0xELF: Reverse Engineering Framework for Flexo-Generated Weird Machines

Gianmarco Lusvardi[1]✉[0009−0003−5341−0257], Mauro Andreolini[1][0000−0002−7408−6906], Luca Ferretti[1][0000−0001−5824−2297], Yuval Yarom[2][0000−0003−0401−4197], and Anirban Chakraborty[3][0000−0001−7411−7509]

[1] University of Modena and Reggio Emilia ({gianmarco.lusvardi,mauro.andreolini,luca.ferretti}@unimore.it)
[2] Ruhr University Bochum (yuval.yarom@rub.de)
[3] Max Planck Institute for Security and Privacy (anirban.chakraborty@mpi-sp.org)

Abstract. Microarchitectural weird machines (μWMs) perform computation entirely within the microarchitectural state of the processor, leaving no trace in registers or memory. Recent compiler toolchains such as Flexo (USENIX Security 2024) make this capability accessible to non-experts, turning μWMs into a practical obfuscation threat. No existing reverse-engineering tool can recover the hidden computation, as it is invisible at the architectural level. We present 0xELF, the first framework for reverse engineering Flexo-generated μWMs. 0xELF recovers the Boolean function of each weird gate through dependency analysis of transient instruction sequences, and reconstructs the full circuit wiring through symbolic execution of the enclosing weird function, producing a complete gate-level netlist. We evaluate 0xELF on all Flexo-supplied examples—including AES, SHA-1, and the Simon cipher—as well as on 16 circuits from the EPFL combinational benchmark suite compiled with the default parameters, and confirm equivalence with the original netlist in every case.

Keywords: microarchitectural weird machines, reverse engineering, binary analysis, obfuscation

1 Introduction

Modern processors achieve high performance through complex microarchitectural mechanisms, such as caches, out-of-order execution, and speculative execution. These mechanisms operate below the level of the Instruction Set Architecture (ISA) and are designed to be transparent to software. Yet over the past two decades, researchers have repeatedly demonstrated that this hidden complexity leads to a gap between the abstraction the processor is designed to expose and the one it actually exposes. This gap gives rise to a wide range of microarchitectural attacks that can leak sensitive information [6, 12, 16–18, 21, 22, 26, 39], bypassing memory isolation [21, 22, 24, 28], and establishing covert channels [16], without exploiting any software bugs, but rather by creatively using the documented ISA to manipulate the microarchitectural state of the processor.

Microarchitectural weird machines (μWMs) [15, 19, 20, 34–36] exploit this gap to perform computation entirely within the microarchitectural layer. By triggering transient execution—where the processor speculatively executes instructions whose architectural effects are later discarded—these weird machines can manipulate cache-state to encode and process data, leaving no trace in architectural registers or memory. As the computation performed by μWMs is invisible to the architecture, it can be used for stealthy computation and obfuscation, making it a powerful tool for delivery of malware and evasion of detection software [33, 34, 36].

To facilitate and simplify μWMs generation, Wang et al. [36] introduce Flexo, a new design for μWMs built on differential encoding, along with a compiler toolchain that translates C/C++ code into μWMs. Thus, Flexo gives non-experts an easy method for producing obfuscated binaries, whose computation is hidden in microarchitecture. Wang et al. [36] further show practical applications, including a packer that conceals its decryption function as a weird machine. Thus, Flexo enables a new class of obfuscated malware that is difficult to analyze and detect using traditional reverse engineering techniques.

Despite this growing threat, tooling to reverse-engineer microarchitectural weird machines remains essentially nonexistent. Traditional reverse engineering tools, such as Ghidra [5] and IDA Pro [27], operate exclusively at the architectural level, rendering the microarchitectural semantics of weird machine instructions invisible to their analysis frameworks; adapting these tools to capture such computation is not a straightforward task. Debuggers operate at the architectural level as well and destroy the fragile microarchitectural state that μWMs depend on [36]. Even microarchitectural emulators like WeMu [31], which can efficiently emulate weird machines in Qemu-based environments [1], can only trace the execution of a weird machine on concrete inputs— they cannot extract a logical description of the computation being performed. In short, no existing tool is capable, as-is, to capture and reverse-engineer the computation performed by a μWM, leaving analysts with no choice but to manually analyze the binary in an attempt to reconstruct the underlying logic. Given the complexity μWMs and the lack of visibility into their operation, such manual analysis is a daunting task.

In this work, we present `0xELF`, the first framework for reverse-engineering Flexo-generated μWMs. Given the entry point of a suspected Flexo Weird Function, `0xELF` automatically recovers the combinational circuit it implements and emits it as a gate-level netlist in the standard BLIF format [30], making the hidden computation accessible to the ecosystem of existing circuit analysis and verification tools.

At its core, `0xELF` uses symbolic execution to analyze the target weird function (circuit) while emulating the Return Stack Buffer (RSB) and maintaining an abstract cache model that tracks which addresses are affected by the weird computation and how. When `0xELF` detects an RSB misprediction—the mechanism Flexo uses to trigger transient execution—it identifies the boundaries of a weird gate. For each gate, `0xELF` spawns a separate symbolic execution that

records the register and memory effects of every instruction. From these records, it constructs a *RAW dependency forest*, that captures instruction-level data dependencies within the gate. A key insight of our work is that the structure of this forest directly encodes the gate's Boolean function. Dependent chains correspond to conjunctions and independent chains targeting the same output correspond to disjunctions, allowing 0xELF to extract the function as a Disjunctive Normal Form (DNF) directly from the forest. At the function level, 0xELF traces data flow across the entire execution, tracking how the function's inputs are encoded into weird registers, how gates consume and produce weird register values through differential encoding, and how outputs are read back into architectural state through cache-timing measurements. Finally, the output of the circuit is extracted as standard BLIF format for further analysis using existing EDA tools.

Contributions. In summary, we make the following contributions:

- We analyze Flexo-generated binaries and provide the first analysis of the memory and code layouts of its weird gates and circuits (Section 3)
- We identify the challenges inherent to reverse-engineering μWMs and highlight that existing binary analysis tools are currently unable to recover the computation they perform (Section 4).
- We present 0xELF, to the best of our knowledge the first tool capable of automatically reverse-engineering μWMs. 0xELF uses symbolic execution to detect weird gates via RSB emulation, extract their Boolean functions through *RAW dependency forest*, and reconstruct the complete circuit by tracing function-level data flow (Section 5).
- We evaluate 0xELF on all examples distributed with Flexo—including basic gates, arithmetic circuits, an ALU, SHA-1, AES, and the Simon cipher [9]—as well as on circuits from the EPFL benchmark suite [7]. The output is a standard BLIF netlist, directly usable by existing EDA tools for further analysis, optimization, or verification (Section 6).

Ethics and Open Science. 0xELF is a reverse-engineering tool, which can be used for analyzing and understanding Flexo-generated μWMs. 0xELF and the data used for its evaluation are available at https://github.com/0xADE1A1DE/0xELF

2 Background and Related Work

2.1 Modern Computer Architecture

Modern processors employ several microarchitectural features to improve execution performance, including cache memories, out-of-order execution, and speculative execution. These features operate below the ISA level and are designed to be transparent to software.

Caches. A *cache* is a small, fast memory that retains recently accessed data so that subsequent accesses to the same addresses can be served without reaching main memory. When the processor accesses an address whose data resides in the cache—a *cache hit*—the access is served directly. Otherwise—a *cache miss*—the data is fetched from main memory and placed into the cache, potentially evicting an existing line to make room. The latency difference between a hit and a miss is large enough to be measured by software, making cache residency a well-known side channel for inferring information about memory accesses [26, 39].

Out-of-Order Execution. Rather than executing instructions strictly in program order, modern processors may reorder them so long as the final architectural state is equivalent to that of a sequential execution [21]. This reordering is constrained by data dependencies: when one instruction produces a value that a later instruction consumes, the producer must complete before the consumer can execute. This relationship is known as a *Read-After-Write* (RAW) dependency. Instructions that are not linked by such dependencies may execute in parallel on different execution units, enabling better performance and throughput.

