WeMu: Effective and Scalable Emulation of Microarchitectural Weird Machines

Dries Vanspauwen
DistriNet, KU Leuven
Leuven, Belgium
dries.vanspauwen@gmail.com

Lesly-Ann Daniel
DistriNet, KU Leuven, Belgium
& Eurecom, Biot, France
lesly-ann.daniel@eurecom.fr

Jo Van Bulck
DistriNet, KU Leuven
Leuven, Belgium
jo.vanbulck@kuleuven.be

Abstract

Recent research on *Microarchitectural Weird Machines (μWMs)* has shown that microarchitectural optimization features, originally exploited for data exfiltration, can also facilitate hidden computation. Emerging μWMs, enabled by dedicated compilers, have become increasingly practical, evading conventional analysis tools by executing complex cryptographic algorithms and unpacking malware entirely within the microarchitectural domain. To address the lack of defensive capabilities against this growing threat, we introduce WeMu, the first emulation-based framework specifically designed for the analysis of μWMs.

WeMu enables security analysts to observe and reverse engineer hidden microarchitectural computations through novel abstractions that accurately replicate μWM behavior without the overhead and limitations of full microarchitectural simulation. We validate WeMu's effectiveness by successfully emulating μWMs ranging from basic logic gates to sophisticated cryptographic routines consisting of thousands of gates. WeMu establishes the first practical foundation for the analysis and reverse engineering of microarchitectural computations, paving the way for more effective defenses.

1 Introduction

Modern processors achieve remarkable computational performance through sophisticated microarchitectural optimization features such as speculative execution, out-of-order execution, and complex memory hierarchies. While these optimizations dramatically improve performance under normal operating conditions, they also introduce unintended security vulnerabilities that can be exploited by attackers [8, 15]. A prominent class of attacks leveraging such vulnerabilities are *timing side-channel attacks* [15, 37], which exploit observable timing differences in microarchitectural operations to infer sensitive information such as cryptographic keys or private data. More recently, researchers have also demonstrated *transient execution attacks* [8, 24, 26, 34, 38, 39], which leverage speculative and out-of-order execution to violate security boundaries and leak confidential information through side channels.

Building upon the foundations of transient execution attacks, a new line of research has shifted focus from data exfiltration to covert computation. ExSpectre [40] first demonstrated how transient execution could be used to hide malicious computations in speculative code paths. This concept has been extended and formalized in subsequent works as *Microarchitectural Weird Machines (μWMs)* [14, 21, 22, 43, 44], which represent a novel computational paradigm that exploits microarchitectural side effects for stealthy computation. Particularly, μWMs encode data using Microarchitectural Weird Registers (μWRs), with all existing implementations employing cache-based μWRs [14, 21, 22, 43, 44]. These "weird" registers encode state through cache presence: a memory location being cached corresponds to state '1', whereas absence from cache indicates state '0'. Computations occur through Microarchitectural Weird Gates (μWGs), which leverage transient execution to perform boolean operations over μWRs. Specifically, weird gates consist of carefully crafted load patterns that construct transient race conditions following an exception or mispredicted branch, causing the cache state of an output μWR to change based on the cache states of the input μWRs. Recent research demonstrates that μWMs are becoming increasingly sophisticated and practical, with advances including complete cryptographic implementations and an automated compiler that makes μWMs accessible to non-security experts [44].

A compelling aspect of performing computations in microarchitectural weird machines is that they remain inherently invisible to conventional static and dynamic binary analysis tools. Static analysis tools, like symbolic execution [23], traditionally model only the architectural CPU state. While recent work [9, 11, 17, 41, 42] extends symbolic analysis to reason about some microarchitectural effects, these efforts focus solely on leakage detection, relying on overapproximations that cannot accurately model μWM computations. Worse, dynamic analysis tools that monitor native execution, such as single-step debuggers, inevitably disrupt the timing-sensitive

behavior critical to μWMs [44]. Similarly, existing emulators fail to accurately model the microarchitectural effects upon which μWMs depend [14]. This invisibility to binary analysis tools makes μWMs particularly attractive for obfuscating computations and concealing malware.

The growing threat posed by μWMs has prompted researchers to explicitly call for defensive mechanisms [43] and specialized analysis tools [14, 44] to enable understanding and reverse engineering of these hidden computational systems. This paper responds to this need by developing WeMu[1], a specialized emulation framework for practical and accurate μWM analysis. The gap addressed by WeMu is non-trivial. Indeed, Evtyushkin et al. previously asserted that "*emulating [the microarchitectural effects that μWMs rely on] with an acceptable precision is extremely difficult as it would require first to reverse engineer the target hardware platform*" [14], suggesting that effective μWM emulation would require extensive hardware reverse engineering and detailed cycle-accurate simulations. Our work challenges this assertion by demonstrating that accurate μWM emulation can be achieved through abstract models that focus on the essential microarchitectural effects exploited by existing μWM designs, without requiring extensive, platform-specific hardware reverse engineering or high-overhead and inaccurate microarchitectural simulators like gem5 [3]. We demonstrate that WeMu's abstractions accurately expose the previously invisible microarchitectural computations, breaking the fundamental barrier of μWM analysis and providing a valuable stepping stone towards effective reverse engineering of state-of-the-art μWMs.

We implement WeMu as an extensible research prototype on top of Unicorn, a mature and open-source ISA-level CPU emulator. Particularly, we extend Unicorn with an infinite, fully associative cache model and time-bounded transient execution following exceptions and Return Stack Buffer (RSB) mispredictions. Our model introduces a key distinction from prior work by capturing how cache state dynamically influences the length of the transient window. Furthermore, we introduce a novel *cache-aware transient scheduling* abstraction that emulates transient out-of-order execution effects through sequential instruction processing, eliminating the need for complex pipeline simulation while correctly handling the out-of-order effects that μWGs exploit. We demonstrate the effectiveness of WeMu's abstract microarchitectural models in accurately emulating hidden μWM computations through a comprehensive evaluation on both hand-crafted and compiler-generated binaries from prior work [43, 44], ranging from simple 2-input boolean gates to complex cryptographic computations, including AES rounds and Simon cipher blocks. As a baseline, we first validate our benchmark μWMs on a recent Intel Xeon CPU, developing an improved calibration method to facilitate the reproducibility of prior work. Our results show that WeMu is both effective and scalable. Through cross-validation against native execution and gem5 simulation, we reveal that gem5 fails to accurately emulate μWMs while incurring substantial performance overhead. In contrast, WeMu deterministically produces correct computational results in 22 of our 24 evaluation scenarios. The only two failures occur in iterative cryptographic circuits, where WeMu produces a single undiagnosed bit error. Nevertheless, these errors only occur after correct emulation of multiple consecutive rounds involving tens of thousands of gates. Specifically, our results show that WeMu correctly emulates μWMs of up to 35,904 gates at an acceptable execution speed of 9.67 minutes – up to $\times$177 faster than gem5 – demonstrating its suitability for practical analysis workflows.

[1] A concatenation of Weird, Emulator and Mu (from *micro*architecture).

Contributions. To summarize, our main contributions are:

- We propose WeMu, the first analysis framework capable of accurately emulating hidden μWM computations.
- We extend Unicorn with an abstract cache model and cache-aware transient scheduling, enabling accurate and efficient emulation of weird registers and gates.
- We evaluate WeMu's functional correctness and performance on 24 μWMs, demonstrating that it can correctly emulate circuits of up to 35,904 gates – including complex cryptographic routines – in less than 9.67 minutes.
- We cross-validate WeMu with native execution, introducing improved calibration methods to facilitate μWM reproduction.
- We compare WeMu to gem5 simulation, finding that gem5 fails to accurately emulate weird circuits while incurring much higher overheads.

Open Science. To ensure the reproducibility of our work and to enable future science on the security analysis of μWMs, we open sourced all code and data used in this paper at https://github.com/driesvanspauwen/wemu/tree/uasc26-artifact.

2 Background

Cache Hierarchy. Modern processors typically implement multi-level cache hierarchies, typically featuring three levels (L1, L2, and L3/LLC), organized as set-associative caches. In this design, each cache level is divided into multiple sets, with each set containing several cache lines (ways). A memory address maps deterministically to a specific set, but it can occupy any line within that set. When the processor accesses a memory address, it first checks the cache: if the data is found in the appropriate set (a *cache hit*), access is fast – typically 1–4 cycles for L1 cache. If the data is not present (a *cache miss*), it must be fetched from a lower cache level or main memory, which can take 100–300 cycles for main memory.

