits' accuracy from 81% to 98.9% without relying on error correction or detection. Flexo also reduces the SHA-1 circuit size by 75.36% (with a 5.3× speedup) and increases accuracy by 38.8% (Table 3).

[7] https://github.com/0xADE1A1DE/GoT/blob/main/circuits/run_experiment_all_with_prefetcher.sh. This script executes the circuit one hundred times to measure accuracy and runtime, and repeats this ten thousand times to collect the median.

Results with error correction The bottom portions of Table 2 and 3 compare error-corrected GoT to error-corrected Flexo. Table 2 and 3 show that, with error correction, Flexo achieves perfect accuracy with tiny performance penalty. Our bit-by-bit error correction mechanism (§3.2) increases the accuracy while maintaining high performance, yielding only 1.76% (ALU) or 6.48% (SHA-1) longer runtime.

While error-corrected GoT also enjoys much higher accuracy, its runtime is 48.73× and 24.73× slower than Flexo for ALU and SHA-1, respectively. This is because GoT implements error correction using majority voting: it executes the same circuit five times, and then outputs the most common result. While this improves the circuit's accuracy, it also increases the runtime by ≈5× compared to the un-corrected version. In contrast, Flexo re-runs the circuit (correction) *only* when it detects an error (§3.2), and thus incurs minimal overhead when the circuit is already accurate.

5.2 How large can a Flexo circuit be?

Scaling μWMs to large circuits is challenging. As the circuit grows, it needs more storage for the values of wires and may start incurring self-evictions in the cache. Using more cache lines might also increase interference from hardware prefetchers. This results in less accurate circuits. For example, when the circuit in Section 5.1 grows by ≈10×—from a 4-bit ALU to the SHA-1 circuit—GoT's accuracy degrades by 24%.

In this section, we show that Flexo's design can handle much larger circuits than those in prior work. We do so by executing four circuits across a range of sizes: in addition to the simple 4-bit ALU and the first round of SHA-1 from the prior section, we evaluate circuits implementing one round of AES encryption and the entire Simon block cipher [12], which are five and eight times larger than the SHA-1 circuit, respectively. We evaluate the accuracy and runtime of Flexo on the eight machines in Table 1, finding that Flexo supports a wide range of CPU microarchitectures and that its error correction mechanism dramatically increases accuracy—in one case, from under 0.3% to over 99.9%.

Circuit implementation The ALU and SHA-1 implementations are the same ones reported in Section 5.1.

To build one round of AES encryption with a 128-bit key, we implement its four operations, which are *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey* (214 lines of C). The most challenging operation is *SubBytes*, which requires computing a multiplicative inverse over GF(2^8). While *SubBytes* is usually implemented as a lookup table (i.e., the AES S-box), this approach would be inefficient in Flexo. We instead use the method of Canright et al. [23] and Satoh et al. [102] to compute a multiplicative inverse directly.

We also implement Simon [12], a lightweight block cipher designed for microcontrollers. We use the variant with 32-bit blocks and a 64-bit key. Because this cipher is so simple, we can execute the entire cipher in one circuit (50 lines of C).

Results Table 4 shows the results of the four test circuits on our eight testbed machines. We find that Flexo can compute large circuits (> 4k gates) with very high accuracy (≥ 99.3%) on all testbed machines, demonstrating that it supports complex computations that prior μWMs could not handle. As Tables 2 and 3 demonstrate, Flexo's circuit construction (§3) results in ≈4–8× smaller circuits than prior work, meaning that a 4k-gate Flexo circuit supports significantly more complex functionality than prior work with that number of gates.

For ALU and SHA-1, most of our testbed machines give high accuracy even without error correction (EC). The AES circuit, in contrast, is ≈5× as large as SHA-1 and its accuracy is much lower. But even when its accuracy is as low as 1.4% (on Zen 3 and Zen 4), EC can still correct all errors at the cost of longer runtime (≈7×).

Simon is the largest circuit, and most AMD CPUs give very low accuracy. For Zen 3, we must reduce the circuit size from 4322 gates to 3282 gates by reducing the number of rounds in the cipher (from 32 to 25) in order to generate correct outputs. In other words, the maximum size of a Flexo circuit for Zen 3 is ≈3k gates. Beyond this size accuracy drops to zero, at which point error correction cannot help. Other Zen machines still give nonzero accuracy when computing the full 32 rounds. In these cases, even when accuracy is as low as 0.3%, error correction still improves it to 99.9%, although the performance penalty is high (≈298×).

5.3 How does Flexo perform on composed circuits?

A standard technique for running large computations in μWMs is to stitch several smaller circuits together via *architectural* computations. While this approach is helpful, it does not entirely address the problem of scaling μWMs to large computations, for two reasons. First, it exposes more of the computation in architectural states, which may increase the likelihood of detection—so fewer stitches (i.e., larger μWM circuits; §5.2) is better for stealth. Second, as a natural consequence of composing circuits with imperfect accuracy, the accuracy and/or speed of the overall computation suffers.

In this section, we evaluate Flexo's accuracy and runtime when composing several circuits. In particular, we use the single round SHA-1 and AES circuits as building blocks to compose a complete SHA-1 hash function and AES encryption. As in prior work, we stitch these circuits together by using architectural computation to transform the outputs of one circuit into the inputs of the next one.