Speculative and Transient Execution. When the processor encounters a control-flow instruction whose outcome is not yet resolved—such as an indirect branch or a return—it predicts the target and speculatively continues along the predicted path. If the prediction is correct, the speculative results are committed to architectural state. Otherwise, the processor detects the misprediction, discards the speculative results, and re-executes from the correct target. Instructions executed along a mispredicted path are said to have undergone *transient execution* [21]. Although their architectural effects are rolled back, their microarchitectural side-effects—most notably changes to cache residency—persist. The interval between the start of the mispredicted path and the moment the processor detects the error is called the *transient window*.

Return Stack Buffer. One of the mechanisms for triggering transient execution is the use of *Return Stack Buffer* (RSB)—a microarchitectural structure that predicts return addresses. On each function `call`, the processor pushes the address of the following instruction onto the RSB. On a subsequent `ret`, it pops this address and begins speculatively executing from it. If the called function has overwritten its return address on the stack, the actual return target differs from the RSB prediction, causing a misprediction. The speculatively executed instructions undergo transient execution, and the processor eventually rolls back to the correct address.

2.2 Microarchitectural Weird Machines

Evtyushkin et al. [15] introduced the idea of microarchitectural weird machines (μWMs). Building on the concept of weird machines [13], they first use microarchitectural state to represent Boolean values, and then construct basic

blocks that allow operating on the microarchitectural state to compute arbitrary Boolean function. They further introduce the terms weird state and weird gates to refer to the Boolean values and the basic blocks operating on them, respectively. Evtyushkin et al. [15] demonstrate that their weird gates allow computing complex functions, such as the SHA-1 hash function, and suggest that μWMs can be used for code obfuscation because traditional reverse-engineering approaches are oblivious to weird machines.

In 2023, three independent works proposed exploiting speculative execution following mispredicted control-flow instructions for constructing cache-based μWMs [19, 20, 35]. They demonstrate that the construction provides faster and more stable gates, with better composability. They further show how μWMs can improve microarchitectural side-channel attacks. Spec-o-Scope [18] further investigates μWMs, abstracting them as a race condition between *instruction chains*, sequences of instructions that have data dependencies, and introduces vocabulary that allows describing weird gates construction. Finally, Slice+Slice Baby [25] show how to use weird gates to improve eviction-set construction [23, 32], an essential step in some cache attacks.

Flexo [36] introduces the idea of differential encoding for cache-based μWMs. Instead of using the cache residency of a memory location to indicate whether a Boolean value is true or false, they use a pair of locations, such that one location is cached for true value and the other for false. Differential encoding allows Flexo to automatically construct gates that implement arbitrary Boolean functions, whose size is only limited by microarchitectural constraints. Flexo exploits this capability through a compilation toolchain that compiles C code into a μWM. Recently, Wang et al. [34] show how to combine ideas from ExSpectre [33] with weird-gate-based μWMs, introducing *weird programs* as a new computational model for μWMs.

Although obfuscation is one of the main perceived use cases for μWMs, deobfuscation efforts remain limited. To the best of our knowledge, WeMu [31] is the only attempt in this direction. In a nutshell, WeMu emulates the execution of a weird circuit in order to recover the final results of the weird computation. However, WeMu does not extract the Boolean function the circuit implements.

2.3 Program analysis

Program analysis encompasses a range of techniques for understanding the behavior of a program by examining its code or its execution. We briefly introduce the three families of techniques relevant to this work.

Static analysis. Static analysis examines the code of a binary without executing it, reasoning about its behavior from the machine instructions alone. Examples of tools that perform static analysis are Ghidra [5], IDA Pro [27], and Radare2 [3] that automate disassembly, control-flow recovery, and decompilation into high-level languages.

Dynamic analysis. Dynamic analysis observes a program while it runs. Debuggers such as GDB [2] and emulators such as Unicorn [4] allow inspection of the architectural state between instructions. More specialized tools can also model microarchitectural behavior. The gem5 [11] simulator provides a cycle-accurate model of the processor, including a detailed microarchitectural model, but has been proven not suitable for accurately emulating weird machines [31]. The WeMu [31] emulator faithfully reproduces the microarchitectural effects exploited by μWMs, exposing their computation through execution traces.

Symbolic execution. Symbolic execution is a form of static analysis in which selected inputs are replaced by symbolic variables, so that the output is computed as a function of those variables rather than as a concrete value. angr [29] is a widely used framework for symbolic execution of binaries. angr leverages an execution engine which is capable of emulating instructions on an internal representation of the program state, which can consist of both concrete values (i.e., known numbers) or symbols (i.e., variables), and it gives the user the ability to intercept relevant events during program execution (e.g., memory loads or stores) through *breakpoints*.

3 Flexo

Flexo [36] is a μWM design that allows automated construction of weird gates that compute any Boolean function of N inputs[4]. Flexo is combined with a compilation toolchain that translates a functions written in a subset of a high-level language (e.g. C) into weird machines. At a high level, the toolchain composes multiple *Flexo Weird Gates* into a *Flexo Weird Function*, connecting them by means of weird registers. A *weird register* is the basic memory element in a Flexo μWM: it is identified by a memory address and its 1-bit value is the cache residency of that address (i.e. 1 if the memory address is cached, 0 otherwise). In order to set a weird register, it is possible to make a load from, or a store to, its address, while to reset it its address must be evicted from the cache (e.g., using the x86 instruction `clflush`). For the sake of simplicity, we use the term Flexo to refer to both the μWM construction and the compilation toolchain.

3.1 Flexo Weird Gates

A Flexo Weird Gate is the basic building block of Flexo μWMs. Each gate is capable of computing an arbitrary N-to-1 Boolean function. Inputs and outputs are *differentially encoded*, where each *wire* $\mathbf{w} = (w, \overline{w})$ is a pair of weird registers carrying a Boolean value and its complement. An N-input gate therefore consumes $2N$ weird registers and produces one output wire, i.e., two weird registers holding dual values.

[4] N is limited by microarchitectural constraints to $N \leq 4$.

Flexo exploits Return Stack Buffer (RSB)-based misprediction to create a transient window which can execute a Flexo weird gate. Within that window, the Boolean function is encoded through a chain of *dependent* memory loads. Each load probes a weird register, and if any register in the chain is unset (i.e, the memory is not cached), the additional latency is sufficient to saturate the transient window, preventing the output weird register from being set (i.e, the output is not cached). Conversely, *non-dependent* loads can execute in parallel, thanks to out-of-order execution. Combining these two effects allows the gate to realise any Boolean function in Disjunctive Normal Form (DNF): each conjunction term corresponds to a dependent chain, while the individual chains target the same output weird register independently. Since both polarities of every input are available through differential encoding, the gate can express all 2^{2^N} possible N-to-1 Boolean functions. Producing a differentially encoded output in turn requires two subgates, one computing f and the other $\neg f$, so that downstream gates again receive both polarities of the result.