Transient Execution. Transient execution occurs when processors execute instructions that are never committed to the architectural state. The *transient window* defines the time period during which processors execute these transient instructions before detecting that they should not have been executed and rolling back their effects. Transient execution occurs as a result of speculative and out-of-order execution. Modern CPUs employ speculative execution to predict branch outcomes or data dependencies, allowing them to execute instructions along predicted paths and avoid pipeline stalls. If a misprediction is detected, the processor rolls back any changes to the architectural state and resumes execution along the correct path. Additionally, processors execute instructions out-of-order to maximize performance and utilization of execution units. In particular, when exceptions occur, instructions following the exception might have already executed transiently. Critically, while architectural state is rolled back after transient execution, microarchitectural changes persist [24, 26].

Various microarchitectural mechanisms can trigger transient execution, which we call *transient primitives*. Previous work has demonstrated five transient primitives in practical μWM implementations: exceptions [43], Branch Predictor (BP) [22, 43], Branch Target Buffer (BTB) [43], RSB [21, 43] and Intel's Transactional Synchronization Extensions (TSX) [14, 43]. This section focuses on the two primitives currently supported by WeMu: exceptions and RSB (cf. § 4.3). The remaining three are discussed in Appendix A.

Exceptions. Exceptions act as transient primitives by exploiting out-of-order execution. With this primitive, an instruction guaranteed to raise an exception is placed immediately before the code implementing the μWG. While the exception should terminate execution by transferring control to an exception handler, modern CPUs may execute subsequent instructions out-of-order before the exception is fully processed. Wang et al. typically leverage division-by-zero exceptions (`tmp /= 0`) as transient primitives [43].

Return Stack Buffer. The *RSB* is a compact per-core hardware stack that predicts return instruction destinations. On function calls, the processor pushes the return address (the address of the instruction immediately following the function call) onto the RSB and uses it to predict the return destination. As a transient primitive, the RSB can be exploited by deliberately modifying return addresses on the stack during function execution. This creates a mismatch between the predicted destination (stored in the RSB) and the actual destination (stored on the modified stack), causing transient execution at the mispredicted address until the processor resolves the true return destination [21, 25, 27, 44].

Weird Machines. The concept of exploiting computational environments for obfuscation purposes predates μWMs and originates from the broader field of Weird Machines (WMs).

WMs arise when attackers leverage unintended computational environments within systems to perform arbitrary computation [4, 35]. These environments are the result of the gap between a system's intended behavior and its actual implementation [13]. For example, researchers have demonstrated Turing-complete computation through *ELF header processing* during program loading [35]. In general, WMs provide good obfuscation capabilities because *(i)* they exploit legitimate features of the target system; and *(ii)* they implement high-level computational logic through unconventional mechanisms that resist reverse engineering [14].

3 Microarchitectural Weird Machines

Recent research has identified microarchitectural optimization features as viable computational substrates [14, 40]. This led to the introduction of *μWMs*, a specialized type of WMs that shifts computation to the microarchitecture by leveraging microarchitectural side effects. μWMs consist of three building blocks: *(i)* registers; *(ii)* gates; and *(iii)* circuits.

Microarchitectural Weird Registers. Instead of using CPU registers or memory (which are architecturally visible), μWMs encode data in microarchitectural states, in so called *μWRs*. While various microarchitectural components could be used to encode data [14], all practical implementations use the cache [14, 21, 22, 43, 44].

In *cache-based μWRs*, a register's logical state is encoded through cache presence of a variable. If the variable is cached, the state is '1'; otherwise, it is '0'. The state of a μWR is modified through two operations: *(i)* loading the variable brings it into the cache, setting the state to '1'; and *(ii)* flushing the variable evicts it from cache, setting the state to '0'. Reading a μWR requires a *timed memory read*. By comparing the variable's load time against a threshold, cache hits (state '1') can be distinguished from cache misses (state '0').

Microarchitectural Weird Gates. *μWGs* execute boolean operations on μWRs. We discuss first how they rely on transient execution to perform architecturally invisible computations, then we give representative examples of μWGs encoding AND, OR and NOT gates.

μWGs encode their computational results in μWRs during a transient execution window. They exploit data races in transient execution, causing output μWRs to be conditionally updated before transient execution is rolled back, depending on the value of input μWRs. As the microarchitecture is not impacted by the rollback, μWRs persist after the rollback, allowing the results to be extracted later through timing measurements. We now demonstrate how data races can establish boolean operations for the AND, OR and NOT gates. For these μWGs to operate correctly, the output variable must always

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

(a) AND Implementation [43]

(b) AND(1, 1) (c) AND(1, 0)

Figure 1: AND gate implementation and data race visualization. Squares indicate loads, arrows stores.

be flushed initially (state '0'), and the architectural values of all μWRs must be zero.

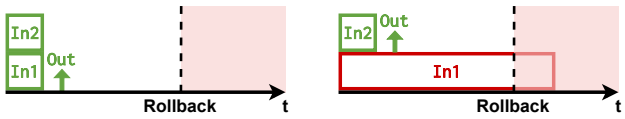
AND Gate [43]. Figure 1a demonstrates an implementation of an AND gate. The inputs (`In1`, `In2`) and output (`Out`) are cache-based μWRs. As architectural values of μWRs are zero, the listing just stores zero to `Out[0]`. From a microarchitectural perspective however, storing to `Out[0]` causes it to be cached, changing the output μWR state to '1'. Whether this store can be performed within the transient window, depends on the cache state of the inputs, as illustrated in Figs. 1b and 1c. If both inputs are cached (1b), their values can be retrieved quickly enough to issue the store to `Out[0]` within the transient window (before the rollback). This causes `Out[0]` to be cached, setting the output μWR to '1'. Conversely, if one of the inputs is not cached (1c), loading it exceeds the transient window. A rollback will flush the pipeline before the input value is available, preventing the store to `Out[0]` and leaving it in its initial, uncached state of '0'.

OR Gate [43]. Figure 2a demonstrates the implementation of an OR gate. Unlike the AND gate's sequential loads, the OR gate employs two independent store operations that an out-of-order processor can execute simultaneously. When both inputs are cached (2b), both loads execute quickly and `Out[0]` is written. Critically, when only one input is cached (2c), the processor can complete the store depending on this input quickly while waiting for the slow cache miss of the other input to resolve. Therefore, if at least one of the inputs is cached, the output will be cached as well, achieving the desired OR functionality.

NOT Gate [43]. Unlike the AND and OR gates which rely on fixed transient window lengths, the NOT gate exploits a variable-length transient window that adapts based on input cache state. Figure 3a demonstrates the NOT gate implementation, where `Aux[0]` is an auxiliary variable initially flushed. Rather than using an immediate division by zero as in AND and OR gates, it uses a division by zero with a dependency on the input μWR (line 1). This dependency enables a variable transient window length, as visualized by Figs. 3b and 3c When `In[0]` is cached (3b), it loads quickly, creating a short window during which the cache miss from `Aux[0]` cannot resolve. As a result, the store to `Out[0]` does not occur, and the output remains uncached. In contrast, when `In[0]` is not cached (3c), the window is extended with the time required to resolve the cache miss from `Aux[0]`, allowing the store to `Out[0]` to complete and the output to be cached.

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

(a) OR Implementation [43]



(b) OR(1, 1) (c) OR(0, 1)

Figure 2: OR gate implementation and data race visualization. Squares indicate loads, arrows stores.

```
1 tmp /= In[0]
2 Out[Aux[0]] = 0;
```

(a) NOT Implementation [43]

(b) Not(1) (c) Not(0)

Figure 3: NOT gate implementation and data race visualization. Squares indicate loads, arrows stores.

Microarchitectural Weird Circuits. Transient execution imposes temporal constraints that limit the computational complexity achievable within a single μWG. To overcome this limitation, *Microarchitectural Weird Circuits (μWCs)* chain multiple μWGs together, each operating within its own transient window. Data flows between gates when they share the same memory address (pointing to the same cache line), allowing the output of one μWG to serve as input for the next through cache presence state [14].

Flexo Encoding. Wang et al. [44] significantly enhanced μWM performance and accuracy through their Flexo framework, which computes arbitrary N-input boolean functions within a single transient window (where N depends on the CPU architecture, typically 4). Flexo employs differential encoding, representing each logical value w using two wires (w_- and w_+) stored in separate μWRs. This encoding enables the transformation of logical functions into inversion-free minterm canonical form (sum of products), addressing two critical limitations: *(i)* it increases the computational complex-

ity achievable within individual transient windows, and *(ii)* it resolves the incompatibility between inverting and non-inverting logic within single transient windows. This incompatibility previously arose because NOT gates require variable-length transient windows while AND and OR gates operate with fixed-length windows, forcing their combination to span separate transient windows.

Wang et al. complemented the Flexo framework with an automated compiler that transforms high-level C/C++ code into optimized Flexo μWMs, making these techniques accessible to non-experts and enabling complex implementations such as cryptographic algorithms.

4 Problem Statement

Recent μWM research has demonstrated obfuscation systems that challenge conventional malware analysis techniques. This section examines two representative obfuscation applications from prior work (§ 4.1) and explores why existing analysis approaches struggle with such systems (§ 4.2). We then propose our solution to address this analysis gap and outline the specific objectives for developing our analytical tool (§ 4.3).