Results Table 5 shows the results of the SHA-1 hash function with two input blocks and the AES encryption with one block and a 128-bit key. The SHA-1 hash function is composed of 160 small circuits (one circuit per round, 80 rounds per block), while the AES encryption comprises 19 small circuits (ten rounds and nine round keys).

Circuit		4-bit ALU		SHA-1 (1 round)		AES (1 round)		Simon (1 block)	
Size		Wires	Gates	Wires	Gates	Wires	Gates	Wires	Gates
		62	32	1061	544	4990	2524	8288	4322
Machine	EC	Accuracy	Runtime	Accuracy	Runtime	Accuracy	Runtime	Accuracy	Runtime
Zen 1	✘	97.90%	14.74	76.40%	230.30	5.50%	1058.65	0.40%	1808.68
	✓	100.00%	15.59	100.00%	314.57	100.00%	10152.12	99.90%	432834.08
Zen 2	✘	99.00%	14.24	97.20%	228.43	5.80%	953.82	8.60%	1582.57
	✓	100.00%	14.32	100.00%	235.29	100.00%	5228.24	99.90%	21105.43
Zen 3	✘	99.50%	12.44	68.90%	199.47	1.40%	843.91	0.60%*	1042.26*
	✓	100.00%	12.41	100.00%	255.20	100.00%	5732.02	99.30%*	102473.62*
Zen 4	✘	99.50%	13.53	71.50%	219.43	1.40%	942.37	0.30%	1525.78
	✓	100.00%	13.68	100.00%	302.44	100.00%	7069.24	99.90%	455014.59
Skylake	✘	98.90%	13.58	95.80%	219.12	10.30%	1167.93	22.30%	2851.97
	✓	100.00%	13.82	100.00%	233.32	100.00%	7922.10	99.90%	14028.95
Cascade Lake	✘	98.70%	13.46	95.90%	219.78	9.40%	1072.33	25.60%	2070.49
	✓	100.00%	13.64	100.00%	230.21	100.00%	5038.20	99.90%	7955.66
Icelake	✘	92.50%	10.98	80.90%	167.22	10.90%	1416.23	34.30%	2431.28
	✓	100.00%	11.86	100.00%	208.10	100.00%	6540.76	99.90%	13177.12
Sapphire Rapids	✘	96.50%	14.20	83.90%	221.37	14.40%	1036.83	13.10%	1665.37
	✓	100.00%	14.82	100.00%	265.29	100.00%	4196.98	99.90%	11014.05

Table 4: The accuracy and runtime of Flexo with or without error correction (EC) on different CPUs. The runtime is in microseconds. *For Zen 3, we reduce the number of encryption rounds from 32 to 25 since it cannot correctly compute Simon encryption with more than 25 rounds. The circuit for 25 rounds has 6528 wires and 3282 gates.

	SHA-1		AES	
	Acc.	Runtime	Acc.	Runtime
Zen 1	99.00%	55.95	99.80%	347.72
Zen 2	99.30%	45.07	99.70%	204.16
Zen 3	97.80%	46.85	99.90%	206.72
Zen 4	99.60%	53.84	100.00%	215.39
Skylake	99.60%	44.68	100.00%	265.91
Cascade Lake	99.80%	43.68	100.00%	218.68
Icelake	99.80%	38.51	100.00%	239.33
Sapphire Rapids	98.00%	48.28	99.80%	210.47

Table 5: The accuracy and runtime of the SHA-1 hash function and the AES encryption of Flexo with error correction on different CPUs. The runtime is in milliseconds.

We find that Flexo is able to achieve high accuracy ($\geq$ 97.8%) for both computations. For comparison, the GoT authors report 95.1% accuracy for two blocks (i.e., 160 one-round circuits) of SHA-1 [63, §4.2].[8] Runtime is also better than in prior work: composing 160 GoT SHA-1 round circuits with error correction (using the single-round runtime from §5.1) would be more than 20$\times$ slower than our SHA-1 results.

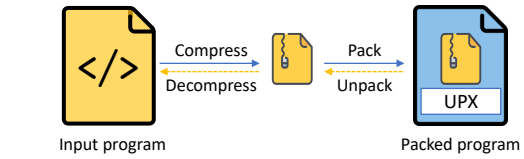
[8]The GoT measurement appears to require disabling the prefetcher to give better accuracy. As discussed in Footnote 6, this may not be possible for a user-level attacker because it requires root or BIOS access.

6 A weird machine packer

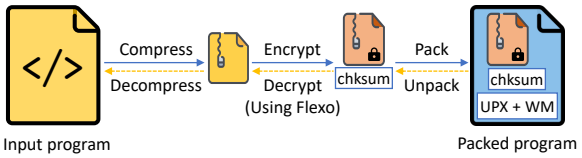
In this section we modify UPX [88], a binary packer [114] that is among the most widely used for malware [86], by obfuscating its unpacking process in order to make analysis more difficult. Our modified UPX uses a Flexo decryption stage during unpacking (and a corresponding encryption stage when packing; this is not obfuscated because it is never run on a victim machine) using Simon [12] or AES (§5.2).

Prior work [38, 125] also obfuscates malware with a μWM, but the obfuscation is limited to simple XOR operations and requires expertly hand-crafted μWM designs. Evtyushkin et al. [38] report that computing `XOR` on twenty bytes of data takes around 500 ms; Flexo takes less time to compute AES-CTR decryption on that amount of data. This work also demonstrates for the first time that using a μWM to obfuscate large computations is feasible for a non-expert: as mentioned in the prior section, our implementations of Simon and AES comprise 119 and 402 lines of C, respectively, and our compiler is able to automatically add it to UPX. We also modify the original UPX project with 412 lines of C++ to add an encryption stage during the encryption process and invoke the weird machine during unpacking.