Listing 1 shows a Flexo XOR gate with input wires $\mathbf{i_1} = (\texttt{rsi}, \texttt{rdi})$ and $\mathbf{i_2} = (\texttt{r9}, \texttt{r8})$ and output wire $\mathbf{o} = (\texttt{rbx}, \texttt{r11})$. Each register holds the address of a weird register; for brevity, we use the same name to refer to its Boolean value. Line 1 triggers an RSB misprediction, causing all subsequent instructions to execute transiently. Lines 2–5 load the four input weird registers, zeroing the architectural registers in the process.[5] The gate's logic is encoded in the `lea`/`mov` pairs that follow. Each `lea` computes a base address plus a zeroed register, reducing to just the base (e.g., line 6: $\texttt{r10} \leftarrow \texttt{rbx} + \texttt{rdi} = \texttt{rbx}$). The subsequent `mov` loads from that base plus another zeroed register (e.g., line 7: load from $\texttt{r10} + \texttt{r9} = \texttt{rbx}$), creating a data dependency on two input loads. If either input weird register is unset (not cached), the latency of its corresponding `movzx` is enough to exhaust the transient window. Consequently, the output weird register at `rbx` is set only when both `rdi` and `r9` are set (cached)—computing $\texttt{rbx} = \texttt{rdi} \wedge \texttt{r9}$. Similarly, lines 8–9 form an independent chain that computes $\texttt{rbx} = \texttt{rsi} \wedge \texttt{r8}$. Since both chains target the same weird register and are independent, out-of-order execution allows either to succeed, giving $\texttt{rbx} = (\texttt{rdi} \wedge \texttt{r9}) \vee (\texttt{rsi} \wedge \texttt{r8})$. Lines 10–13 similarly compute $\texttt{r11} = (\texttt{rdi} \wedge \texttt{r8}) \vee (\texttt{rsi} \wedge \texttt{r9})$. Substituting by taking into account the differential encoding ($\texttt{rdi} = \neg\texttt{rsi}$, $\texttt{r8} = \neg\texttt{r9}$):

$$\texttt{rbx} = (\neg\texttt{rsi} \wedge \texttt{r9}) \ \vee \ (\texttt{rsi} \wedge \neg\texttt{r9}) = \texttt{rsi} \oplus \texttt{r9},$$
$$\texttt{r11} = (\neg\texttt{rsi} \wedge \neg\texttt{r9}) \ \vee \ (\texttt{rsi} \wedge \texttt{r9}) = \neg(\texttt{rsi} \oplus \texttt{r9}).$$

3.2 Flexo Weird Function

A Flexo Weird Function connects multiple Flexo Weird Gates into a Boolean circuit. It is responsible for encoding inputs into weird registers, executing the circuit in topological order, and finally reading results out of weird registers into architectural state, once the circuit completes. To read a weird register, Flexo

[5] The design of Flexo Weird Gates assumes that memory loads return 0

```
call    trigger_RSB_misprediction
movzx   rdi, byte ptr[rdi]
movzx   rsi, byte ptr[rsi]
movzx   r8, byte ptr[r8]
movzx   r9, byte ptr[r9]
lea     r10, [rbx + rdi]
mov     r10b, byte ptr[r10 + r9]
lea     r10, [rbx + rsi]
mov     r10b, byte ptr[r10 + r8]
lea     r10, [r11 + rdi]
mov     r10b, byte ptr[r10 + r8]
lea     r10, [r11 + rsi]
mov     r10b, byte ptr[r10 + r9]
```

Listing 1: A Flexo XOR Gate

uses two `rdtscp` instructions to measure the latency of a load from the register's address. Before invoking a gate, the Weird Function flushes the gate's output weird registers and initializes any input weird registers that were not already set by a previous gate. When a wire feeds into multiple gates, a *repeater*, i.e., a weird gate that replicates one input wire into multiple output wires, duplicates it into independent copies.

Figure 1 shows a circuit with inputs **a**, **b**, **c**, **d** and outputs **o**, **p**. Execution begins with gate A, which computes $\mathbf{b} \vee \mathbf{c}$. Flexo initializes the weird-register pairs encoding **b** and **c**, flushes the two output registers of A, and invokes the gate by triggering the RSB-misprediction. Transient execution loads the appropriate output location, yielding the differentially-encoded results $t_A = b \vee c$ and $\overline{t}_A = \overline{b} \wedge \overline{c}$. Since the output of A is consumed by two subsequent gates, a repeater R is invoked to duplicate the wire, producing $t_R = t_{R'} = t_A$ and $\overline{t}_R = \overline{t}_{R'} = \overline{t}_A$. No additional input initialization is required at this stage, as R reads directly from the output registers produced by A and propagates their values to two new weird register pairs. Gate B then computes $t_B = t_R \wedge a$. Only the input **a** must be initialized, as t_R is already encoded in weird registers. Similarly, gate C computes $t_C = t_{R'} \vee d$, requiring initialization of **d** but using the other replicated output of A. After all gates have executed, Flexo reads the weird registers corresponding to the circuit outputs, t_B and t_C, and writes the result in **o** and **p**, respectively.

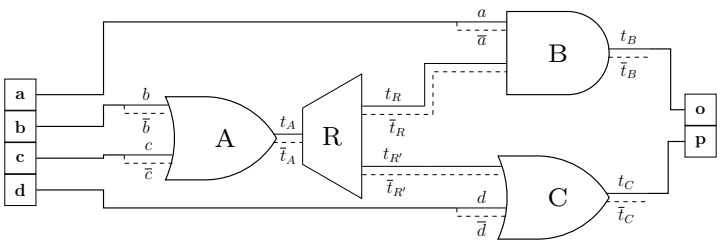

Fig. 1: Flexo circuit example

4 Challenges in Reverse Engineering μWMs

μWMs perform computation entirely through microarchitectural side-effects, which existing binary analysis tools—designed around architectural semantics—do not capture. In this section, we identify the challenges inherent to reverse engineering Flexo μWMs, discuss why current tools fall short, and outline how 0xELF addresses them.

4.1 Challenges

C1 – Identification. Identify the constituent elements of a Flexo μWM within a binary, i.e., the weird gates, the weird functions, and the weird registers they operate on.

C2 – Function Extraction. Determine the Boolean function each weird gate computes by analysing RAW dependencies between instructions with respect to their effect on cache state during transient execution.

C3 – Compositional Analysis. Reconstruct how weird gates are connected via wires to form a complete circuit, including how function inputs are encoded into weird registers and how circuit outputs are read back into architectural state.

Successfully addressing these challenges allows the systematic recovery of the computation performed by Flexo μWMs as a combinational circuit, enabling understanding and further analysis at a higher level of abstraction.

4.2 Limitations of Existing Approaches

Existing binary analysis techniques fail to uncover the computation performed by μWMs. Standard reverse engineering tools, like Ghidra [5] or IDA Pro [27], reason exclusively about the architectural semantics of machine instructions and therefore cannot recover microarchitectural side-effects without extensive modifications. Likewise, debuggers follow the architectural flow of the program and typically introduce timing perturbations that may disrupt μWMs computation [36]. Even tools that do model microarchitectural state, such as WeMu [31], can only observe gate outputs for concrete inputs and cannot extract the Boolean function a gate computes.

4.3 Our Approach

To address these challenges, we develop 0xELF, a framework capable of extracting the combinational circuit implemented by a Flexo μWM using symbolic execution. It takes as input the address of a suspected Flexo Weird Function and an upper bound on the number of its parameters,[6] both provided by an expert user.

[6] If a function has N parameters, any number greater than or equal to N will do

To address **C1**, `0xELF` emulates an RSB to detect RSB mispredictions and models the cache as an infinite address space that tracks the cache state of each relevant memory address. This allows it to automatically identify weird gates and the associated weird registers.

For **C2**, `0xELF` symbolically executes each weird gate to obtain an abstract representation of every instruction's effect on the architecture. Using this information, it builds a *RAW dependencies forest*, a data structure that captures the dependencies between gate instructions, from which the Boolean function can be reconstructed in a Disjunctive Normal Form (DNF).

For **C3**, `0xELF` traces data flow across the weird function using a combination of static analysis and symbolic execution. It tracks how function arguments are encoded into weird registers, how those registers serve as gate inputs and outputs, and how the final results are decoded back into architectural state.

Given these components, `0xELF` produces a gate-level netlist equivalent to the circuit implemented by the weird function.

5 `0xELF` Internals

In this section, we present the design of `0xELF` in detail. We first provide an overview of the working flow of `0xELF`, describe how it extracts the Boolean function of an individual weird gate (cf. Section 5.1), then how it analyzes a complete weird function to detect gates, trace their wiring, and reconstruct the implemented circuit (cf. Section 5.2).

`0xELF` takes as input the address of a suspected Flexo Weird Function and an upper bound on the number of its parameters, both provided by an expert user. It symbolically executes the function from its entry point, using a simulated RSB to detect weird gates as they are encountered. For each detected gate, `0xELF` first checks whether its Boolean functions have already been computed. If not, it spawns a separate symbolic execution context to extract the Boolean functions the gate computes. The results are saved so that gates reused at multiple call sites are only analyzed once. The extracted functions are then *wired* into the weird function-level state—their symbolic variables are concretized using the current register values—to record the gate's effect on the weird registers of the function. Execution continues until the function returns, at which point `0xELF` reconstructs the full circuit. The overall flow of `0xELF` is depicted in Figure 2.