4.1 μWMs Adversary Applications

Prior work [14, 44] has suggested concrete μWM-based obfuscation applications with the purpose of evading both static and dynamic analysis approaches.

Weird Obfuscation System. Evtyushkin et al. [14] developed an obfuscation system that conceals a reverse shell in a benign ping processing program. The reverse shell is AES-encrypted and stored alongside the encryption key, which itself is XOR-encrypted with a one-time pad. When the program receives ping payloads, it uses them as potential decryption keys in a TSX-based XOR μWM to attempt recovery of the encryption key and decrypt the reverse shell. For incorrect payloads, the μWM is guaranteed to abort transient execution through a division-by-zero error.

μWM-Enhanced UPX. Executable packers compress binaries to reduce file size while providing basic obfuscation through self-extracting compressed formats. Wang et al. extend UPX,[2] one of the most widely used packers [18], to integrate μWMs into the unpacking process. Their modifications encrypt compressed payloads and use μWM-based decryption during unpacking, hiding the decryption computation entirely within the microarchitectural state.

Obfuscation Effectiveness. Both applications are effective in obfuscating programs by hiding cryptographic operations in the microarchitectural state. μWMs perform computations that appear to analysis tools as ordinary memory loads and data dependencies – often in dead code paths – making the actual cryptographic operations difficult to identify. During the cryptographic operation, the computational state exists entirely within microarchitectural components like the cache, remaining invisible to traditional binary analysis techniques that operate solely on architectural abstraction.

[2] https://upx.github.io/

4.2 μWM Analysis Challenges

Analyzing μWMs requires modeling the underlying microarchitecture with sufficient precision. This presents a considerable challenge for analysis tools, as modern commercial CPUs feature proprietary, chip-specific, and highly intricate microarchitectures that have been described as "extremely difficult" to emulate accurately [14]. Consequently, analysts encounter a fundamental trade-off: while enhancing microarchitectural fidelity can improve analysis precision, it also requires significant reverse engineering and development efforts, renders the analysis approach inherently more chip-specific, and incurs a considerable performance penalty.

This section reviews standard binary analysis techniques, including symbolic execution, single-step debugging, hardware emulation, and hardware simulation, assessing their current limitations in facilitating the analysis of μWMs.

Symbolic Execution. Symbolic Execution (SE) [23] abstracts over input values by treating them as symbolic variables. As the program is executed using symbolic inputs, a path constraint (formula) is built for each possible execution path. This technique provides strong formal guarantees and allows reasoning about entire classes of inputs. Some of the most widely recognized SE tools include angr [36] and KLEE [6], but their focus on the architectural domain makes mainstream SE tools insufficient for analyzing microarchitectural computations.

Recent work has extended SE tools to model caches [5, 12], microarchitectural side-channels [16, 20] and speculative execution [9, 11, 17, 41, 42]. However, these tools focus on detecting microarchitectural *leakage* rather than accurately modeling microarchitectural *computation*. As a result, they tend to produce abstract models that prioritize soundness over the precision required for effective μWM analysis. Moreover, SE inherently suffers from state explosion, making these tools unsuitable for scalable μWM analysis.

Debugger Inspection. Single-step debuggers inspect and manipulate a program's architectural state during execution. Prior work [44] has observed that single-step debugging inherently breaks μWM computations by interfering with their transient execution, cache state, and timing behavior. In principle, more targeted approaches could place breakpoints only at μWM entry and exit points to avoid disrupting the computational environment. However, this strategy would require an

understanding of μWM behavior in the binary and execution on hardware representative of the target microarchitectural environment, capturing only input/output behavior at best without providing insight into the actual computations.

Microarchitectural Simulation. Microarchitectural simulators aim to reproduce microarchitectural behavior in detail, enabling cycle-accurate analysis of complex interactions such as caching, speculative execution, and out-of-order processing. A prominent example is *gem5* [3], a hardware simulator capable of modeling a wide range of targets and microarchitectural components. Its microarchitectural details makes it attractive for μWM analysis. Indeed, Ayoub et al. [1] successfully used gem5 to reproduce Spectre-style attacks. However, our evaluation (§ 6.3) reveals significant limitations for μWM analysis. First, the level of detail incurs substantial computational overhead, with execution orders of magnitude slower than real hardware, severely limiting practicality for program analysis. More critically, gem5 cannot guarantee accurate simulation of proprietary real-world microarchitectures with closed-source hardware implementations.

Hardware Emulation. Hardware emulators – such as QEMU [2] and Unicorn [33] – reproduce system behavior at the Instruction Set Architecture (ISA) level, enabling cross-platform execution of compiled programs. Unlike hardware simulators, emulators typically lack built-in microarchitectural modeling, making existing emulators unsuitable for analyzing μWMs. For instance, *Unicorn* is a lightweight CPU emulation framework based on QEMU that focuses exclusively on ISA-level emulation. However, Unicorn provides an extensible hooking mechanism that allows interception and modification of instruction execution, offering a foundation for custom microarchitectural extensions.

4.3 Goals

μWM literature shows significant improvements in computational complexity, accuracy and performance of μWMs. Furthermore, Wang et al.'s automated μWM compiler [44] makes μWMs accessible to non-security experts. Given these trends, existing literature consistently emphasizes the need for effective *detection* and *analysis* tools [14, 43, 44].

This work addresses the analysis challenge by developing an emulator capable of emulating the microarchitectural effects that μWMs exploit for computation. Our approach enables observing, analyzing and reverse engineering μWM computations that would remain hidden when using existing binary analysis approaches.

A Unicorn Extension. We select Unicorn as our base emulation platform and extend its ISA-level capabilities with microarchitectural models. Rather than replicating complete microarchitectures, our implementation is based on *abstract models* that capture only the essential microarchitectural effects exploited by μWMs. This targeted approach reduces implementation complexity and runtime overhead, while maintaining emulation accuracy. Unicorn's lightweight architecture and extensible hooking system facilitate integration of these microarchitectural features, making our tool *easily extensible* with new μWM designs.

Microarchitectural Emulation Scope. μWMs feature extensive flexibility in their implementation as they can potentially exploit numerous microarchitectural components for encoding weird registers or gates. This natural flexibility makes it practically infeasible to create *universal* analysis tools covering all possible μWMs. Instead, we scope our emulation to support the most practical and accurate μWMs demonstrated to date, including those generated by state-of-the-art automated μWM compilers [44].

First, we implement support for cache-based μWRs, which are the only type used in existing practical μWM implementations [14, 21, 22, 43, 44].

Second, we prioritize RSB-based transient primitives due to their integration into the Flexo framework [44]. Flexo-generated μWMs achieve the highest accuracy and execution speeds reported in current literature, while the Flexo compiler significantly increases the accessibility of μWMs.

Finally, we include exception-based transient primitives, based on Wang et al.'s findings [43]. Their comparative analysis shows exception-based μWMs achieve 99.99% accuracy, outperforming BP (94.79%) and BTB (93.52%) approaches.

5 WeMu's Modeling of the Microarchitecture

This section covers the core technical contributions of this work, discussing the design and implementation of WeMu. We identify the specific microarchitectural behaviors essential to μWM operation and describe the corresponding models we develop to emulate these effects. WeMu's architecture enables emulation of four key microarchitectural mechanisms: *(i)* a minimalistic cache model that captures the essence of μWR without the complexity of modern hardware cache implementations (§ 5.1); *(ii)* a transient-execution mechanism to capture timing-dependent data races between memory operations and transient window constraints (§ 5.2); *(iii)* a novel cache-aware transient scheduling abstraction that captures the essence of out-of-order execution without requiring complex pipeline simulation (§ 5.3); and *(iv)* a model of the time-stamp counter (§ 5.4). Our WeMu implementation comprises 897 lines of Python code for the emulator and 1,462 lines for the evaluation framework.

5.1 Single-Level Infinite Cache Model

In order to emulate μWRs, WeMu needs to include a model of the cache. When modeling the cache in WeMu, we focus on the specific microarchitectural effects that μWRs exploit, rather than implementing full cache complexity. Modern cache implementations are defined by two key design characteristics (cf. § 2) that must be taken into account.

First, with respect to cache hierarchy, current μWR implementations depend only on whether a variable is present in the cache, regardless of the specific cache level. Hence, a *single-level cache model* suffices for WeMu.

Second, cache organization, including placement and replacement policies, can directly affect the correctness of μWMs. When two μWRs map to the same cache set, operations on one of them may evict the other if the set reaches capacity, causing μWM failure. In existing literature, authors try to actively avoid this unwanted and unintended behavior [44]. To prevent these failures during emulation, WeMu implements an *infinite-size cache model*. Since existing μWMs explicitly manage cache state through loads and flushes rather than relying on eviction-based computation, an infinite cache model captures their intended behavior.