Background: program obfuscation Program obfuscation makes a program more difficult to understand by increasing its complexity without changing its functionality [132]. Developers may use obfuscation to protect intellectual property [37], enforce software licensing [103], prevent tamper-



(a) UPX



(b) Our μWM packer (UPX + Flexo)

Figure 2: The packing (right arrows) and unpacking process (left arrows) of UPX and our weird machine packer.

ing [138], defeat anti-virus protection, or raise the cost of malware analysis [114].

Many program obfuscation techniques focus on making static and dynamic analysis more difficult. Anti-debugger, anti-sandbox, and anti-VM techniques let programs change behavior when they detect a dynamic analysis tool [108], and control flow flattening [124], junk code [75], and constant data encoding [132] make static analysis harder. Packers make malware less visible to static analysis tools by compressing an executable, then creating a new executable that contains the compressed data along with the code to unpack and run it.

These obfuscation techniques, however, happen in architectural states, and their computations and states are visible. While ExSpectre [123] aims to hide these tracks in transient executions, they can only move the computations into transient executions, and their states are still stored architecturally. In contrast, our Flexo-based packer hides both computation and program state in the microarchitecture.

μWM packers Figure 2 gives an overview of the packing and unpacking processes for our μWM packer. At a high level, unmodified UPX (Fig. 2a) compresses the original binary and creates a self-unpacking executable. Our μWM packer (Fig. 2b) makes three changes: first, it computes a set of checksums after compressing the data; second, it encrypts the compressed data; and third, in addition to the encrypted data, it injects the checksums and the decryption μWM into the self-unpacking executable. For the encryption scheme, users can choose AES or Simon in counter mode depending on their requirements.[9] Since Simon is simpler, it generally runs faster; AES, on the other hand, makes unpacking more complex, which may improve obfuscation.

During unpacking, the μWM decrypts the payload, then the unpacking code checksums the result, retrying decryption if there is a mismatch with the packed checksums. Finally, the UPX runtime decompresses the program and executes it.

The net result is that our μWM packer makes analysis even more difficult than standard packing. Performing unpacking steps in μWMs, for example, foils existing static analysis tools that depend on running a standard unpacker before analyzing code [16]. Flexo also breaks existing dynamic deobfuscation techniques that rely on emulation or single-stepping through the unpacking process. Such techniques interrupt transient execution, which causes unpacking to fail.

[9] Users can also utilize our compiler to implement a different encryption scheme for the packer.

	Simon	AES	Single binary
Zen 1	909.67	355.63	
Zen 2	64.66	242.17	
Zen 3	304.72	259.49	
Zen 4	2008.87	263.98	
Skylake	79.37	354.48	81.15
Cascade Lake	58.92	300.12	109.40
Icelake	146.58	306.12	146.58
Sapphire Rapids	50.13	268.35	60.14

Table 6: Our unpacker's median runtime in seconds using Simon and AES ciphers across the testbed CPUs (§6). The rightmost column is the runtime of the Icelake-specific binary using Simon on all Intel microarchitectures.

How fast is the unpacking process? We evaluate our packer on the `ls` binary from Ubuntu 23.04 (132 kiB uncompressed, 58 kiB compressed) using the testbed machines in Table 1. For each machine, our unpacker injects a Flexo decryption circuit with the same transient window length used in Section 5. We run the unpacker 100 times and calculate the median runtime, and Table 6 shows the results for Simon and AES. In summary, AES gives similar speed on all machines, roughly five minutes. On Zen 2 and Intel machines, Simon is faster: it finishes unpacking within eighty seconds, i.e., 2.1–5.3× faster than AES.

Can we create a cross-microarchitecture packed binary? The rightmost column of Table 6 shows the runtime of the Icelake-specific binary with Simon μWM on all four Intel microarchitectures. While the results are slower than with microarchitecture-specific binaries, the slowdown is always less than 2×, showing that Flexo can target multiple architectures simultaneously.

7 Conclusion and future work

This work started by asking whether μWMs should be considered a practical threat, or if they are likely to remain limited to simple computations and accessible only to a small group of experts. Our results show that μWMs have orders of magnitude more potential as a computational primitive than previously known, and that non-experts assisted by a compiler

can craft μWMs with state-of-the-art performance. History has shown that, beyond some minimum viability threshold, attack techniques tend to become more powerful over time. Considering both their extremely clever application to cache side-channel attacks in prior work [62, 63] and the performance and broad applicability demonstrated here, we believe μWMs are now well beyond that threshold. We thus conclude that μWMs merit further study, both to understand their limits and to develop detection methods and countermeasures. To that end, we offer a few specific suggestions:

Circuit optimizations Flexo circuits (and μWM circuits more generally) are very different from traditional circuits: the wires (weird registers) are volatile, which means their values are destroyed after a read, and the size of a logic gate is determined by the number of minterms of its boolean function. In contrast, traditional circuits can read from a wire multiple times without additional cost, and the size of a lookup table in FPGA circuits is determined by the number of inputs instead of the number of minterms. This raises interesting questions with regard to optimizations: (1) should we account for the cost of reusing wires in a circuit, and if so, how? and (2) how can we minimize the number of transient windows required by a weird circuit (e.g., by packing as many minterms as possible into each window)? Accounting for these special characteristics may result in much more efficient circuits.