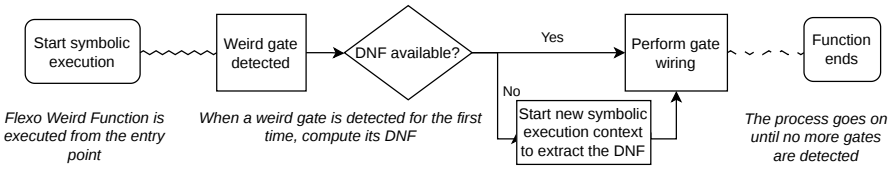

Fig. 2: High-level workflow of `0xELF`

5.1 Analyzing Flexo Weird Gates

Given the instruction sequence of a single weird gate, 0xELF extracts its Boolean function in three steps: it symbolically executes the gate to record which registers and memory addresses each instruction reads from and writes to, builds a RAW dependency forest from those records, and extracts the DNFs directly from the forest structure.

Symbolic Execution. To analyze a weird gate, 0xELF creates a fresh execution state S in which every general-purpose register (except for the stack and instruction pointers) is initialized to a distinct symbol (e.g., $\texttt{rax} = s_{\texttt{rax}}$, $\texttt{rbx} = s_{\texttt{rbx}}$, etc.). Making the registers symbolic ensures that all addresses computed during the gate's execution remain symbolic expressions rather than concrete values, which is essential for extracting the gate's Boolean function independently of any particular input.

Before execution begins, 0xELF configures angr [29] breakpoints to intercept every register read, register write, memory load, memory store, and instruction execution within the gate. Additionally, it configures angr so that memory loads always return zero, thereby avoiding symbolic memory dereferences but keeping execution consistent with the design of Flexo Weird Gates where memory reads yield zero. Then, it symbolically executes the gate on state S from the first instruction to the last, recording the instruction sequence $P = \{p_0, p_1, \ldots, p_n\}$ and populating three maps for each instruction $p \in P$:

$$\begin{aligned}\mathcal{R}(p) &= \{r \mid \text{instruction } p \text{ reads register } r\},\\ \mathcal{W}(p) &= \{r \mid \text{instruction } p \text{ writes register } r\},\\ \mathcal{M}(p) &= \{a \mid \text{instruction } p \text{ accesses memory address } a\}.\end{aligned}$$

Since registers hold symbols, every address in $\mathcal{M}(p)$ is a symbolic expression over the initial state S. These three maps provide all the information 0xELF needs to construct the RAW dependency forest described next.

RAW Dependency Forest. As described in Section 3.1, the Boolean function of a Flexo Weird Gate is encoded in the dependency structure of its instructions: dependent chains implement conjunction, and independent chains targeting the same output implement disjunction. To recover this structure, 0xELF builds a directed forest, $F = (P, E)$, using $\mathcal{R}$ and $\mathcal{W}$, that captures the Read-After-Write (RAW) dependencies between instructions. An edge $(p, p') \in E$ exists iff p reads a register that was last written by p':

$$(p, p') \in E \iff \exists\, r \in \mathcal{R}(p) \cap \mathcal{W}(p') \text{ s.t. } \forall\, \tilde{p} \text{ between } p' \text{ and } p,\ r \notin \mathcal{W}(\tilde{p}).$$

The resulting structure has multiple roots, one per output chain, forming a collection of trees that we refer to as the *RAW forest.*[7] Within each tree, instruc-

[7] Strictly, if nodes are shared across trees, the structure becomes a Directed Acyclic Graph (DAG); however, DNF extraction treats each root's reachable set independently, so we reason about it as a forest without loss of generality.

tions must execute in order, from the leaves to the root, whereas instructions in different trees are independent and may execute in any order, or in parallel, via out-of-order execution. Not all instructions in the forest interact with weird registers. Instructions that do not access memory (e.g., `lea` in the XOR gate of Listing 1) serve only to propagate address values between registers and have no direct effect on cache state. `0xELF` therefore prunes every node p with $\mathcal{M}(p) = \emptyset$, reconnecting edges to preserve reachability. If two nodes were reachable from one another in the original forest, they remain reachable in the pruned one. We refer to the result as the *pruned forest*.

DNF Extraction. The pruned forest directly encodes the gate's Boolean function. The roots of the forest correspond to the gate's output weird registers, since setting an output is the last memory operation in each dependency chain. The nodes reachable from each root are the instructions that read input weird registers, whose latency determines whether the chain completes within the transient window. Every node reachable from a root must execute before the root can, so the entire chain must complete within the transient window for the output to be set. Each weird register is uniquely identified by its memory address, so `0xELF` assigns a Boolean variable to each distinct address in the pruned forest. For brevity, we use memory addresses as Boolean variables in the remainder of this section.

Let $B = \{\mathcal{M}(p) \mid p \text{ is a root}\}$ be the set of output addresses. For each $b \in B$, let $R(b) = \{p \mid p \text{ is a root with } \mathcal{M}(p) = b\}$ be the set of roots targeting b, and let $C(r) = \{p \mid p \text{ is reachable from } r\}$ be the descendants of root r in the pruned forest. Each root $r \in R(b)$ represents one minterm: all nodes in $C(r)$ must complete for r to set b. Multiple roots targeting the same address b represent disjunction, since their chains are independent and any one succeeding suffices. The Boolean function for output b is therefore:

$$\mathrm{DNF}(b) = \bigvee_{r \in R(b)} \bigwedge_{p \in C(r)} \mathcal{M}(p) \tag{1}$$

Illustrative Example. We illustrate the full procedure on the weird OR gate in Listing 2, which takes two input wires and produces one output wire.

Map collection. `0xELF` creates a fully symbolic state (`rdi` $= s_{\texttt{rdi}}$, `rsi` $= s_{\texttt{rsi}}$, etc.) and executes the gate, producing the maps in Figure 3. For instance, line 5 (`mov r10b, byte ptr [rbx + r9]`) reads registers `rbx` and `r9`, writes `r10`, and accesses memory address $s_{\texttt{rbx}} + 0 = s_{\texttt{rbx}}$ (since `r9` was zeroed by line 4).

Forest construction. `0xELF` builds the RAW forest from $\mathcal{R}$ and $\mathcal{W}$. Consider line 5: it reads `r9`, which was last written by line 4, producing the edge $5 \to 4$. It also reads `rbx`, but no instruction in the gate writes `rbx`, so no edge is created for that register. Similarly, line 7 reads `rdi` last written by line 1, producing

another edge. Line 8 reads `r10` (last written by line 7) and `r8` (last written by line 3), producing two edges. The complete edge set is:

$$E = \{5 \to 4,\ \ 6 \to 2,\ \ 8 \to 7,\ \ 7 \to 1,\ \ 8 \to 3\}.$$

Pruning. Line 7 (`lea r10, [r11 + rdi]`) does not access memory, so $\mathcal{M}(7) = \emptyset$ and it is pruned. Since 8 reaches 1 through 7 (via register `r10` then `rdi`), the edge is reconnected: $8 \to 1$. The pruned edge set becomes:

$$E' = \{5 \to 4,\ \ 6 \to 2,\ \ 8 \to 1,\ \ 8 \to 3\}.$$

DNF extraction. The pruned forest has three roots: lines 5 and 6, both of which access $s_{\texttt{rbx}}$, and line 8, which accesses $s_{\texttt{r11}}$. For output $b = s_{\texttt{rbx}}$, there are two roots: $R(s_{\texttt{rbx}}) = \{5, 6\}$. Root 5 has one descendant, 4, with $\mathcal{M}(4) = s_{\texttt{r9}}$, giving the minterm $s_{\texttt{r9}}$. Root 6 has one descendant, 2, with $\mathcal{M}(2) = s_{\texttt{rsi}}$, giving the minterm $s_{\texttt{rsi}}$. The two minterms are ORed together: $\mathrm{DNF}(s_{\texttt{rbx}}) = s_{\texttt{r9}} \vee s_{\texttt{rsi}}$. For output $b = s_{\texttt{r11}}$, there is one root: $R(s_{\texttt{r11}}) = \{8\}$. It has two descendants, 1 ($\mathcal{M}(1) = s_{\texttt{rdi}}$) and 3 ($\mathcal{M}(3) = s_{\texttt{r8}}$), giving a single minterm with both variables ANDed as $\mathrm{DNF}(s_{\texttt{r11}}) = s_{\texttt{rdi}} \wedge s_{\texttt{r8}}$.

```
1 movzx   rdi, byte ptr [rdi]
2 movzx   rsi, byte ptr [rsi]
3 movzx   r8, byte ptr [r8]
4 movzx   r9, byte ptr [r9]
5 mov     r10b, byte ptr [rbx + r9]
6 mov     r10b, byte ptr [rbx + rsi]
7 lea     r10, [r11 + rdi]
8 mov     r10b, byte ptr [r10 + r8]
```