Implementation of a Single-Level Infinite Cache. We extend Unicorn with a *single-level infinite-size cache model* by implementing the cache as a set of addresses. The model is parameterized by two constants: t_{miss} and t_{hit}, representing the cycle counts for cache misses and hits respectively.[3]

Integration with Unicorn occurs through hooks on memory operations – load and store operations trigger the cache's $update(a)$ method which adds address a to the cache, while flush operations call $flush(a)$ which removes address a. The cache model is implemented as a plugin, enabling straightforward replacement with more sophisticated models to accommodate potential evolution of μWMs.

5.2 Transient Execution Support

μWGs encode architecturally invisible boolean operations via transient data races. To faithfully emulate μWGs in WeMu, we must extend the framework with several components: *(i)* models of microarchitectural components that may trigger transient execution; *(ii)* a mechanism to capture transient data races; and *(iii)* a rollback system that restores architectural state without flushing microarchitectural effects.

WeMu adapts and extends the transient model proposed by Revizor [31], a framework for testing microarchitectural leakage in CPUs against speculation contracts, incorporating refined timing precision essential for modeling the subtle data races in μWM computations.

[3]These parameters are used for modeling data races, cf. next section.

Transient Primitives. WeMu supports both exception-based and RSB-based transient primitives. For exceptions, we intercept division-by-zero errors from Unicorn and trigger transient execution of subsequent instructions.

For RSB-based primitives, we model the RSB as a stack that tracks return addresses. On function calls, we push the return address; on returns, we compare the RSB prediction with the actual stack address. Address mismatches trigger transient execution at the mispredicted location.

Timing Parameters. μWGs rely on transient data races where memory operations compete against the transient window length to conditionally update μWRs (cf. Figs. 1b, 1c, 2b, 2c, 3b and 3c). In real processors, the transient window length depends on multiple factors including return order buffer size, instruction complexity, and data dependencies [40]. Accurately modeling these factors would require detailed processor-specific reverse engineering and complex pipeline simulation. Instead, WeMu employs *configurable timing parameters* that abstract these complexities away, but still accurately capture transient data races involved in μWM.

Table 1 shows WeMu's default configuration parameters. These parameters capture timing of cache hits (t_{hit}), cache misses (t_{miss}), and other instructions (t_{other}), and the base transient window length (t_{trans}). The default values for these parameters are chosen such that multiple normal instructions and cache hits fit within one transient window, but a single cache miss does not, as required by many μWGs. Note that to accurately model the transient data races exploited by μWGs, only the *relative* differences between parameter values must be correct; accuracy of absolute values is not required.

Table 1: WeMu's default timing parameters for modeling transient data races.

Param.	Value (# cycles)	Description
t_{hit}	1	Cache hit
t_{miss}	300	Cache miss
t_{other}	1	Other instructions
t_{trans}	250	Base transient window length

Dynamic Transient Window Length. The fixed transient window length introduced in Table 1 is sufficient to correctly capture data races for simple gates like AND (Fig. 1a). However, the NOT gate (Fig. 3a) exploits the fact that transient windows vary based on the cache state of the input. To capture this behavior, we add a *dynamic transient window length* to WeMu. When a division-by-zero exception depends on a variable (e.g., `tmp /= In[0]`), WeMu dynamically adjusts the effective transient window length (Δ_{trans}) based on the variable's cache presence. If the variable is cached, the transient window is simply set to the base transient window length

$\Delta_{trans} = t_{trans}$. If the variable is not cached, WeMu extends the window by adding cache miss latency to the default length: $\Delta_{trans} = t_{trans} + t_{miss}$. This accounts for the additional time required to resolve the divisor before triggering the exception. Using default values from Table 1, this extension increases the limit from 250 to 550 cycles, permitting a cache miss (300 cycles) to complete within the transient window only if the input variable is not cached.

Transient State and Rollback. When in transient execution mode, WeMu commits the effects of transiently executed instructions directly to the architectural state tracked by Unicorn, unlike actual hardware where such effects are buffered and remain uncommitted. This design reduces implementation complexity and allows analysts to examine the effects of transiently executed instructions interactively using Unicorn's familiar interface.

Rollbacks after reaching the end of the transient window are implemented using a checkpointing mechanism similar to Revizor [31] and SpecFuzz [30]. The emulator creates a checkpoint before transient execution begins, which can later be resumed. When transient execution is rolled back WeMu reverts the architectural state to this checkpoint, while preserving microarchitectural state changes.

In-Order Transient Data Races (AND, NOT). With all the required mechanisms in place, we can now reason about how WeMu uses the parameters from Table 1 to effectively emulate simple, in-order transient data races as employed in the AND and NOT gates (cf. Figs. 1 and 3). When a transient primitive is activated, WeMu enters transient execution mode. WeMu first determines the effective transient window length Δ_{trans} (as explained above) and subsequently keeps track of the amount of clock cycles executed transiently as a transient depth (d_{trans}). At each sequential instruction, d_{trans} is updated according to the timing parameters defined in Table 1. When d_{trans} exceeds Δ_{trans}, transient execution is rolled back.

5.3 Cache-Aware Transient Scheduling

The in-order transient model described in the previous section works for the AND and NOT gates, where data dependencies are established through sequential load and store operations. However, the OR gate relies on out-of-order execution to process two independent instruction sequences simultaneously. Hence, the simple sequential transient model proves insufficient here: it would prematurely end transient execution after encountering a cache miss, without attempting to execute the following instructions like a real out-of-order processor would. This section presents an abstraction that captures the essential effects underlying OR gates and similar constructs, avoiding the complexity of modelling a full out-of-order pipeline.

```
1 B1:
2   movzx rcx, byte [r13] ; Load input 1 (r13)
3   add   rcx, r15        ; Add output address (r15)
4   mov   al, byte [rcx]  ; Load output to dummy
5 B2:
6   movzx rcx, byte [r14] ; Load input 2 (r14)
7   add   rcx, r15        ; Add output address (r15)
8   mov   dl, byte [rcx]  ; Load output to dummy
```

Listing 1: x86 OR gate assembly implementation (cf. Fig. 2a). Input registers addresses are in `r13` and `r14` and output is in `r15`. The colors show how an out-of-order processor applies renaming to the `rcx` register, differentiating true dependencies (same color) from false dependencies (different colors).

OR Gate on Actual Hardware. To examine how out-of-order processors execute the OR gate (Fig. 2a), we analyze a corresponding assembly implementation in Listing 1. This implementation deviates slightly from the pseudocode by caching the output μWR (address in `r15`) via a load to a dummy register (`al` and `dl`), rather than by *storing* zero to the output register, as done in the pseudocode. This implementation difference aligns with prior works' code artifacts [43, 44].

The listing contains two similar code blocks `B1` (lines 2-4) and `B2` (lines 6-8), each handling one input. Each block loads the input μWR's architectural value (always zero) into `rcx`, adds the output address to `rcx`, then loads this output address to a dummy register, causing it to be cached.

Inside block `B1`, the use of `rcx` introduces Read-After-Write (RAW) dependencies between lines 2 and 3, as well as lines 3 and 4. Block `B2` is analogous. The processor cannot resolve RAW hazards and thus processes the instructions within a block sequentially. However, between the two blocks themselves there are only *false dependencies*: Write-After-Write (WAW) hazards (between lines 2 & 6 and 3 & 6) and Write-After-Read (WAR) hazards (between lines 4 & 6). Through register *renaming*, the processor assigns different physical registers to `rcx` in each block, enabling out-of-order execution. Thus, `B2` can begin executing before `B1` completes, particularly when `B1` encounters a cache miss at line 2. This mechanism ensures that a cache hit from either input will eventually cache the output.

Overflowing Instruction Tracking. Rather than emulating a full out-of-order execution engine, we introduce a simplified abstraction called *cache-aware transient scheduling*. This model captures the essential microarchitectural effects of out-of-order execution by processing instructions sequentially. The core idea is to identify instructions that cannot complete within the transient window (we call them *overflowing instructions*) and to skip all instructions with RAW dependencies on overflowing instructions. An instruction is classified as overflowing for one of two reasons:

1. **Intrinsic latency:** The instruction has an intrinsic latency from performing its own effects (predetermined

Algorithm 1: Cache-aware transient scheduling. r_{dst} and r_{src} denote instruction destination and source register sets, $load_{addr}$ denotes the load address in case the instruction is a load.