Hiding and detecting weird machines Although μWMs frustrate program analysis by hiding computational states, behaving differently when debugged, and expressing high-level computations as low-level logic gates, their implementations are still visible in the program's code. While it would be a challenge, an analyst could reverse engineer the assembly code of a μWM to reconstruct the computation.

It may be possible to further hinder this type of analysis by hiding μWMs in dead code that is usually ignored by binary analysis tools—or even by camouflaging real μWMs with dummy ones. Of course, this is most helpful if the defender's goal is *understanding* the μWM—it is likely much easier to *detect* binaries containing μWMs (say, in a malware scanner). As μWM obfuscation becomes more of a threat, good detection and analysis techniques will become more important. These may include, for example, using hardware performance counters [33] and checking if a program's execution differs between normal execution and single-step debugging.[10]

Runtime calibration for transient windows To improve performance and accuracy, our compiler can generate binaries with different transient window lengths depending on the target microarchitecture (§4–§6). With additional engineering work, however, we could extend our compiler to produce binaries that calibrate transient window length at runtime. Our evaluation shows that this *one* parameter suffices to retarget Flexo across a wide range of microarchitectures, indicating that such runtime calibration is plausible because it does not involve a complex multi-parameter optimization.

Microarchitecture-specific μWMs In Section 6, we showed that Flexo binaries compiled for a specific microarchitecture can port to other microarchitectures. Such cross-microarchitecture binaries are useful because they do not require tailoring to individual machines, enabling attackers to build obfuscated, portable malware that can infect as many machines as possible. An intriguing alternative direction would be to explore the design of compilers for μWMs that work *only* on specific microarchitectures, by leveraging properties or optimizations that are unique to those microarchitectures. These microarchitecture-specific binaries would be useful for targeted attack scenarios, where an attacker may want their malware to activate only on the specific target machines. Such binaries could also raise the analysis cost by requiring analysts to examine them on specific microarchitectures.

[10] Recall that single-step debugging breaks μWMs, e.g., by affecting its transient execution and timing behaviors.

Acknowledgments

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

References

[1] Verilog. *IEEE Std 1364-2005*, 2006.

[2] Coşku Acay, Rolph Recto, Joshua Gancher, Andrew C. Myers, and Elaine Shi. Viaduct: An extensible, optimizing compiler for secure distributed programs. In *PLDI*, 2021.

[3] Onur Aciiçmez. Yet another microarchitectural attack: Exploiting i-cache. In *CSAW*, 2007.

[4] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *CT-RSA*, 2007.

[5] Onur Acıiçmez and Werner Schindler. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In *CT-RSA*, 2008.

[6] Onur Acıiçmez and Jean-Pierre Seifert. Cheap hardware parallelism implies cheap security. In *FDTC*, 2007.

[7] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *S&P*, 2019.

[8] Diego F Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. LadderLeak: Breaking ECDSA with less than one bit of nonce leakage. In *CCS*, 2020.

[9] Thomas Ball, Ella Bounimova, Vladimir Levin, Rahul Kumar, and Jakob Lichtenberg. The Static Driver Verifier research platform. In *CAV*, 2010.

[10] Thomas Ball, Vladimir Levin, and Sriram K Rajamani. A decade of software model checking with SLAM. *CACM*, 54(7):68–76, July 2011.

[11] Julian Bangert, Sergey Bratus, Rebecca Shapiro, and Sean W Smith. The page-fault weird machine: Lessons in instruction-less computation. In *WOOT*, 2013.

[12] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK lightweight block ciphers. In *DAC*, 2015.

[13] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, Frank Mckeen, Fangfei Liu, Ron Gabor, Christopher W Fletcher, Abhishek Basak, and Alaa Alameldeen. Speculative interference attacks: Breaking invisible speculation schemes. In *ASPLOS*, 2021.

[14] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In *CCS*, 2008.

[15] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting speculative execution through port contention. In *CCS*, 2019.

[16] Bitdefender. Technologies used in the antimalware engine. https://www.bitdefender.com/files/News/CaseStudies/study/351/Bitdefender-OEM-Antimalware-tech-eng-TechBrief-crea4549-210x297-en-EN-interactive.pdf, 2024.

[17] Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, and Anna Shubina. Exploit programming: From buffer overflows to "weird machines" and theory of computation. *login Usenix Mag.*, 36, 2011.

[18] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *SOSP*, 2013.

[19] Robert Brayton and Alan Mishchenko. ABC: An academic industrial-strength verification tool. In *CAV*, 2010.

[20] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. HyCC: Compilation of hybrid protocols for practical secure computation. In *CCS*, 2018.

[21] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant cpus. In *CCS*, 2019.

[22] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security*, 2019.

[23] David Canright. A very compact S-box for AES. In *CHES*, 2005.

[24] Edward Chen, Jinhao Zhu, Alex Ozdemir, Riad S Wahby, Fraser Brown, and Wenting Zheng. Silph: A framework for scalable and accurate generation of hybrid MPC protocols. In *S&P*, 2023.

[25] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS*, 2004.

[26] Shaanan Cohney, Andrew Kwong, Shahar Paz, Daniel Genkin, Nadia Heninger, Eyal Ronen, and Yuval Yarom. Pseudorandom black swans: Cache attacks on CTR DRBG. In *S&P*, 2020.