Listing 2: The transient part of a Flexo-generated weird OR

P	$\mathcal{R}$	$\mathcal{W}$	$\mathcal{M}$
1	{rdi}	{rdi}	$\{s_{\texttt{rdi}}\}$
2	{rsi}	{rsi}	$\{s_{\texttt{rsi}}\}$
3	{r8}	{r8}	$\{s_{\texttt{r8}}\}$
4	{r9}	{r9}	$\{s_{\texttt{r9}}\}$
5	{rbx, r9}	{r10}	$\{s_{\texttt{rbx}}\}$
6	{rbx, rsi}	{r10}	$\{s_{\texttt{rbx}}\}$
7	{r11, rdi}	{r10}	$\emptyset$
8	{r10, r8}	{r10}	$\{s_{\texttt{r11}}\}$

Fig. 3: Maps built by 0xELF

Handling Dual Encoding. At the gate level, 0xELF cannot determine which weird registers belong to the same wire for all weird registers in the gate. For a standard gate with exactly two output addresses, 0xELF can determine that they are both part of the same wire, but cannot infer which one carries the same value of the wire and which one carries the negated one. Therefore 0xELF assigns a Boolean attribute $\mathcal{N}$ such that $\mathcal{N}(w_1) = \neg\mathcal{N}(w_2)$. This choice is arbitrary and is for keeping consistency when multiple instances of the same gate are used in a circuit. On the other hand, *Repeaters*, i.e., gates that replicate one input wire into multiple output wires, are handled differently. The $\mathcal{N}$ assignment is made on the input weird register pair, since they trivially form a single wire, and it is propagated to the outputs in the same way as the inputs.

5.2 Analyzing Weird Functions

While gate analysis (Section 5.1) operates on an isolated instruction sequence, function-level analysis must handle the full execution of a Flexo Weird Function.

Specifically, it needs to detect gates as they are triggered, track the state of weird registers, and capture how outputs are read back into architectural state. `0xELF` performs all three tasks in a single symbolic execution pass, using angr breakpoints on every instruction (for detecting cache flush and calls) and memory access. Throughout this pass, `0xELF` maintains a map $\mathcal{V}(a) = v$ that records the current value of the weird register at address a. In the simplest case, a memory load or store to address a sets $\mathcal{V}(a) = 1$, and a `clflush` targeting a sets $\mathcal{V}(a) = 0$. Entries in $\mathcal{V}$ can also carry richer metadata, as described in the following subsections.

Gate Detection and Wiring. `0xELF` maintains a simulated RSB, implemented as an infinite stack. On every `call`, it pushes the address of the following instruction on the RSB. On encountering a `ret`, it pops the top entry and compares it with the current instruction pointer. A mismatch indicates an RSB misprediction. The instructions between the popped address and the current IP form a weird gate, which is analyzed as described in Section 5.1.

Once the gate's DNFs are extracted, `0xELF` *wires* the gate by substituting the current register values into the symbolic variables of the DNFs, thus concretizing them to actual weird register addresses. It then updates $\mathcal{V}$ for each output weird register. For a standard (non-repeater) gate with output register pair $(w, \overline{w})$:

$$\mathcal{V}(w) = (\texttt{Gate}\text{: } id,\ \texttt{negated}\text{: } \mathcal{N}(w),\ \texttt{dual}\text{: } \overline{w}),$$
$$\mathcal{V}(\overline{w}) = (\texttt{Gate}\text{: } id,\ \texttt{negated}\text{: } \mathcal{N}(\overline{w}),\ \texttt{dual}\text{: } w),$$

where $\mathcal{N}(w)$ is the attribute assigned during gate analysis. For repeaters, the `dual` field is initially unresolved.

At this point, `0xELF` also resolves the `dual` field for the current gate's input weird registers. For each input lacking a `dual`, such as weird registers set by a repeater or an input argument to the function, it uses $\mathcal{V}$ to search another input weird register of the current gate with the same source (same `Gate` or same argument bit $a_{k,b}$) but opposite `negated` attribute, and pairs them.

Tracking Input Encoding. Flexo encodes each bit of a function argument into a wire by selectively loading and flushing two weird registers. For the b-th bit of the k-th argument, denoted $a_{k,b}$, a Flexo Weird Function loads address $f_1(a_{k,b})$ and flushes address $f_2(a_{k,b})$, where f_1 and f_2 are functions that depend on $a_{k,b}$[8]. When `0xELF` encounters a load or store to a symbolic address of the form $f(a_{k,b})$, it resolves both branches and updates $\mathcal{V}$:

$$\mathcal{V}(f(0)) = (a_{k,b},\ \texttt{negated}\text{: }\texttt{True},\ \texttt{dual}\text{: } \bot),$$
$$\mathcal{V}(f(1)) = (a_{k,b},\ \texttt{negated}\text{: }\texttt{False},\ \texttt{dual}\text{: } \bot).$$

The `dual` field is left unresolved ($\bot$) because the same argument bit may be encoded into multiple weird register pairs for use in different gates. A `clflush` to address $f(a_{k,b})$ produces the same entries with the `negated` attributes swapped.

[8] For example, Flexo may use $f_1(x) = B + 256x$ and $f_2(x) = (B + 256) - 256x$, so that $f_1(0) = f_2(1) = B$ and $f_1(1) = f_2(0) = B + 256$.

Reading Circuit Output. The remaining challenge is to determine how the values of specific weird registers are read and written back into architectural state via reference parameters. As described in Section 3.2, Flexo reads a weird register by taking a timestamp with `rdtsc`/`rdtscp`, loading from the register's address, and taking a second timestamp to measure the access latency against a threshold T. The result is then stored to one of the function's output parameters, passed by reference. 0xELF captures this pattern symbolically. It intercepts every `rdtsc`/`rdtscp` instruction and makes it return a fresh symbol, representing a timestamp read. When a load to address a occurs between two such symbols t_0 and t_1, 0xELF records $\mathcal{T}(t_0, t_1) = a$[9]. The threshold comparison then produces an expression that depends on (t_0, t_1).

0xELF monitors all memory stores to addresses derived from the function's symbolic arguments. Each such store is recorded in a map $\mathcal{O}(a) = e$, where a is the output address and e is the stored expression. If a bit of e is computed as an expression involving timestamp symbols (t, t'), 0xELF uses $\mathcal{T}(t, t')$ to identify which weird register was read and $\mathcal{V}$ to recover that register's value, thereby connecting the circuit output to the gate that produced it.

5.3 Circuit Reconstruction

Once execution completes, 0xELF has the DNFs of every gate, their wiring, and the mapping from circuit outputs to weird registers. It remains to compute truth tables for each gate and emit the final netlist.

Truth Table Generation. For each circuit output, 0xELF inspects the `negated` attribute of the corresponding weird register. If `negated` = `True`, the originating gate is marked ●; all other gates are marked ○, except repeaters already marked ●, which are promoted to ◐. The marking determines which DNF(s) 0xELF evaluates: ○ uses the non-negated output's DNF(s), ● the negated output's DNF(s), and ◐ both[10]. For each selected DNF, 0xELF enumerates all 2^n assignments to the n non-negated input weird registers, assigns each dual the opposite value, and evaluates the DNF to produce the truth table.