$overflowing_{regs} \leftarrow \emptyset$
$overflowing_{addr} \leftarrow \emptyset$

while $d_{trans} < \Delta_{trans}$ **do**

  $(r_{dst}, r_{src}, load_{addr}) \leftarrow$ parse next instruction
  $t \leftarrow$ instruction's intrinsic latency

  if *match between* r_{src} *and* $overflowing_{regs}$ ① *RAW*
    add r_{dst} to $overflowing_{regs}$

  else if $t > \Delta_{trans} - d_{trans}$ ② *Intrinsic latency overflows*
    add r_{dst} to $overflowing_{regs}$
    add $load_{addr}$ to $overflowing_{addr}$

  else ③ *Non-overflowing instr executes normally*
    remove r_{dst} from $overflowing_{regs}$
    execute instruction
    $d_{trans} += t$

update cache with $overflowing_{addr}$
rollback architectural state

using configuration values from Table 1) which exceeds the transient window length, e.g., due to a cache miss for a load instruction.

2. **RAW dependencies:** The instruction is reading a register last written by an overflowing instruction.

Algorithm 1 details how cache-aware transient scheduling works. WeMu tracks dependencies on overflowing instructions using a list of registers, denoted $overflowing_{regs}$. When an instruction overflows – either due to a RAW dependency or because its intrinsic latency exceeds the transient window – WeMu adds all of its destination registers to $overflowing_{regs}$. This allows overflow conditions to propagate through the instruction stream, similar to e.g. taint tracking.

On actual hardware, instructions that overflow due to RAW dependencies do not begin executing, because they cannot access their required input data. WeMu reflects this by skipping such instructions (case ①).

In contrast, instructions that overflow due to instrinsic latency *do* start executing on actual hardware, and while their architectural results are never committed, they might cause microarchitectural changes. For instance, when a cache miss exceeds the transient window, its asynchronous load operation can still cause the loaded address ($load_{addr}$) to be cached, even after the rollback. WeMu reflects this by storing those addresses in a separate list, $overflowing_{addr}$, and applies the corresponding cache updates only after the transient window ends (case ②).

Example. Listing 2 provides a curated WeMu execution trace demonstrating the effectiveness of cache-aware transient execution on the OR gate assembly implementation of

```
Execution mode: transient (limit: 250)
; B1
Executing line 2: movzx rcx, byte ptr [r13]
  CACHE MISS
  ② Skipped: overflows (rcx now overflowing)
Executing line 3: add rcx, r15
  ① Skipped: RAW dep. with overflowing rcx
Executing line 4: mov al, byte ptr [rcx]
  ① Skipped: RAW dep. with overflowing rcx
; B2
Executing line 6: movzx rcx, byte ptr [r14]
  CACHE HIT
  ③ rcx overwritten (now non-overflowing)
Executing line 7: add rcx, r15
Executing line 8: mov dl, byte ptr [rcx]
  CACHE MISS
  ② Skipped: overflows (dl now overflowing)
  Output gets cached during async load

Output value: 1 (r15 cached)
```

Listing 2: Curated WeMu execution traces for exception-based OR gate (Listing 1) with inputs (0, 1), demonstrating cache-aware transient scheduling in action.

Listing 1 with inputs '0' (`r13` is not cached) and '1' (`r14` is cached). The cache miss for the first input `r13` (line 2 in Listing 1) is overflowing, as its intrinsic latency of 300 cycles exceeds the transient window length of 250 cycles. This causes `rcx` to be added to $overflowing_{regs}$ (case ②). Due to RAW dependencies with `rcx`, lines 3 and 4 are skipped (case ①).

The second input `r14` is loaded in line 6, creating a WAW dependency with line 2 because they both use `rcx` as destination register. Real processors handle this through register renaming and simultaneous execution. We emulate a similar effect by allowing non-overflowing instructions to "overwrite" overflowing registers (case ③). When a non-overflowing instruction writes to a overflowing register (line 6), we remove the register from $overflowing_{regs}$ and execute the instruction normally. In real out-of-order processors, this overwriting would cause correctness issues since older instructions might expect the previous `rcx` value. However, our transient model avoids this problem because older instructions are either executed immediately or permanently skipped – they never remain waiting for values that could be overwritten.

Note that this strategy applies exclusively to *transient out-of-order execution*. Using it for non-transient execution would violate ISA-level correctness, as we rely on skipping instructions that should never be omitted architecturally. During transient execution, however, skipping is permissible because instructions exceeding the transient window would effectively be skipped on actual hardware as well, and we restore the checkpoint upon aborting transient execution anyway.

5.4 Time-Stamp Counter Support

Existing μWM implementations rely on *timed memory reads* to extract cache-encoded data back into architectural state.

These timing measurements use the x86 `rdtscp` instruction to read the processor's time-stamp counter and determine memory access latency. The typical approach reads the counter before and after accessing a target variable, then compares the cycle difference against an experimentally determined threshold to distinguish cache hits from misses.

Since Unicorn does not model timing behavior, WeMu implements a *global time-stamp counter* which is incremented for every instruction according to the timing parameters introduced in the transient model (Table 1). When WeMu encounters an `rdtscp` instruction, it intercepts the call and writes the current counter value to the appropriate registers, simulating the instruction's behavior on actual hardware. As with the transient model, only relative timing differences are relevant, eliminating the need for cycle-accurate processor modeling.

5.5 WeMu's Analysis Capabilities

WeMu offers analysts three key analytical capabilities for examining microarchitectural weird machines. First, it enables inspection of transiently executed instructions through Unicorn's familiar interface. While real processors buffer transient effects without committing them to architectural state, WeMu directly commits these transient effects, making them immediately visible for analysis. Second, analysts can directly examine WeMu's microarchitectural components (cache, RSB, time-stamp counter) to understand how μWMs manipulate their states. Third, WeMu provides comprehensive and extensible emulation logging, generating detailed execution traces (cf. Listing 2) that can be configured to capture specific instruction ranges or microarchitectural events of interest, such as transient window timing behavior, cache hits and misses, etc. Given these capabilities, WeMu transforms the typically invisible "opaque box" microarchitectural domain into an observable, debuggable "clear box" environment where analysts can step through and inspect state at all times.

6 Evaluation

This section evaluates WeMu's effectiveness as an analysis tool and validates our design choices. We aim to address the following research questions:

RQ1 Reproducibility. We examine the reproducibility of prior μWMs on hardware different from that used by the original authors and identify the calibrations required for successful reproduction.

RQ2 Effectiveness. We examine WeMu's functional correctness by verifying that it produces identical computational results to μWMs executing on real hardware across various implementations from previous work [43, 44].

RQ3 Scalability. We assess WeMu's scalability by including in our benchmark μWMs ranging from single gate logic functions to the most complex circuits in literature, spanning thousands of gates.

RQ4 Performance. We analyze WeMu's execution performance to determine whether it achieves speeds suitable for practical analysis workflows.

RQ5 Comparison with gem5. We evaluate gem5 as an alternative emulation framework, examining its limitations for μWM analysis regarding functional correctness and execution speeds.

Our evaluation benchmark comprises all exception-based [43] and RSB-based [44] μWMs that fall within WeMu's scope, totaling 24 implementations. All experiments are conducted on a server equipped with an Intel Xeon Gold 5515+ CPU at a base frequency of 2.0 GHz, running Ubuntu 24.04.2 LTS.

6.1 Native μWM Reproduction (RQ1)

Before evaluating WeMu, we validate that the original μWMs achieve high accuracy on our testbed under native execution, confirming the correctness of our baseline. This independent reproduction additionally contributes to the field by validating the results of prior work [43, 44].

Calibration. μWM implementations require microarchitecture-specific calibration to achieve high accuracy, as acknowledged in existing work [43, 44]. This calibration involves tuning parameters that control μWM behavior. For instance, exception-based μWMs implement configurable delays after each transient window to ensure completion of asynchronous operations, such as loads issued during transient execution to cache output μWRs. We determine optimal calibration by executing μWMs over a range of different parameter values and selecting the configuration achieving the highest accuracy.

Appendix B provides a detailed discussion of our grid search calibration methodology, parameter sensitivity analysis, and our optimal configuration findings.

Results. Table 2 presents the native accuracy results for all tested μWMs. Exception-based gates generally achieve high accuracies ($> 99.7\%$), with one notable exception: the XOR gate consistently fails for the (1,1) input combination, yielding only 75.179% overall accuracy despite extensive parameter tuning. RSB-based gates achieve accuracies slightly below those reported in existing work ($\geq 97.8\%$ [44]) but remain sufficiently high to validate correct implementation.

Overall, the results in Table 2 clearly demonstrate that, following suitable parameter calibration, the μWMs explored in prior work can be reliably reproduced and may pose a real threat on commodity hardware. More extensive parameter tuning could likely further improve these results, though hardware optimization is not the focus of this evaluation.

Table 2: Evaluation results of exception-based and RSB-based μWMs, listing amount of gates, average speed, and accuracy for native execution; and average speed and emulation success (✓) or failure (✗) for emulated execution. Measurements are taken over 1,000,000 (‡), 1,000 (†), or 50 (✠) iterations. Circuits with Error Correction (EC) are indicated.