[27] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. In *ASE*, 2009.

[28] Lucas Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtik. JBMC: A bounded model checking tool for verifying Java bytecode. In *CAV*, 2018.

[29] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *S&P*, 2015.

[30] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C. In *SEFM*, 2012.

[31] Miles Dai, Riccardo Paccagnella, Miguel Gomez-Garcia, John McCalpin, and Mengjia Yan. Don't mesh around: Side channel attacks and mitigations on mesh interconnects. In *USENIX Security*, 2022.

[32] Luka Daoud, Dawid Zydek, and Henry Selvaraj. A survey of high level synthesis languages, tools, and compilers for reconfigurable high performance computing. In *ICSS*, 2013.

[33] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. Sok: The challenges, pitfalls, and perils of using hardware performance counters for security. In *S&P*, 2019.

[34] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A timer-free high-precision L3 cache attack using intel TSX. In *USENIX Security*, 2017.

[35] Stephen Dolan. mov is Turing-complete, 2013.

[36] Thomas Dullien. Weird machines, exploitability, and provable unexploitability. *IEEE Transactions on Emerging Topics in Computing*, 8(2), 2017.

[37] Shouki A. Ebad, Abdulbasit A. Darem, and Jemal H. Abawajy. Measuring software obfuscation quality–a systematic literature review. *IEEE Access*, 2021.

[38] Dmitry Evtyushkin, Thomas Benjamin, Jesse Elwell, Jeffrey A. Eitel, Angelo Sapello, and Abhrajit Ghosh. Computing with time: Microarchitectural weird machines. In *ASPLOS*, 2021.

[39] Dmitry Evtyushkin and Dmitry Ponomarev. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *CCS*, 2016.

[40] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *MICRO*, 2016.

[41] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Understanding and mitigating covert channels through branch predictors. *TACO*, 13(1), 2016.

[42] Stefan Gast, Jonas Juffinger, Martin Schwarzl, Gururaj Saileshwar, Andreas Kogler, Simone Franza, Markus Köstl, and Daniel Gruss. SQUIP: Exploiting the scheduler queue contention side channel. In *S&P*, 2023.

[43] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *JCEN*, 8(1), 2018.

[44] Daniel Genkin, Luke Valenta, and Yuval Yarom. May the fourth be with you: A microarchitectural side channel attack on several real-world applications of Curve25519. In *CCS*, 2017.

[45] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. ABSynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures. In *NDSS*, 2020.

[46] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *USENIX Security*, 2018.

[47] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *NDSS*, 2017.

[48] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A fast and stealthy cache attack. In *DIMVA*, 2016.

[49] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security*, 2015.

[50] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. Cache storage channels: Alias-driven attacks and verified countermeasures. In *S&P*, 2016.

[51] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games - bringing access-based cache attacks on AES to practice. In *S&P*, 2011.

[52] Berk Gulmezoglu, Andreas Zankl, M Caner Tol, Saad Islam, Thomas Eisenbarth, and Berk Sunar. Undermining user privacy on mobile devices using AI. In *CCS*, 2019.

[53] Yanan Guo, Xin Xin, Youtao Zhang, and Jun Yang. Leaky way: A conflict-based cache covert channel bypassing set associativity. In *MICRO*, 2022.

[54] Yanan Guo, Andrew Zigerelli, Youtao Zhang, and Jun Yang. Adversarial prefetch: New cross-core cache side channel attacks. In *S&P*, 2022.

[55] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: tool for automating secure two-party computations. In *CCS*, 2010.

[56] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure two-party computations in ANSI C. In *CCS*, 2012.

[57] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *S&P*, 2013.

[58] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES. In *S&P*, 2015.

[59] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross processor cache attacks. In *ASIACCS*, 2016.

[60] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, cross-VM attack on AES. In *RAID*, 2014.

[61] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods Symposium*, 2011.

[62] David Kaplan. Optimization and amplification of cache side channel signals, 2023.

[63] Daniel Katzman, William Kosasih, Chitchanok Chuengsatiansup, Eyal Ronen, and Yuval Yarom. The Gates of Time: Improving cache attacks with transient execution. In *USENIX Security*, 2023.

[64] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on the last level cache. In *DAC*, 2016.

[65] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. In *FAC*, 2015.

[66] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses, 2018.

[67] Ofek Kirzner and Adam Morrison. An analysis of speculative type confusion vulnerabilities in the wild. In *USENIX Security*, 2021.

[68] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *S&P*, 2019.

[69] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! Speculation attacks using the return stack buffer. In *WOOT*, 2018.

[70] Ahmed E. Kosba, Charalampos Papamanthou, and Elaine Shi. xJsnark: A framework for efficient verifiable computation. In *S&P*, May 2018.

[71] Daniel Kroening and Michael Tautschnig. CBMC–C bounded model checker. In *TACAS*, 2014.

[72] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical cache attacks from the network. In *S&P*, 2020.

[73] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, 2004.

[74] K. Rustan M. Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with Chalice. In *FOSAD*. 2009.

[75] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS*, 2003.

[76] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a way: Exploring the security implications of AMD's cache way predictors. In *ASIACCS*, 2020.

[77] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security*, 2018.

[78] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *S&P*, 2015.