Polarity Invariance. For inner gates—those whose outputs feed other gates rather than circuit outputs—the `negated` assignment made during gate analysis may be wrong. 0xELF may have swapped the negation attribute of the output wire. We now argue that this does not affect the correctness of the reconstructed circuit. If 0xELF inverts the polarity of an inner gate's output, two things happen simultaneously: the gate's truth table is computed from the complementary DNF, and the downstream gate receives the swapped wire, so every input that was treated as non-negated becomes negated and vice versa. Concretely, the

[9] 0xELF does not update $\mathcal{V}(a)$ for this load, preserving the weird register's value from the computation phase.

[10] A repeater will have multiple non-negated and negated DNFs.

downstream gate's DNF expects the non-negated input but receives the negated one, so it effectively evaluates the complementary function of what it would have computed under the correct assignment. Combined with the inverted truth table of the upstream gate, the two negations cancel each other, and the composed input–output mapping is identical to the one obtained under the correct polarity assignment.

Illustrative Example. Consider two OR gates A and B wired in series (Figure 4). Without taking differential encoding into account, gate A computes $\mathrm{DNF}(o_1) = v_1 \vee v_3$ and $\mathrm{DNF}(o_2) = v_2 \wedge v_4$ and gate B computes $\mathrm{DNF}(o_3) = v_5 \vee v_7$ and $\mathrm{DNF}(o_4) = v_6 \wedge v_8$. Since both gates compute an OR, the `negated` attribute $\mathcal{N}(o_1) = \mathcal{N}(o_3) =$ `False` and $\mathcal{N}(o_2) = \mathcal{N}(o_4) =$ `True`.

Assume that during the DNFs extraction of the gates, `0xELF` makes a mistake and assigns $\mathcal{N}(o_1) =$ `True` and conversely $\mathcal{N}(o_2) =$ `False`, while assigns $\mathcal{N}(o_3)$ and $\mathcal{N}(o_4)$ correctly. So after the circuit is completely analyzed, `0xELF` gets the map $\mathcal{V}$ as in table Table 1. Therefore, at the end of the symbolic execution, since o_3 is read and its `negated` attribute is `False`, gate B is marked as ○, and gate A is marked as ○ since its outputs go into gate B.

For the truth table of gate A, `0xELF` assigns v_1 and v_3 (the non-negated input weird registers of the gate) all possible values and computes $o_2 = v_2 \wedge v_4$, Keeping the wiring into account, the resulting truth table is reported in Table 2. Then `0xELF` processes gate B, and assigns to the non-negated inputs, v_6 and v_7 all the possible values and computes $o_3 = v_5 \vee v_7$. `0xELF` thus obtains the truth table in Table 3.

In the end, `0xELF` will output the truth tables reported in Table 4. To verify the equivalence of the circuit with the correct assignments for $\mathcal{N}$, it suffices to show that the only combination that produces $\mathbf{o} = 0$ is $\mathbf{a} = \mathbf{b} = \mathbf{c} = 0$. Note that only one combination in the truth table for B produces 0 as an output, which happens when $\mathbf{c} = 0$ and $t_A = 1$ and the only combination in the truth table for A which gives $t_A = 1$ is $\mathbf{a} = \mathbf{b} = 0$.

Circuit Output. The output of `0xELF` is a circuit equivalent to the one implemented by the Flexo weird function. `0xELF` returns it by computing and printing all the computed truth tables encoded as a BLIF [30].

6 Evaluation and Results

We now evaluate `0xELF` on a wide range of circuits, including simple examples from the Flexo repository [37] and more complex circuits from the EPFL combinational benchmark suite [7, 8]. We further assess the performance of `0xELF` in terms of accuracy, generality, and scalability.

6.1 Experimental Setup

Benchmarks. We draw test cases from two sources. The first is the set of examples distributed with the Flexo repository [37], which includes basic gates, adders

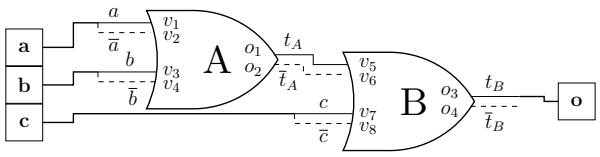

Fig. 4: Two weird OR gates analyzed by 0xELF. The outputs of the first one have the assignment of the negated attribute swapped.

w	Val.	Neg.	Dual
v_1	a	F	v_2
v_2	a	T	v_1
v_3	b	F	v_4
v_4	b	T	v_3
v_5	A	T	v_6
v_6	A	F	v_5
v_7	c	F	v_4
v_8	c	T	v_3

Table 1: Values for $\mathcal{V}$

v_1	v_3	v_2	v_4	o_2
0	0	1	1	1
0	1	1	0	0
1	0	0	1	0
1	1	0	0	0

Table 2: Truth table for A

v_6	v_7	v_5	v_8	o_3
0	0	1	1	1
0	1	1	0	1
1	0	0	1	0
1	1	0	0	1

Table 3: Truth table for B

a	b	$\bar{t}_A$
0	0	1
0	1	0
1	0	0
1	1	0

$\bar{t}_A$	c	**o**
0	0	1
0	1	1
1	0	0
1	1	1

Table 4: Output truth tables for the circuit

of width 8–64, multipliers, an ALU, SHA-1, AES, and the Simon cipher. The second is the EPFL combinational benchmark suite [7, 8], from which we select all circuits in the Arithmetic and Random categories. We exclude DIV (56 059 gates) and HYP (138 176 gates) due to their large size. In total, the evaluation covers circuits ranging from a single gate to over 23 000 gates.

Compilation and ground truth. Flexo examples are compiled following the instructions in their repository [37]. EPFL benchmarks are compiled from Verilog using the Flexo parameter WM_CIRCUIT_FILE and an empty stub function. All other compilation parameters are left at their default values, including WR_TYPE which is the encoding of the weird function input parameters into the weird registers. The default value for this parameter makes Flexo encode such inputs using expressions f_1 and f_2 as explained in Section 5.2. During compilation, Flexo invokes Yosys [38] to synthesize the circuit as a sequence of LUTs. We intercept this step to extract the synthesized netlist as a BLIF file, which serves as the ground truth for equivalence checking.

Equivalence checking. We verify correctness using ABC [10] tool, which performs combinational equivalence checking (CEC) between two BLIF descriptions. Since 0xELF recovers circuit structure but not the original argument names, we manually rename the inputs and outputs of the extracted circuit to match those in the ground truth before invoking ABC.

Configuration. All experiments are run on a machine running Debian 12, equipped with an Intel Xeon Gold 6252N processor (2.30GHz) and 64 GiB of RAM. 0xELF is configured with 6 symbolic input arguments for all circuits except CTRL, which requires 30. For circuits whose outputs include leading zero bits that are not set

explicitly by the weird function, `0xELF` is run with the *ignore leading zeros* (`IZ`) option, which strips constant-zero output bits before equivalence checking. Note that none of the μWMs were executed on real hardware; equivalence is assessed against the synthesized netlist, not against microarchitectural behavior.

6.2 Results

We discuss the overview of the results in terms of correctness, generalization, and scalability. The detailed results are presented in Table 5.

Correctness. `0xELF` successfully extracts an equivalent circuit for every benchmark compiled with default parameters, except for the ROUTER circuit from the EPFL benchmark suite. The ROUTER circuit explicitly sets certain leading bits of an output to zero, but the output width is not byte-aligned. As a result, `0xELF` cannot distinguish the leading zeros that are part of the computation from those introduced for byte alignment, and the extracted circuit retains extra constant-zero outputs. After manually removing the additional leading zeros, ABC confirms equivalence for ROUTER as well. In summary, `0xELF` achieves full correctness on all tested circuits, with one requiring minor manual post-processing.