			Native		*Emulated*	
	Computation	**# μWG**	**Speed (s)**	**Acc.**	**Speed (s)**	**Corr.**
Exception-based	ASSIGN ‡	1	1.6E-6	99.98%	16E-3	✓
	AND ‡	1	1.6E-6	99.98%	16E-3	✓
	OR ‡	1	1.6E-6	99.99%	16E-3	✓
	NOT ‡	1	2.6E-6	99.97%	42E-3	✓
	NAND ‡	3	7.4E-6	99.89%	91E-3	✓
	XOR ‡	5	13.7E-6	75.18%	374E-3	✓
	↪ (0, 0)	—"—	—"—	99.99%	—"—	✓
	↪ (0, 1)	—"—	—"—	99.97%	—"—	✓
	↪ (1, 0)	—"—	—"—	99.97%	—"—	✓
	↪ (1, 1)	—"—	—"—	0.54%	—"—	✓
	MUX ‡	6	9.0E-6	99.71%	94E-3	✓
RSB-based	AND ‡	1	703E-9	94.17%	14E-3	✓
	OR ‡	1	809E-9	94.51%	14E-3	✓
	NOT ‡	1	707E-9	93.49%	12E-3	✓
	NAND ‡	1	816E-9	93.11%	14E-3	✓
	XOR ‡	1	788E-9	94.67%	15E-3	✓
	XOR3 ‡	1	935E-9	94.89%	21E-3	✓
	XOR4 ‡	1	1.8E-6	94.65%	32E-3	✓
	MUX ‡	1	855E-9	93.39%	17E-3	✓
	ALU (EC) ‡	32	16.7E-6	100.0%	311E-3	✓
	8-bit ADD ‡	21	14.6E-6	94.78%	245E-3	✓
	16-bit ADD ‡	66	41.2E-6	97.82%	630E-3	✓
	32-bit ADD ‡	150	91.8E-6	97.43%	1.3	✓
	SHA-1 round (EC) †	544	250.1E-6	100.00%	4.5	✓
	AES round (EC) †	2,524	2.7E-3	100.00%	21.4	✓
	Simon block (EC) ✠	4,322	4.2E-3	99.90%	35.5	✓
	SHA-1 2 blocks (EC) ✠	87,240	53.3E-3	98.60%	690.1	✗
	AES block (EC) ✠	32,302	211.3E-3	99.80%	261.2	✗

6.2 Emulated μWM Evaluation

Table 2 reports the results of emulating all μWMs from our benchmark with WeMu. These results demonstrate the effectiveness, scalability, and generality of WeMu.

Unit Testing Framework. To facilitate WeMu's evaluation, we developed a comprehensive unit testing framework that serves several purposes: *(i)* evaluation of WeMu's correctness; *(ii)* facilitate practical analysis workflows; and *(iii)* help ensuring the reliability and backwards compatibility of any future WeMu extensions. The framework features a robust ELF loader that automatically detects and maps memory segments from ELF binaries, plus an assembly loader for rapid testing of smaller μWM implementations. A practical command-line interface enables easy execution of individual tests, test classes, or complete test suites over ranges of input combinations, providing clear pass/fail output. Beyond evaluation, this framework is valuable for rapid prototyping and analysis of μWMs. Analysts can quickly set up and run tests using high-level abstractions while retaining full control over low-level execution parameters, including memory layout, register initialization, and execution trace generation.

Effectiveness (RQ2). Table 2 presents WeMu's emulation correctness results. WeMu successfully emulates all exception-based μWMs [43], including fundamental gates (AND, OR, ASSIGN, NOT) and composite circuits (NAND, XOR, MUX), producing the correct computational output across all input combinations. Notably, WeMu emulates the exception-based XOR gate correctly despite its poor hardware performance. This discrepancy indicates that the XOR gate contains no logical flaws but suffers from timing issues on our testbed, which WeMu's abstract models do not encounter.

For RSB-based μWMs from the Flexo framework [44], WeMu correctly emulates all gates and most composite circuits, with the exception of the SHA-1 2-block and AES block μWCs. SHA-1 2-block emulation fails consistently at round 67 out of 180 total rounds due to an undiagnosed single-bit discrepancy that propagates through subsequent rounds. We validated that WeMu correctly emulates round 67 in isolation when provided with clean architectural and microarchitectural state, but the failure occurs when executing as part of the full iterative algorithm. The error stems from an incorrectly calculated μWR address offset, causing access to the wrong cache line and leaving the intended line unaffected. We suspect that microarchitectural state from previous rounds may interfere with address calculations, though our attempts to clear cache state and reset the transient scheduler between rounds have not resolved the issue at the time of this writing. AES block emulation fails during the initial key expansion round, preventing execution of the full cipher. As the error occurs in the initial round, we suspect an issue with the WeMu configuration (e.g. wrongly mapped memory) rather than inter-round interference. Further investigation is needed to identify the specific cause.

Scalability (RQ3). WeMu successfully emulates complex circuits including 32-bit adders and complete cryptographic computations such as SHA-1 rounds, AES rounds, and Simon cipher blocks. Among these implementations, Simon represents the largest complete cryptographic algorithm that WeMu can correctly emulate in its entirety, encompassing 8288 registers and 4322 gates. More significantly, WeMu demonstrates its scalability by correctly emulating up to 35,904 consecutive gates when executing 2-block SHA-1 (up to round 66). This consecutive execution capability showcases WeMu's potential for handling large and complex circuits.

Performance (RQ4). We evaluate WeMu's performance by measuring and comparing the runtime of μWMs under both native hardware execution and WeMu emulation. Table 2 presents our findings. On average, emulation is 15,043× slower than native execution, with a maximum slowdown of

27,317× (exception-based XOR) and a minimum of 1,236× (RSB-based AES block). As expected, WeMu executes multiple orders of magnitude slower than real hardware due to emulation overhead – the host system must translate and interpret each target operation in software rather than executing it directly on hardware [19]. However, native-like speeds are unnecessary for our purposes, as WeMu's primary goal is providing a controlled, analyzable environment for studying μWM behavior rather than optimizing for attack deployment.

The crucial finding is that WeMu achieves acceptable execution speeds for practical analysis workflows. It emulates 2 blocks of SHA-1 (87,240 gates – our largest circuit) in 690.1 seconds or 11.50 minutes. A bit error occurs during round 67 after 4.91 minutes of successful emulation. For the Simon cipher (4,322 gates – our largest successfully emulated circuit), WeMu completes emulation in 35.46 seconds.

Beyond WeMu's performance evaluation, our native execution measurements reveal an interesting characteristic of μWM behavior. We observe that AES block encryption (32,302 gates) contains far fewer gates than SHA-1 2-blocks (87,240 gates) yet requires nearly four times longer native execution time. This discrepancy stems from the error correction mechanism triggering more retry attempts for AES rounds compared to SHA-1 rounds. Since individual AES rounds are substantially longer (up to 2,669 gates) than SHA-1 rounds (maximum 553 gates), errors are statistically more likely to occur within any given AES round, requiring more correction attempts that inflate overall execution time. Notably, WeMu avoids this issue entirely because its deterministic nature eliminates the need to emulate error correction mechanisms.

Generality. All experiments were executed using WeMu's default configuration values (as specified in Table 1), demonstrating that WeMu requires no calibration even when emulating μWMs that are calibrated to specific microarchitectures. This robustness and generality result from our design choice of abstracting away low-level, microarchitecture-specific hardware details through generic microarchitectural modeling.

6.3 Comparison with gem5 (RQ5)

gem5 [3] is a cycle-accurate computer system simulator that provides detailed microarchitectural modeling, including caching, speculative execution, and out-of-order execution. Given these capabilities and prior work demonstrating gem5's ability to reproduce Spectre attacks [1], we evaluate gem5 as an alternative platform for μWM analysis.

We evaluate both of gem5's primary simulation modes using its default configuration scripts: System-call Emulation (SE) mode and Full System (FS) mode. SE mode provides faster simulation by directly emulating system calls but omits certain system-level timing behaviors. FS mode offers complete system emulation including operating systems and more accurate system-level timing behavior but executes significantly slower.

System-call Emulation Mode. SE mode presents fundamental limitations for μWM analysis. Most critically, it currently does not support signal handling,[4] which is essential for exception-based μWMs that rely on exception handlers to continue execution after transient gate computation. Without signal handling support, exception-based μWMs simply crash when exceptions occur.

For RSB-based μWMs, we tested all gates from our evaluation and observed complete failure: all gates achieved approximately 0% accuracy with 100% error detection rates in Flexo's differential encoding scheme. This suggests fundamental issues with μWR state management during transient execution or timed memory read operations.