[79] Jason Luu, Jeffrey Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Miad Nasr, Sen Wang, Tim Liu, Nooruddin Ahmed, Kenneth B. Kent, Jason Anderson, Jonathan Rose, and Vaughn Betz. VTR 7.0: Next generation architecture and CAD System for FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 7(2), 2014.

[80] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *CCS*, 2018.

[81] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay–A secure two-party computation system. In *USENIX Security*, 2004.

[82] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-cores cache covert channel. In *DIMVA*, 2015.

[83] Edward McCluskey. Minimization of Boolean functions. *Bell System Tech. J.*, 35(6), November 1956.

[84] Daniel Moghimi. Downfall: Exploiting speculative data gathering. In *USENIX Security*, 2023.

[85] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural data leakage via automated attack synthesis. In *USENIX Security*, 2020.

[86] Trivikram Muralidharan, Aviad Cohen, Noa Gerson, and Nir Nissim. File packing from the malware perspective: Techniques, analysis approaches, and directions for enhancements. *ACM Comput. Surv.*, 2022.

[87] Michael Neve and Jean-Pierre Seifert. Advances on access-driven cache attacks on AES. In *SAC*, 2006.

[88] Markus Oberhumer, Laszlo Molnar, and John Reiser. UPX: The ultimate packer for executables. https://upx.github.io/, 2024.

[89] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *CCS*, 2015.

[90] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA*, 2006.

[91] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. CirC: Compiler infrastructure for proof systems, software verification, and more. In *S&P*, 2022.

[92] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. Lord of the ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical. In *USENIX Security*, 2021.

[93] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *S&P*, 2013.

[94] Colin Percival. Cache missing for fun and profit. In *BSDCan*, 2005.

[95] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for cross-CPU attacks. In *USENIX Security*, 2016.

[96] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the observer effect for high-precision cache contention attacks. In *CCS*, 2021.

[97] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *USENIX Security*, 2021.

[98] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *S&P*, 2021.

[99] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. PACMAN: Attacking ARM pointer authentication with speculative execution. In *ISCA*, 2022.

[100] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *CCS*, 2009.

[101] Vivek Sagdeo. *The Complete Verilog Book*. Springer, New York, 1998.

[102] Akashi Satoh, Sumio Morioka, Kohji Takano, and Seiji Munetoh. A compact Rijndael hardware architecture with S-Box optimization. In *ASIACRYPT*, 2001.

[103] Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. Loki: Hardening code obfuscation against automated attacks. In *USENIX Security*, 2022.

[104] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. Keydrown: Eliminating software-based keystroke timing side-channel attacks. In *NDSS*, 2018.

[105] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*, 2019.

[106] Srinath Setty, Benjamin Braun, Victor Vu, Andrew J. Blumberg, Bryan Parno, and Michael Walfish. Resolving the conflict between generality and plausibility in verified computation. In *EuroSys*, 2013.

[107] Srinath T. V. Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J. Blumberg, and Michael Walfish. Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security*, 2012.

[108] Amit Sharma, Brij B. Gupta, Awadhesh Kumar Singh, and V.K. Saraswat. Orchestration of APT malware evasive manoeuvers employed for eluding anti-virus and sandbox defense. *Computers & Security*, 2022.

[109] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. Prime+ Probe 1, JavaScript 0: Overcoming browser-based side-channel defenses. In *USENIX Security*, 2021.

[110] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *USENIX Security*, 2019.

[111] Florian Sieck, Sebastian Berndt, Jan Wichelmann, and Thomas Eisenbarth. Util:: Lookup: Exploiting key decoding in cryptographic libraries. In *CCS*, 2021.

[112] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *POPL*, 2016.

[113] Daniël Trujillo, Johannes Wikner, and Kaveh Razavi. Inception: Exposing new attack surfaces with training in transient execution. In *USENIX Security*, 2023.

[114] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G. Bringas. SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *S&P*, 2015.

[115] John P. Uyemura. *CMOS Logic Circuit Design*. Springer, New York, 2001.

[116] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, 2018.

[117] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking transient execution through microarchitectural load value injection. In *S&P*, 2020.

[118] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *S&P*, 2019.

[119] Niki Vazou. *Liquid Haskell: Haskell as a theorem prover*. PhD thesis, UC San Diego, 2016.

[120] Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for haskell. In *ICFP*, 2014.

[121] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. SoK: Fully homomorphic encryption compilers. In *S&P*, 2021.

[122] Riad S. Wahby, Srinath Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS*, February 2015.

[123] Jack Wampler, Ian Martiny, and Eric Wustrow. ExSpectre: Hiding malware in speculative execution. In *NDSS*, 2019.

[124] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical report, University of Virginia, 2000.

[125] Ping-Lun Wang, Fraser Brown, and Riad S. Wahby. The ghost is the machine: Weird machines in transient execution. In *WOOT*, 2023.

[126] Zhenghong Wang and Ruby B Lee. Covert and side channels due to processor architecture. In *ACSAC*, 2006.

[127] Johannes Wikner and Kaveh Razavi. Retbleed: Arbitrary speculative code execution with return instructions. In *USENIX Security*, 2022.

[128] Johannes Wikner, Daniël Trujillo, and Kaveh Razavi. Phantom: Exploiting decoder-detectable mispredictions. In *MICRO*, 2023.

[129] Claire Wolf. Yosys open synthesis suite. https://yosyshq.net/yosys/, 2024.

[130] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *USENIX Security*, 2012.