Ignore leading zeros. The `IZ` option is required for circuits which logical output width is narrower than the architectural word size (e.g., a 6-bit result stored in a byte). Without `IZ`, constant-zero bits appear as additional outputs in the extracted circuit, causing a mismatch with the ground truth. As shown in Table 5, `IZ` is needed primarily for some EPFL benchmarks and for the ALU, where output widths do not align with byte boundaries.

Scalability. While execution time and memory consumption generally grow with the number of gates in the circuit, this scaling is not linear because `0xELF` saves the computed Boolean formulas from gates and reuses it if the gate is used in multiple call sites. For single-gate circuits, `0xELF` completes in under ten seconds using approximately 200 MiB of memory. For medium-sized circuits such as BAR (2304 gates) the AES rounds (∼2500 gates) and SIMON (4322 gates), extraction takes 9–25 minutes and requires 1–3 GiB. The largest circuits in our evaluation—LOG2 (23 710 gates) and SQRT (20 711 gates)—require approximately 2 hours, with peak memory around 11–13 GiB. The dominant cost is symbolic execution of the weird function, which must process every gate invocation and track the symbolic state of all weird registers.

7 Discussion and Limitations

We now discuss some of the limitations and future directions for `0xELF`.

Table 5: 0xELF results. The Weird Function column is shown only for binaries containing multiple weird functions; a dash indicates a single function. EQ denotes equivalence with the ground truth after renaming. IZ indicates whether the ignore-leading-zeros option was enabled (●) or not (○). The number of gates refers to the number of instances of gates, and not the number of different unique gates, present in each function.

Binary	Time mm:ss	Memory MiB	Gates	EQ	IZ
AND	0:06	205.1	1	✓	○
MUX	0:07	212.2	1	✓	○
NAND	0:06	204.9	1	✓	○
NOT	0:06	196.8	1	✓	○
OR	0:06	205.0	1	✓	○
XOR	0:06	205.0	1	✓	○
XOR3	0:07	211.8	1	✓	○
XOR4	0:07	219.6	1	✓	○
ADDER-8	0:29	380.0	21	✓	○
ADDER-16	0:50	518.3	66	✓	○
ADDER-24	1:05	699.6	108	✓	○
ADDER-32	1:30	752.5	150	✓	○
ADDER-40	1:50	933.5	195	✓	○
ADDER-48	2:06	928.9	238	✓	○
ADDER-56	2:37	1011.1	305	✓	○
ADDER-64	2:58	1191.4	369	✓	○

Binary	Time mm:ss	Memory MiB	Gates	EQ	IZ
ADDER	3:16	1950.9	465	✓	●
BAR	11:45	1951.1	2304	✓	○
LOG2	127:23	12943.6	23710	✓	○
MAX	9:52	3305.1	1611	✓	●
MULTIPLIER	81:25	9031.8	15669	✓	○
SIN	24:47	2840.6	4515	✓	●
SQRT	115:04	12519.0	20711	✓	○
SQUARE	62:38	6674.0	12779	✓	○
ARBITER	65:16	6741.8	9557	✓	●
CAVLC	3:48	1051.5	328	✓	●
CTRL	1:07	671.0	67	✓	●
DEC	3:12	539.6	544	✓	●
INT2FLOAT	1:31	841.1	101	✓	●
PRIORITY	3:50	905.4	580	✓	●
ROUTER	1:38	800.1	160	✗	○
VOTER	52:43	11453.6	8381	✓	●

Results for simple Flexo examples

Results for benchmarking circuits

Binary	Weird Function	Time mm:ss	Memory MiB	Gates	EQ	IZ
ALU	-	0:30	440.5	32	✓	●
MUL-8	-	2:21	1387.2	327	✓	○
MUL-16	-	8:02	7840.5	1355	✓	○
SHA1-2BLOCKS	`__weird__sha1_round1`	4:16	1851.8	544	✓	○
SHA1-2BLOCKS	`__weird__sha1_round2`	4:04	1868.3	537	✓	○
SHA1-2BLOCKS	`__weird__sha1_round3`	4:17	1808.5	553	✓	○
SHA1-2BLOCKS	`__weird__sha1_round4`	4:10	1835.0	547	✓	○
SHA1-ROUND	-	4:13	1773.4	544	✓	○
AES-BLOCK	`__weird__aes_first_round`	17:13	3549.0	2636	✓	○
AES-BLOCK	`__weird__aes_last_round`	16:41	2942.6	2669	✓	○
AES-BLOCK	`__weird__aes_round`	15:46	2665.2	2524	✓	○
AES-BLOCK	`__weird__round_key`	5:01	1433.8	669	✓	○
AES-ROUND	-	15:42	1563.7	2524	✓	○
SIMON32	-	24:43	2780.4	4322	✓	○

Results for more complex Flexo examples

Weird Register Encoding. Flexo supports three input encoding schemes, selectable via the `WR_TYPE` compilation parameter: `Dual` (the default), `NoBranch`, and `Baseline`. `Dual` and `NoBranch` both encode inputs using the pair of functions f_1 and f_2 described in Section 5.2. `0xELF` handles both with the same mechanism and achieves identical results. `Baseline`, however, uses branches conditioned on the input argument to decide which weird registers to load or flush. These branches introduce symbolic forks during execution, requiring a modified analysis. We extended `0xELF` to handle this case, achieving a success rate of 78% across all evaluated circuits.

Other Compilation Parameters. To test robustness beyond the default configuration, we compiled the ADDER circuits with all non-default compilation parameters while keeping `WR_TYPE` at its default value, producing a test suite of 56 binaries. `0xELF` successfully extracted a correct circuit from all of them, except when `WR_SYSCALL_RAND` was set to `True`. In this case, `0xELF` did not terminate within a reasonable time, likely due to how angr handles the system call that Flexo uses to generate random numbers during initialization. This can be addressed by hooking the system call with a concrete implementation that returns a random value, which we leave for future work.

Other μWM Designs. `0xELF` is designed for Flexo-generated μWMs and assumes Flexo's gate structure and encoding conventions. Hand-crafted μWMs or those produced by alternative compilers may use different transient-execution triggers, encoding schemes, or gate layouts that `0xELF` does not currently support. Extending `0xELF` to handle other designs would require generalising the gate detection and extraction mechanisms, potentially guided by runtime microarchitectural monitoring tools that could identify the type of μWM before analysis begins. We leave addressing this challenge to future work.

Integration with Hardware Analysis Tools. Since `0xELF` outputs a standard BLIF netlist, its results can be directly consumed by existing hardware analysis frameworks such as ABC [10] and HAL [14]. For instance, ABC can map the look-up tables (LUTs) in `0xELF`'s output to a library of standard gates (AND, OR, NOT, etc.), producing a conventional gate-level representation that may be easier to interpret or to compare against known circuits. This integration demonstrates that `0xELF` does not merely extract a circuit but connects μWM analysis to the broader hardware verification ecosystem.

8 Conclusions

In recent years, μWMs have emerged as effective tools for obfuscating computation and have become more reliable and exhibit ever increasing computational power. Flexo provides the first compilation toolchain for producing μWMs, enabling non-experts to generate μWMs with significantly reduced effort.

In this paper we presented `0xELF`, a framework that performs reverse engineering of Flexo Weird Functions by extracting an equivalent circuit. To do this,

0xELF uses simplified models of the microarchitectural components exploited by Flexo μWMs to perform their computation, namely the cache and the RSB. By combining these models with symbolic execution, 0xELF can track how the input arguments of Flexo Weird Functions are encoded into weird registers and detect weird gates present in the function. To extract the Boolean formulas computed by the detected Flexo Weird Gates, 0xELF leverages symbolic execution to build the dependency graph between memory instructions of the gate and turns it into the set of formulas implemented by the gate in the form of DNFs. Finally, 0xELF handles Flexo differential encoding and produces a circuit which is equivalent to the one implemented by the analyzed Flexo Weird Function.

To sum up, through 0xELF we show that Flexo's obfuscation can be successfully reversed. Additionally, 0xELF produces output in a standard format, which allows leveraging existing hardware analysis tools like ABC [10] and HAL [14] to process 0xELF output for further analysis.