Full System Mode. FS mode shows improved but still inadequate μWM emulation capabilities. We conducted evaluation using the exception-based AND gate [43]. Despite extensive parameter tuning across a wide range of configuration values, the maximum achievable accuracy reached only 75%, with a clear pattern where the (1,1) input combination consistently failed while other combinations succeeded. This behavior indicates that the μWM systematically outputs '0', suggesting problems with either variable caching mechanisms or cache state interpretation during timed memory reads.

Performance Comparison. Beyond correctness issues, gem5 exhibits significant performance limitations compared to WeMu. An exception-based AND gate executes in 3.25 seconds in FS mode – over 177 times slower than WeMu's execution time of 0.0185 seconds.

Conclusion. These limitations validate our architectural decision to build WeMu on Unicorn rather than extending gem5. While gem5's comprehensive microarchitectural modeling could potentially enable more sophisticated analysis capabilities with additional research and development, WeMu's purpose-built design delivers immediate functionality for μWM analysis with acceptable performance.

7 Limitations and Future Work

While WeMu clearly demonstrates that accurate μWM emulation is achievable through targeted abstract modeling, several limitations in the current prototype present opportunities for future improvement. This section outlines those limitations and highlights promising directions for extending WeMu's

[4] See the function description of `ignoreFunc` in the gem5 source code (https://github.com/gem5/gem5/blob/stable/src/sim/syscall_emul.hh) and GitHub issue https://github.com/gem5/gem5/issues/1544, currently marked as "Not planned".

capabilities. We envision WeMu as an open-source, extensible base platform that may serve as a foundation for continued research into μWM attacks and defenses.

Addressing Implementation Edge Cases. While WeMu successfully emulates 22 of 24 tested μWMs, including several thousands of gates, two remaining edge cases (SHA-1 2-block and AES block) remain unresolved as of this writing. Our analysis indicates that these failures stem from implementation-specific issues rather than fundamental modeling limitations, suggesting they could be addressed through targeted debugging and enhanced state management.

Scope of Transient Primitives. WeMu's current implementation focuses on exception-based and RSB-based transient primitives, which represent the most practical and accurate approaches demonstrated in existing research [43, 44]. Future extensions to WeMu could incorporate support for BP- and BTB-based transient primitives [43] (cf. Appendix A). In present μWM designs, these primitives currently exhibit limited practical applicability because *(i)* they operate relatively slow due to expensive mistraining procedures; and *(ii)* they achieve relatively low accuracy as prediction mechanisms gradually adapt to deliberate mistraining attempts. Nevertheless, future work may improve their execution time and accuracy by applying advanced techniques such as Flexo's differential encoding.

Abstraction vs. Fidelity Trade-offs. WeMu deliberately adopts a microarchitecture-agnostic approach that abstracts away many platform-specific details in favor of simplified, configurable models. This design choice provides a significant advantage: the emulator does not require fine-tuning to specific target microarchitectures.

However, this abstraction might limit WeMu when considering future research directions using μWMs for hardware-software binding techniques [28]. Wang et al. [44] have proposed developing *microarchitecture-specific* μWMs that deliberately target specific microarchitectures while failing on others. Future work should investigate whether WeMu's current abstraction level sufficiently supports such microarchitecture-specific μWMs, as they may require more detailed microarchitectural models. A full system emulator like gem5 could potentially address these limitations, provided the processor features targeted by the μWM are either non-proprietary or have been sufficiently reverse engineered.

Reverse Engineering μWMs using WeMu. Reverse engineering malware presents complex challenges, with analysts facing multiple orthogonal obstacles (e.g., classical obfuscation, packing, anti-debugging defenses). As acknowledged in a recent survey [32], *"there is no single tool that can cover all aspects of malware behavior"*. WeMu addresses one specific aspect of this de-obfuscation challenge by making microarchitectural computations visible, though integration into realistic malware analysis workflows remains an area for future exploration.

Furthermore, while WeMu correctly captures the microarchitectural effects in execution traces, it remains challenging for human analysts to detect the actual computations being performed through these effects. Simple gates (AND, OR, NOT) may be recognizable in load patterns, but analyzing complex cryptographic algorithms spanning thousands of gates is practically infeasible – especially for Flexo μWMs, where data is encoded differentially and computations are implemented in minterm canonical form.

Future work could extend WeMu with automated analysis to reconstruct computational logic from microarchitectural traces. By varying inputs and observing resulting cache states and dependency patterns, such analysis could infer individual μWG functions and reconstruct the circuit, similar to black-box deobfuscation approaches [29]. Ultimately, the goal is to recover high-level algorithmic descriptions, similar to decompilation of obfuscated binaries. For cryptographic μWMs, this could entail identifying AES or SHA-1 from characteristic patterns, or leveraging tools like Aligot [7] to map input-output behavior to known functions.

Towards Precise Side-Channel Analysis. Static analysis tools for Spectre and formal semantics for speculative execution often rely on conservative overapproximations to ensure soundness [10]. While this approach is safe, it tends to include speculative paths that are infeasible in practice, leading to false positives and reports of unexploitable vulnerabilities. We believe that the abstractions developed in this work could help refine these analyses by providing a more precise transient execution model that captures the interaction between cache state and transient window length. Integrating such a model into existing tools could reduce false alarms and focus analysis on realistic, potentially exploitable vulnerabilities.

8 Conclusion

This work addresses the emerging threat posed by μWMs by developing WeMu, the first analysis framework capable of effectively emulating μWMs, transforming their previously hidden computations into observable, debuggable environments. We demonstrate that targeted abstract modeling of microarchitectural effects enables correct μWM emulation without requiring extensive hardware reverse engineering, challenging prior assertions about the difficulty of μWM emulation. Our comprehensive evaluation shows that WeMu successfully emulates μWMs ranging from simple gates to complex cryptographic circuits, providing the foundation for defensive research against microarchitectural obfuscation techniques.

Acknowledgements

This research was partially funded by the Research Fund KU Leuven, the Research Foundation – Flanders (FWO) via grant #12B2A24N, and the Cybersecurity Research Program Flanders.

References

[1] P. Ayoub and C. Maurice. Reproducing Spectre Attack with gem5: How To Do It Right? In *EuroSys Workshop*, 2021.

[2] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX ATC*, 2005.

[3] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2), 2011.

[4] S. Bratus. What hacker research taught me. https://www.cs.dartmouth.edu/~sergey/hc/rss-hacker-research.pdf. Presented at RSS, 2009.

[5] R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. Kandemir. Casym: Cache aware symbolic execution for side channel detection and mitigation. In *S&P*, 2019.

[6] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX OSDI*, 2008.

[7] J. Calvet, J. M. Fernandez, and J.-Y. Marion. Aligot: Cryptographic function identification in obfuscated binary programs. In *CCS*, 2012.

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

[9] S. Cauligi, C. Disselkoen, K. von Gleissenthall, D. Tullsen, D. Stefan, T. Rezk, and G. Barthe. Constant-time foundations for the new spectre era. In *PLDI*, 2020.

[10] S. Cauligi, C. Disselkoen, D. Moghimi, G. Barthe, and D. Stefan. Sok: Practical foundations for software spectre defenses. In *S&P*, 2022.

[11] L.-A. Daniel, S. Bardin, and T. Rezk. Hunting the Haunter — Efficient Relational Symbolic Execution for Spectre with Haunted RelSE. In *NDSS*, 2021.

[12] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke. Cacheaudit: A tool for the static analysis of cache side channels. *TISSEC*, 2015.

[13] T. Dullien. Weird Machines, Exploitability, and Provable Unexploitability. *IEEE TETC*, 8(2), 2020.

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

[15] Q. Ge, Y. Yarom, D. Cock, and G. Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J Cryptogr Eng*, 8(1), 2018.

[16] A. Geimer, M. Vergnolle, F. Recoules, L.-A. Daniel, S. Bardin, and C. Maurice. A systematic evaluation of automated tools for side-channel vulnerabilities detection in cryptographic libraries. In *CCS*, 2023.

[17] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez. Spectector: Principled Detection of Speculative Information Flows. In *IEEE S&P*, 2020.

[18] F. Guo, P. Ferrie, and T.-c. Chiueh. A Study of the Packer Problem and Its Solutions. In *RAID*, 2008.

[19] Y. Hu, H. Jin, Z. Yu, and H. Zheng. An Optimization Approach for QEMU. In *ICISE*, 2009.

[20] J. Jancar, M. Fourné, D. D. A. Braga, M. Sabt, P. Schwabe, G. Barthe, P.-A. Fouque, and Y. Acar. "They're not that hard to mitigate": What cryptographic library developers think about timing attacks. In *IEEE S&P*, 2022.

[21] D. A. Kaplan. Optimization and Amplification of Cache Side Channel Signals. *arXiv preprint arXiv:2303.00122*, 2023.

[22] D. Katzman, W. Kosasih, C. Chuengsatiansup, E. Ronen, and Y. Yarom. The gates of time: Improving cache attacks with transient execution. In *USENIX Security*, 2023.

[23] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), 1976.

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

[25] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. B. Abu-Ghazaleh. Spectre returns! Speculation attacks using the return stack buffer. In *WOOT*.

[26] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security*, 2018.

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

[28] R. Mechelinck, D. Dorfmeister, B. Fischer, S. Volckaert, and S. Brunthaler. GlueZilla: Efficient and Scalable Software to Hardware Binding using Rowhammer. In *DIMVA*, 2024.

[29] G. Menguy, S. Bardin, R. Bonichon, and C. d. S. Lima. Search-based local black-box deobfuscation: understand, improve and mitigate. In *CCS*, 2021.

[30] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer. SpecFuzz: Bringing spectre-type vulnerabilities to the surface. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[31] O. Oleksenko, C. Fetzer, B. Köpf, and M. Silberstein. Revizor: Testing black-box CPUs against speculation contracts. In *ASPLOS*, 2022.

[32] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach. Dynamic malware analysis in the modern era—a state of the art survey. *ACM Computing Surveys (CSUR)*, 2019.

[33] N. A. Quynh and D. H. Vu. Unicorn: Next Generation CPU Emulator Framework. *BlackHat USA*, 476, 2015.

[34] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*, 2019.

[35] R. Shapiro, S. Bratus, and S. W. Smith. "Weird Machines" in ELF: A Spotlight on the Underappreciated Metadata. In *WOOT*, 2013.

[36] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, and G. Vigna. SOK: (state of) the art of war: Offensive techniques in binary analysis. In *IEEE S&P*, 2016.

[37] J. Szefer. Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses. *J Hardw Syst Secur*, 3(3), 2019.

[38] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*, 2018.

[39] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. Ridl: Rogue in-flight data load. In *IEEE S&P*, 2019.

[40] J. Wampler, I. Martiny, and E. Wustrow. ExSpectre: Hiding Malware in Speculative Execution. In *NDSS*, 2019.

[41] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury. Oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE TSE*, 47(11), 2019.

[42] G. Wang, S. Chattopadhyay, A. K. Biswas, T. Mitra, and A. Roychoudhury. KLEESpectre: Detecting Information Leakage through Speculative Cache Attacks via Symbolic Execution. *TOSEM*, 29(3), 2020.

[43] P.-L. Wang, F. Brown, and R. S. Wahby. The ghost is the machine: Weird machines in transient execution. In *IEEE S&P Workshops*, 2023.

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

A Additional Transient Primitives

Branch Predictor. The *BP* monitors recent conditional branch outcomes to predict future branch directions. It can be used as a transient primitive through systematic mistraining: repeatedly executing a branch with consistent outcomes (e.g., always taken) during training, then manipulating the condition to produce the opposite result. The BP mispredicts based on training, causing transient execution of the wrong path until the correct direction is resolved [8, 43].

Branch Target Buffer. The *BTB* records mappings between branch instruction addresses and their targets for both direct and indirect branches. It can be used as a transient primitive through target address mistraining: during training, the BTB learns to associate an instruction address with a specific target, then when the actual target changes, continues predicting the trained destination. This causes transient execution at the wrong address until the processor resolves the correct target [8, 43].

Intel TSX. Intel's *TSX* provides hardware transactional memory through atomic transaction blocks. Intel TSX can be used as a transient primitive by triggering exceptions during the transaction. Exceptions cause the transaction to be aborted, but the instructions following the exception will execute transiently. Furthermore, the abort only rolls back the architectural effects of the transient instructions [14, 34].

B Calibrating Existing μWMs

When validating existing μWMs designs, they have to be calibrated to the target microarchitecture to ensure high accuracy.

B.1 Exception-based μWMs

Configuration Parameters. In the code artifact of Wang et al.'s exception-based μWMs[5] [43], we recognize the following two parameters:

- **`Delay`:** After every μWG, delay loops are inserted to ensure completion of asynchronous operations that were initiated during the transient window. They ensure that the loads or stores causing the output μWR to be cached are completed, before follow-up gates or timed memory reads use the register. The delay parameter controls the duration of these waiting loops.[6]

- **`Threshold`:** When performing timed memory reads to determine register values, this parameter differentiates between cache hits (access time $\leq$ `threshold`) and cache misses (access time $>$ `threshold`).

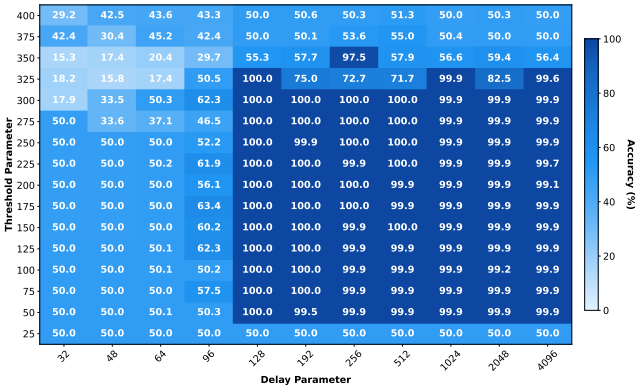
To systematically evaluate the impact of these parameters, we perform a grid search across different combinations of `threshold` and `delay` values. For each parameter combination we execute all μWMs 1 million times, maintaining consistency with the original evaluation methodology [43].

Discussion of Parameter Sensitivity. Figure 4a presents the grid search results for the ASSIGN gate, clearly revealing an operational region where the gate operates reliably. This region requires the delay to exceed 128 cycles and the threshold to fall between 50 and 300 cycles.

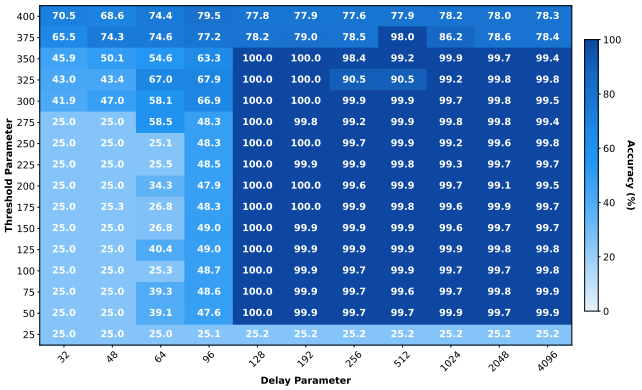
The sensitivity to these parameters stems from cache operation timing characteristics. Insufficient delay (below 128 cycles) prevents the transient load operation from completing its caching effect. During the subsequent timed read to determine output value, the data is still in transit from DRAM, causing '1's to be misinterpreted as '0's. Since '0's are correctly identified (remaining uncached), this timing error yields approximately 50% accuracy with uniform input distribution.

Threshold parameters exhibit similar behavior. Low thresholds (below 50 cycles) misclassify cache hits as misses, again causing '1's to be read as '0's while '0's remain correct, resulting in 50% accuracy. Conversely, high thresholds (above 300 cycles) misclassify cache misses as hits, causing '0's to be misread as '1's while '1's stay correct, again yielding 50% accuracy.

Figure 4b demonstrates the grid search results for the OR gate, which exhibits similar parameter sensitivity but with different accuracy results due to its dual-input design. When delay or threshold values are too low, '1's are misinterpreted as '0's, causing the input cases (0, 1), (1, 0), and (1, 1) to produce incorrect outputs, resulting in approximately 25% overall accuracy. Conversely, when the threshold is too high, '0's are misread as '1's, affecting only the (0, 0) input case and yielding 75% accuracy.



(a) Exception-based ASSIGN gate



(b) Exception-based OR gate

Figure 4: Heatmaps of exception-based gates, illustrating accuracy across various values of `threshold` and `delay`.

B.2 RSB-based μWMs

For RSB-based μWMs [44], we follow the recommended parameter optimization methodology from the original code artifact[7] and tune two critical parameters:

- **`RET_WM_DIV_ROUNDS`:** When triggering transient execution, Flexo-compiled μWMs implement a chain of dependent division operations to calculate the actual return address. This parameter controls the number of division instructions – more divisions extend the time required for return address calculation, thereby increasing the transient window.

- **`WR_OFFSET`:** Controls memory spacing between consecutive weird registers. Appropriate spacing prevents prefetcher interference and cache line conflicts that

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

[6] `DELAY` was not present as a configurable parameter in the original code artifact, but was instead hardcoded throughout the implementation. We modified it to be configurable to facilitate parameter tuning on our testbed.

[7] https://github.com/joeywang4/Flexo

would otherwise corrupt register values, as detailed in Section 3.3 of the original paper [44].

We conduct a grid search across parameter combinations to determine optimal configurations for μWGs, following the exception-based calibration methodology. For μWCs, we utilize the existing reproduction framework from the code artifact, which performs a linear search limited to the `RET_WM_DIV_ROUNDS` parameter. This linear search reduces search space, trading potential accuracy for significantly faster calibration times.