[131] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using Boolean satisfiability. In *TOPLAS*, 2007.

[132] Hui Xu, Yangfan Zhou, Yu Kang, and Michael R. Lyu. On secure and usable program obfuscation: A survey, 2017.

[133] Mengjia Yan, Christopher W Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *USENIX Security*, 2020.

[134] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *S&P*, 2019.

[135] Yuval Yarom and Katrina Falkner. Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, 2014.

[136] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A timing attack on OpenSSL constant time RSA. *JCEN*, 7(2), 2017.

[137] Jiyong Yu, Aishani Dutta, Trent Jaeger, David Kohlbrenner, and Christopher W. Fletcher. Synchronization storage channels (S2C): Timer-less cache side-channel attacks on the Apple M1 via hardware synchronization instructions. In *USENIX Security*, 2023.

[138] Qiang Zeng, Lannan Luo, Zhiyun Qian, Xiaojiang Du, Zhoujun Li, Chin-Tser Huang, and Csilla Farkas. Resilient user-side Android application repackaging and tampering detection using cryptographically obfuscated logic bombs. *IEEE Transactions on Dependable and Secure Computing*, 2021.

[139] Ruiyi Zhang, Taehyun Kim, Daniel Weber, and Michael Schwarz. (M)WAIT for it: Bridging the gap between microarchitectural and architectural side channels. In *USENIX Security*, 2023.

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

[141] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In *CCS*, 2014.

A Compiler input and output

Listing 7 shows how the compiler translates an 8-bit `XOR` from C code (a) into: LLVM IR (b); a Verilog circuit (c); truth tables (d); and assembly (e).

B How does noise from co-tenants on shared machines affect Flexo?

In Section 5, we evaluated the performance of Flexo's circuits by repeatedly running each binary for 100 hours in the AWS EC2 public cloud environment. To assess the impact of noise due to other tenants who may have been co-located with our instances, we now analyze if the runtime and accuracy of our binaries changed during that time window.

Figures 3 and 4 show the accuracy and runtime of Flexo over the 100-hour time window. For all these plots, each data point is the moving median of ten accuracy/runtime samples.

From these plots, we can see that most circuits are very stable on all the machines. We only see temporal variations on the AMD Zen 1 and Zen 3 machines. For AMD Zen 1, the accuracy for all circuits is stable. However, the runtime of the Simon and AES circuits exhibits notable variances, displaying what appears to be a periodic pattern occurring approximately every 24 hours. For AMD Zen 3, the Simon and SHA-1 circuits have larger variances in accuracy, and the Simon and AES circuits have larger variances in runtime. Nevertheless, the accuracy of these circuits is still very high (usually around 97–98%) even in the presence of noise.

```
bool __weird__xor(
    uint8_t in1, uint8_t in2, uint8_t* out
) {
    out[0] = in1 ^ in2;
    return false;
}
```

(a) Input C/C++ program

```
define i1 @__weird__xor(
    i8 %in1, i8 %in2, i8* %out
) {
entry:
    %xor3 = xor i8 %in2, %in1
    store i8 %xor3, i8* %out
    ret i1 false
}
```

(b) Input LLVM IR

```
module __weird__xor(
    input [7:0] _in1, input [7:0] _in2,
    output [7:0] _out
);
    wire [7:0] _xor;

    assign _xor3 = _in2 ^ _in1;
    assign _out[7:0] = _xor3;
endmodule
```

(c) Verilog circuit

```
.model __weird__xor
.inputs _in1[0] ... _in1[7] _in2[0] ... _in2[7]
.outputs _out[0] ... _out[7]
.names _in1[0] _in2[0] _out[0]
01 1
10 1
...
.names _in1[7] _in2[7] _out[7]
01 1
10 1
.end
```

(d) Lookup tables (in BLIF format)

```
call mod_ret_addr
movzbq (in1_0), in1_0
movzbq (in1_1), in1_1
movzbq (in2_0), in2_0
movzbq (in2_1), in2_1
lea (out_1,in1_0), %r10
mov (%r10,in2_1), %r10b
lea (out_1,in1_1), %r10
mov (%r10,in2_0), %r10b
lea (out_0,in1_0), %r10
mov (%r10,in2_0), %r10b
lea (out_0,in1_1), %r10
mov (%r10,in2_1), %r10b
```

(e) An XOR gate of the generated circuit (in assembly)

Listing 7: The output of each compiler pipeline for an XOR circuit.

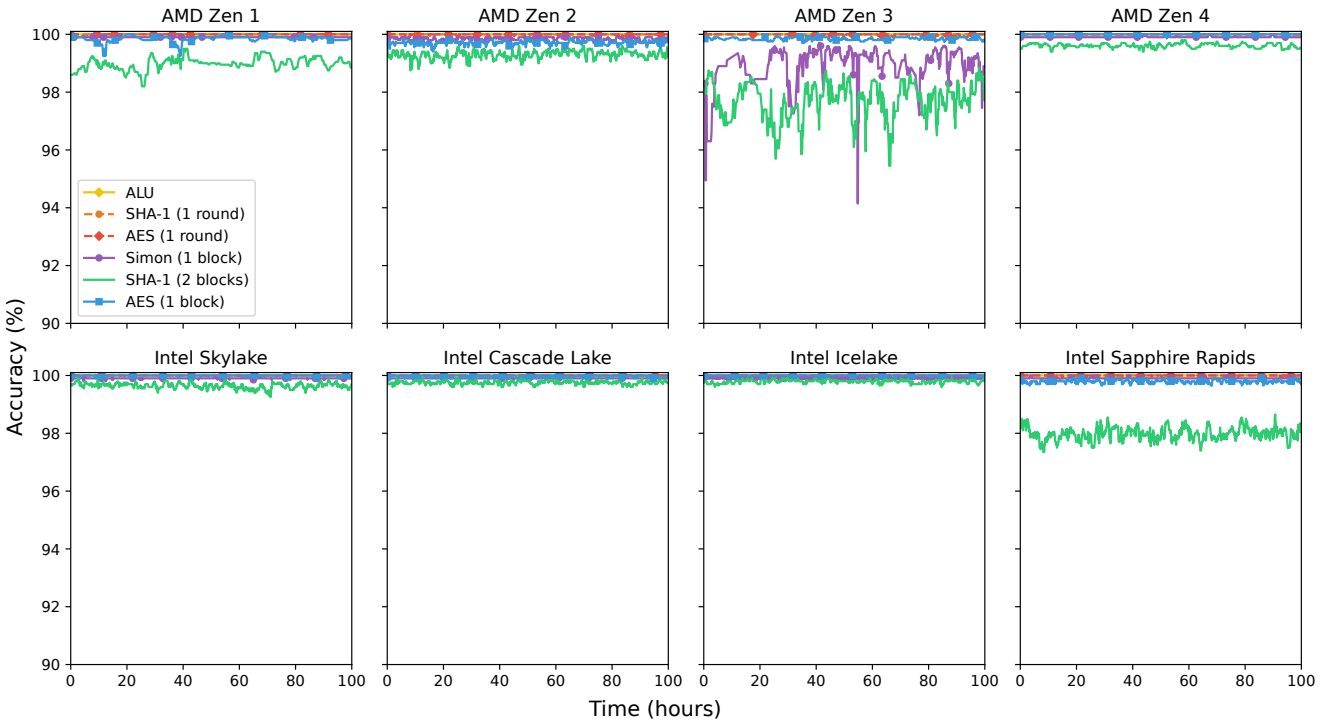


Figure 3: The accuracy of Flexo over 100 hours.

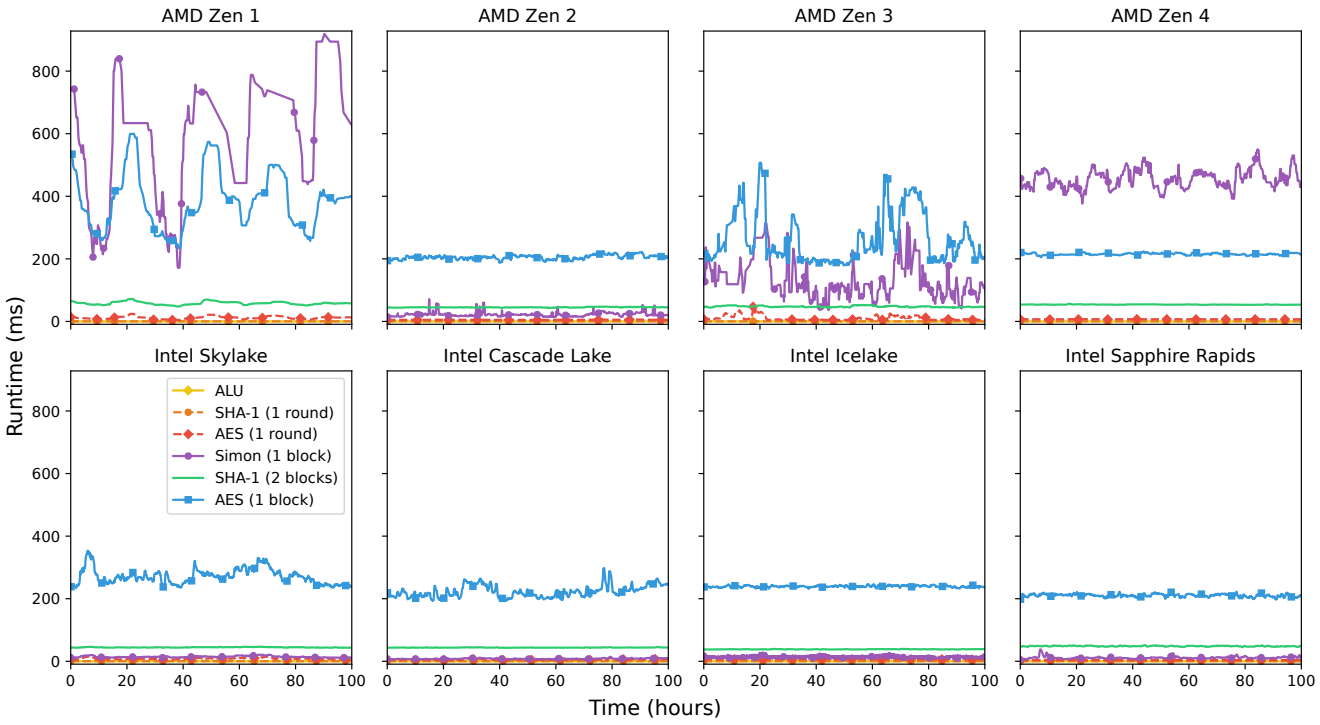


Figure 4: The runtime of Flexo over 100 hours.