Acknowledgments. We thank the anonymous reviewers for their valuable feedback that have helped us to revise this paper.

This research was supported by an ARC Discovery Project number DP210102670; the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and through project number 560392681.

Bibliography

[1] QEMU – A generic and open source machine emulator and virtualizer. https://www.qemu.org/ (Accessed on Apr 2026)

[2] GDB: The GNU Project debugger. https://www.sourceware.org/gdb/ (Accessed on Feb 2026)

[3] radare2. https://rada.re/n/radare2.html (Accessed on Feb 2026)

[4] Unicorn: The ultimate CPU emulator. https://www.unicorn-engine.org/ (Accessed on Feb 2026)

[5] Agency, N.S.: Ghidra software reverse engineering framework. https://github.com/NationalSecurityAgency/ghidra (Accessed on Jan 2026)

[6] Aldaya, A.C., Brumley, B.B., ul Hassan, S., García, C.P., Tuveri, N.: Port contention for fun and profit. In: IEEE SP, pp. 870–887 (2019), https://doi.org/10.1109/SP.2019.00066

[7] Amarú, L., Gaillardon, P.E., De Micheli, G.: The EPFL combinational benchmark suite. Hypotenuse **256**(128), 214335 (2015), https://si2.epfl.ch/demichel/publications/archive/2015/IWLS15.pdf

[8] Amarú, L., Gaillardon, P.E., De Micheli, G.: The EPFL combinational benchmark suite. https://github.com/lsils/benchmarks (Commit ID 8c832d5)

[9] Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: The SIMON and SPECK lightweight block ciphers. In: DAC, pp. 1–6 (2015), https://doi.org/10.1145/2744769.2747946

[10] Berkeley Logic Synthesis and Verification Group: ABC: A system for sequential synthesis and verification. https://people.eecs.berkeley.edu/~alanmi/abc/ (Accessed on Jan 2026), version 1.01

[11] Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D., Wood, D.A.: The gem5 simulator. SIGARCH Comput. Archit. News **39**(2), 1–7 (Aug 2011), https://doi.org/10.1145/2024716.2024718

[12] Canella, C., Genkin, D., Giner, L., Gruss, D., Lipp, M., Minkin, M., Moghimi, D., Piessens, F., Schwarz, M., Sunar, B., et al.: Fallout: Leaking data on Meltdown-resistant CPUs. In: CCS, pp. 769–784 (2019), https://doi.org/10.1145/3319535.3363219

[13] Dullien, T.: Weird machines, exploitability, and provable unexploitability. IEEE Transactions on Emerging Topics in Computing **8**(2), 391–403 (2017), https://doi.org/10.1109/TETC.2017.2785299

[14] Embedded Security Group: HAL - The Hardware Analyzer. https://github.com/emsec/hal (2019)

[15] Evtyushkin, D., Benjamin, T., Elwell, J., Eitel, J.A., Sapello, A., Ghosh, A.: Computing with time: Microarchitectural weird machines. In: ASPLOS, pp. 758–772 (2021), https://doi.org/10.1145/3445814.3446729

[16] Gruss, D., Maurice, C., Wagner, K., Mangard, S.: Flush+Flush: A fast and stealthy cache attack. In: DIMVA, pp. 279–299 (2016), https://doi.org/10.1007/978-3-319-40667-1_14
[17] Gullasch, D., Bangerter, E., Krenn, S.: Cache games–bringing access-based cache attacks on aes to practice. In: IEEE SP, pp. 490–505 (2011), https://doi.org/10.1109/SP.2011.22
[18] Horowitz, G., Ronen, E., Yarom, Y.: Spec-o-Scope: Cache probing at cache speed. In: CCS, pp. 109–123 (2024), https://doi.org/10.1145/3658644.3690313
[19] Kaplan, D.A.: Optimization and amplification of cache side channel signals. https://arxiv.org/pdf/2303.00122 (2023)
[20] Katzman, D., Kosasih, W., Chuengsatiansup, C., Ronen, E., Yarom, Y.: The gates of time: Improving cache attacks with transient execution. In: USENIX Security, pp. 1955–1972 (2023), https://www.usenix.org/conference/usenixsecurity23/presentation/katzman
[21] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. In: IEEE SP, pp. 1–19 (2019), https://doi.org/10.1145/3399742
[22] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading kernel memory from user space. In: USENIX Security, pp. 973–990 (2018), https://www.usenix.org/conference/usenixsecurity18/presentation/lipp
[23] Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-level cache side-channel attacks are practical. In: IEEE SP, pp. 605–622 (2015), https://doi.org/10.1109/SP.2015.43
[24] Maisuradze, G., Rossow, C.: ret2spec: Speculative execution using return stack buffers. In: CCS, pp. 2109–2122 (2018), https://doi.org/10.1145/3243734.3243761
[25] Morgan, B., Horowitz, G., O'Connell, S., van Schaik, S., Chuengsatiansup, C., Genkin, D., Maennel, O., Montague, P., Ronen, E., Yarom, Y.: Slice+Slice Baby: Generating last-level cache eviction sets in the blink of an eye. In: IEEE SP, pp. 3479–3496 (2025), https://doi.org/10.1109/SP61157.2025.00264
[26] Percival, C.: Cache missing for fun and profit. In: BSDCan Ottawa (2005), http://wistp2007.wistp.org/fileadmin/damiensauveron/Cours/M2/sicom/Attacks/TimingAttack/htt.pdf
[27] hex rays: IDA Pro. https://hex-rays.com/ida-pro (Accessed on Jan 2026)
[28] Schwarz, M., Lipp, M., Moghimi, D., Van Bulck, J., Stecklina, J., Prescher, T., Gruss, D.: ZombieLoad: Cross-privilege-boundary data sampling. In: CCS, pp. 753–768 (2019), https://doi.org/10.1145/3319535.3354252
[29] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: SoK: (state of) the art of war: Offensive techniques in binary analysis. In: IEEE SP (2016), https://doi.org/10.1109/SP.2016.17

[30] University of California, Berkeley: BLIF: Berkeley Logic Interchange Format. https://www.cs.columbia.edu/~cs6861/sis/blif/ (1999)
[31] Vanspauwen, D., Daniel, L.A., Van Bulck, J.: WeMu: Effective and scalable emulation of microarchitectural weird machines. In: uASC (2026), https://doi.org/10.46586/uasc.2026.006
[32] Vila, P., Köpf, B., Morales, J.F.: Theory and practice of finding eviction sets. In: IEEE SP, pp. 39–54 (2019), https://doi.org/10.1109/SP.2019.00042
[33] Wampler, J., Martiny, I., Wustrow, E.: ExSpectre: Hiding malware in speculative execution. In: NDSS (2019), https://doi.org/10.14722/ndss.2019.23409
[34] Wang, P.L., Brown, F., Ronen, E., Paccagnella, R., Wahby, R.S., Yarom, Y.: Transient architectural execution: from weird gates to weird programs. In: IEEE SP (2026), https://doi.org/10.1109/SP63933.2026.00066
[35] Wang, P.L., Brown, F., Wahby, R.S.: The ghost is the machine: Weird machines in transient execution. In: WOOT, pp. 264–272 (2023), https://ieeexplore.ieee.org/abstract/document/10188658
[36] Wang, P.L., Paccagnella, R., Wahby, R.S., Brown, F.: Bending microarchitectural weird machines towards practicality. In: USENIX Security, pp. 1099–1116 (2024), https://www.usenix.org/system/files/usenixsecurity24-wang-ping-lun.pdf
[37] Wang, P.L., Paccagnella, R., Wahby, R.S., Brown, F.: Repository for Flexo code and examples on GitHub. https://github.com/joeywang4/Flexo (Commit ID 00186b4)
[38] Wolf, C.: Yosys open synthesis suite. https://yosyshq.net/yosys/ (Accessed on Jan 2026)
[39] Yarom, Y., Falkner, K.: Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In: USENIX Security, pp. 719–732 (2014), https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom