ABSTRACT
Side-channel attacks such as Spectre rely on properties of modern CPUs that permit discovery of microarchitectural state via timing of various operations. The Weird Machine concept is an increasingly popular model for characterization of emergent execution that arises from side-effects of conventional computing constructs. In this work we introduce Microarchitectural Weird Machines (µWM): code constructions that allow performing computation through the means of side effects and conflicts between microarchitectural entities such as branch predictors and caches. The results of such computations are observed as timing variations. We demonstrate how µWMs can be used as a powerful obfuscation engine where computation operates based on events unobservable to conventional anti-obfuscation tools based on emulation, debugging, static and dynamic analysis techniques. We demonstrate that µWMs can be used to reliably perform arbitrary computation by implementing a SHA-1 hash function. We then present a practical example in which we use a µWM to obfuscate malware code such that its passive operation is invisible to an observer with full power to view the architectural state of the system until the code receives a trigger. When the trigger is received the malware decrypts and executes its payload. To show the effectiveness of obfuscation we demonstrate its use in the concealment and subsequent execution of a payload that exfiltrates a shadow password file, and a payload that creates a reverse shell.

CCS CONCEPTS
• Security and privacy → Hardware attacks and countermeasures; Operating systems security; Side-channel analysis and countermeasures; Malware and its mitigation; • Computer systems organization → Other architectures.

KEYWORDS
Microarchitecture security; weird machines; obfuscation; speculative execution; side channel;

1 INTRODUCTION
The ability to model and classify program’s behavior is fundamental for a vast number of security related tasks. It requires a form of emulator which implements in software a reference model of the target machine.

If this model deviates from the actual machine’s behavior, key properties of many security mechanisms are violated. This can be exemplified by a proof-carrying code framework [2, 25, 44] that allows an arbitrary untrusted executable to run securely on a target platform. Security is established by the target system checking a proof provided along with the executable. The proof ensures the executable cannot perform any activity (or computation) outside of formally specified policy. Any divination between the expected and actual target system behavior effectively violates such a proof.

Many security mechanisms are based on either guaranteeing that the program (1) cannot perform an action from the deny-list, or (2) can only perform actions from the allow-list. Examples of such mechanisms include model checking [11, 24, 30], formal verification [7, 8, 29, 35], taint analysis [15, 21, 45], control flow integrity enforcement [1, 68], malware detection [12, 14, 31, 32, 49, 53, 67] and sandboxing [18, 26, 66].

Consider a sandboxing framework such as Google’s native client [66], where untrusted native code can be safely executed within the trusted context of a browser. Although untrusted code is located side-by-side with sensitive data, the framework prevents sandboxed code from accessing restricted data or performing unexpected control flow alterations. This is done via rewriting the native executable.
to mask potentially dangerous memory accesses and control flow
transfers. To guarantee the modified code is safe, the framework
must be based on a fully accurate model of the target CPU. Undoc-
umented CPU functionality can cause deviations between modeled
and actual behavior of the sandboxed code, resulting in attacks.
Such deviations can be a result of errors [16], misspecifications
or intentional backdoors. For instance, a jmp instruction with
the invalid 0x66 prefix can result in deviation between the actual CPU
behavior and the outputs of a disassembler [19]. The effect is due
to different approaches in handling of invalid prefixes in Intel and
AMD processors potentially providing a tool for hiding malicious
instructions.

Program obfuscation [3, 37, 50, 56] is a general problem of trans-
forming programs to prevent reverse engineering or other forms
of analysis. While it is commonly used to hide malware, it can also
be utilized to conceal benign sensitive code in proprietary applica-
tions [17] or to improve security [6]. A strong obfuscation engine
can be constructed if the obfuscated program utilizes the target
platform features that are outside of platform’s reference model
used by the analyzer.

Recently a number of papers introduced the concept of weird
machines (WM) [5, 9, 20] in attempt to formalize exploits. Accord-
ing to this concept, an exploitable vulnerability not only provides
an access to otherwise protected data but creates a new compu-
tational device (or primitive) with its own interface. Such device
and its interface can be formalized and programmed. Then the
exploit itself can be viewed as a program developed specifically
for this computational device. For instance, a buffer overflow cre-
ates an artifact which is programmed by attacker providing data
to the vulnerable program through normal program’s API calls.
The data then is placed on stack and triggers certain abnormal
behavior that is outside of victim’s program specification. Previous
research has demonstrated that many vulnerabilities, such as buffer
overflow [54] can be utilized as fully programmable machines that
implement Turing-complete languages.

WM primitives can be utilized as powerful obfuscation engines.
Previous research demonstrated presence of such primitives in com-
mon implementation of various software and hardware components.
Programming these WM does not require activating any vulner-
abilities. For instance, Turing-complete WM were built utilizing
little known artifacts inside the page-fault handling hardware [4],
ELF-loader [55] and exception handler [46] mechanisms. These
WM provide nearly ideal obfuscation capabilities. First, they use
computer system’s features that are not identified as dangerous
by antimalware software. Second, they are naturally difficult to
analyze. To the best of our knowledge, no universal WM detection
approach has been proposed.

In this paper we establish a new type of WM implemented using
microarchitectural (MA) components of a CPU, their complex inter-
component interactions and how it effects the latency of common
operations. We call such machines 𝜇WMs (short for microarchitec-
tural weird machines). At a high level, the computation is performed
by executing regular instructions such as memory loads and stores,
jumps and conditional branches and observing execution time. The
WM is constructed from three types of abstract components. Weird
registers (WR) are data storing entities implemented using states
of MA components. Weird gates (WG) are basic computation com-
ponents which transform data stored in WR according to their
logic. WGs utilize entanglement of various MA component states
and their side effects such as aliasing, evictions and speculative
execution. Weird circuits (WCs) are ensembles of WGs and im-
plement more complex logic. We demonstrate that the proposed
computation framework can be used to perform general purpose
computations.

Since reverse engineering and binary analysis tools do not emu-
late MA components, we believe that our framework can be used
as a universal approach for program obfuscation. Moreover, even if
detection tools include the MA layer of the system in their refer-
ence model, we argue that precise detection of WM computations
is challenging due to their natural flexibility and differences across
CPU architectures. In addition we discuss how we found several
surprising ways for 𝜇WMs to improve security. We believe that this
paper introduces a new research area by looking at components
responsible for MA attacks from a different angle and studying
them from the perspective of computation artifacts.

2 BACKGROUND AND MOTIVATION

All current processors can be specified at two distinct abstraction
layers. The first layer is architectural. It is defined by the ISA and
represents architectural state of the machine composed from CPU
registers, instruction pointer and addressable memory. This layer
is visible to programs and programs interact with it directly by
executing instructions and providing data. It is well documented
and can be formally specified [36, 41]. The architectural layer is
realized via microarchitectural features that are not directly acces-
sible to the programmer, including internal CPU components such
as latches, buffers, wires, and various performance optimization
mechanisms. These structures compose the MA layer. Modern day
CPUs incorporate a large number of various performance optimi-
ization mechanisms such as caches, prefetchers, various buffers,
special-purpose computing modules. Many of these mechanisms
have internal data structures with a complex state space.

While the presence of these mechanisms is a well known fact,
little data is available on their internal structure and operation,
apart from the textbook-level description. Moreover, these mech-
anisms are completely transparent to programs executing on the
CPU, only affecting program’s execution time. Yet, programs are
capable of implicitly manipulating MA components by perform-
ing normal activity. This property is used in traditional MA side
channel attacks. Where, for instance, a memory access having an
address dependency on sensitive data triggers a change of state
inside the CPU cache (e.g. transferring its state from not-cached to
cached). Then the attacker can probe the state and infer the secret
data. This basic principal lies in the foundation of 𝜇WM.

2.1 Use Cases for 𝜇WM

𝜇WM described in this paper provide an alternative way of per-
forming general purpose computation on the target platform. They
find a number of use cases (both offensive and defensive) listed
bellow.

Hiding Malware. Malicious functionality in sensitive applications
can be easily obscured by implementing it using 𝜇WM. Malware
can avoid being detected by dynamic or static analysis tools if code sequences used in malware are implemented using μWM. Moreover, doing so provides strong anti-debug protection since MA state is not visible by a regular debugger and is highly volatile. For the same reason, μWM can be used to implement a logic-bomb or trojan application [27] which appears benign but activates its malicious functionality when triggered.

**Preventing reverse engineering.** Obfuscation techniques can be used to prevent reverse engineering applications for protection purposes. For instance, proprietary software developers may want to execute secret algorithm on a third-party untrusted machine without disclosing algorithm’s internals. μWM can be used for this purpose since their reverse engineering requires understanding of complex MA effects which is a difficult task as we demonstrate later in this paper.

**Preventing emulation.** μWM exploit unique features of CPU’s internal components and their interactions such as address conflicts and race conditions. Emulating such effects with an acceptable precision is extremely difficult as it would require first to reverse engineer the target hardware platform. Currently existing cycle accurate simulation only provide an approximate performance model and do not have the level of details required to emulate μWM. We propose to use μWM as an emulation detection/prevention tool where computation can only be performed on a real (not emulated) hardware.

**Violating formal proofs, sandboxes, taint analysis, prevent forensics.** Since currently existing analysis tools do not model MA layer, μWM can be used to perform activity outside of the security model. In addition, since μWM’s current state is not located in regular memory but instead is encoded in the state of MA components, traditional forensics tools cannot be used to study μWMs.

### 2.2 The Problem of Program Obfuscation

We consider a broader understanding of program obfuscation where the goal of the attacker is to perform a malicious computation \( c_m \) within program \( p \) while not being detected by the analyzer. The goal of the analyzer is to decide whether the program \( p \) can perform \( c_m \) under any conditions.

The program \( p \) can be described as a finite state machine (FSM) defined as \( M_p = (S_p, \Sigma_p, \delta_p) \), where \( S_p \) is a finite set of all program’s internal states. Each state is unique and fully determines the current program configuration including the state of all its internal components, such as variables, instruction pointer and others. \( \Sigma_p \) is the program’s input alphabet and \( \delta_p : \Sigma_p \times S_p \rightarrow S_p \) is a transition function. Please note, that this model considers program’s execution at the high level of abstraction and does not specify mechanics of the real platform such as contents of registers or memory cells. Any computation performed by the program \( p \) can be viewed as a sequence of state transitions \( c = (s_{p_0}, s_{p_1}, \ldots, s_{p_{\text{end}}}) \), \( s_{p_i} \in S_p \) caused by starting from the initial state \( s_{p_0} \) and repeatedly applying function \( \delta_p(d_i) \), \( d_i \in \Sigma_p \). In this way, the sequence of inputs \( d = (d_0, d_1, \ldots, d_n) \) determines program’s state transitions. The computation result is determined by the end state \( s_{p_{\text{end}}} \) from a range of possible termination states. The computation then can be viewed as a directed graph by associating each state with a graph node.

Considering this model, the problem of detecting malicious computation \( c_m \) can be thought of as pattern matching problem inside the state transition graph. For instance, a malicious behavior can be detected when program transitions into a single known malicious state \( s_m \), a sequence of malicious states \( s_{m_1} \ldots s_{m_n} \) or a more complex pattern that is known to be malicious. To answer the original question of whether a given program \( p \) is malicious or benign, the analyzer supplies the program with various input sequences, observes program state transitions and detects malicious patterns. In practice this is done by either simulating program’s behavior or using other forms of analysis.

Program obfuscation is the process of transforming the program \( p \) or encoding its inputs \( d \) in such a way to achieve functional equivalency of \( c_m \) through a different computation \( c_m' \) when \( c_m \) is known to be malicious. To illustrate this consider the following example. Let there be a state in which a variable is assigned with a certain value \( x = 10 \). Assume this is considered an indicator of a malicious activity. The attacker then can achieve the same result through a series of two separate actions \( x = 5; x += 5 \). While such a naive evasion technique is easily detectable. A more advanced attacker can utilize more sophisticated program transformation techniques [47] such as replacing registers and instructions or utilizing execution abnormalities such as buffer overflows.

More generally, program obfuscation can be viewed as a graph finding task. In particular, the attacker can use \( M_p \) model to construct a graph \( G_p \) by applying function \( \delta_p(d) \) using all possible \( S_p \) and \( \Sigma_p \). Such graph would contain information about all possible state transitions in \( M_p \). Then the target computation \( c_m \) is a subgraph of graph \( G_p \). To obfuscate it, the attacker searches for another subgraph \( c_m' \) that has a similar topology but does not contain any malicious patterns. In practice this is done with various methods such as changing the program code or using alternative encoding schemes for program input data.

Analyzing a program execution on a real machine, instead of a simplified program model \( M_p \), requires considering the whole machine together with all of its internal components. To do so, one can use an architectural machine reference model \( M_A = (S_A, \Sigma_A, \delta_A) \). Each state in the finite set \( S_A \) determines the current configuration of all machine’s internal components, such as data stored in registers and memory, various pointers and others. Additionally, input alphabet \( \Sigma_A \) must include system-wide events such as interrupts. It is clear that \( M_A \) introduces a more detailed view of the target machine operation. For states in \( S_A \), it is possible to find matching states in \( S_A \). Thus the analyzer can use detection techniques previously discussed in this paper for identifying malicious executables. Please note, although a single state in \( S_A \) maps to multiple states in \( S_A \), it should be trivial to perform the analysis by ignoring irrelevant machine components. A rich state space of \( M_A \) introduces opportunities for program obfuscation. In particular, if certain states or state sequences within \( S_A \) are deemed malicious, the attacker can modify the program or input data in such a way to avoid these states. For instance, the attacker can change registers used by the program, memory locations and instructions. With an accurate model of \( M_A \) and advanced analysis, however, detection of obfuscated program is still possible.
2.3 Program Obfuscation and Microarchitectural Layer

In addition to an architectural layer, real world computers also have a microarchitectural layer. This layer is not considered by conventional malicious software detection tools, yet it has properties that make it desirable for obfuscation. First, it provides a rich state space due to numerous structures implemented at MA layer. Second, MA states are affected by programs executing on the machine, making it programmable by executing regular code. Third, MA layer is usually not well documented, making it very difficult or impossible to create a perfect detection system. These properties make the MA layer a desirable target for program obfuscation. Although MA layer is typically not well documented an advanced attacker can study it using reverse engineering techniques and create a MA model. This model is then used to perform critical parts of computations that would otherwise make malware detectable. Later in the paper we demonstrate how simple elements of MA can be discovered, modeled as FSM and manipulated to create basic computational primitives. Note that it is not necessary for this model to be a complete and full representation of the MA layer. Instead, the attacker can reverse engineer only few components and manipulate them to evade detection.

Speculative execution is a common feature in processors that allows the pipeline to perform computations before the control flow is fully determined. In particular, the pipeline relies on predictions from components such as branch predictors to guess the most likely instruction sequence and executes it immediately. If the prediction later is deemed incorrect, the CPU performs a rollback and continues execution with the correct instruction sequence. However, during such erroneous execution, instructions from the mispredicted instruction sequence are allowed to make changes in MA components. This feature provides a unique functionality for constructing pWMs. It allows to create a divergence between the state transitions in MA and M_p. In particular, to implement a pWM, the malicious executable may intentionally trigger a branch mis-prediction causing some instructions to be erroneously executed in speculative execution mode. Due to the later roll-back these instructions cannot trigger any state changes in M_A while causing state transitions in M_p. As a result, an analyzer with fully visible M_A cannot detect malicious computation if its critical components are implemented via M_p state transitions during an erroneous speculative execution.

3 WEIRD REGISTERS AND GATES

In this section we introduce the concepts of weird registers (WR) and weird gates (WG), basic building blocks for constructing pWMs. The former, as in regular machine is used to store data during machine’s computations and constructed from implicit manipulations with microarchitectural components. The latter represent a minimal functional unit of the machine processing data in its registers.

3.1 Weird Registers (WR)

Any computer can be formalized as an abstract finite state machine $M = (S, \Sigma, \delta)$, where $S$ is a finite set of states, $\Sigma$ is the input alphabet, $\delta : \Sigma \times S \rightarrow S$ is a transition function. Each state $s_i \in S$ represents a unique configuration of all of the machine’s internal components such as memory, registers and storage media. Such model may appear excessive and not practical. However, complex FSM can be simplified if the number of observable components is limited. This effectively creates a new individual FSM with fewer states, input symbols and a simpler transition function. Yet, this FSM contains computational logic embedded in the original machine. For example, a CPU cache has a finite set of states and some logic that controls state transitions which suffice to describe its behavior at a high abstraction level. We refer to such FSMs as sub-FSM or sFSM. We utilize these sFSM to construct simple computational devices that will be used for obfuscation. In particular, they are used to implement data storage entities in the form of WR and a computational primitive in the form of WG. We begin our discussion with explaining construction of WR.

By definition sFSM does not have full information about the machine $M$ but they are useful for analyzing behavior specific to individual subsystems of the machine. Suppose there is a MA resource that we want to utilize as a storage entity and construct a WR $r$. We use CPU data cache as an example. We first select some variable $\text{var}$. Then a simple sFSM $M_r = (S_r, \Sigma_r, \delta_r)$ can be defined for the chosen variable and MA resource. While $M$ contains the full information about the status of the variable $\text{var}$ (e.g. one state for each possible value of $\text{var}$, states representing cache status of $\text{var}$ in L1, L2 and L3 caches, etc.), we define a simpler sFSM with a smaller set of observable states $S_r$. For instance, we consider only two states for the variable represented by its L1 cache status. Such abstraction is useful because it allows to ignore specifics of complex cache organization and treat $r$ as a virtual entity. Let those states be $S_r = \{s_0, s_1\}$. Then $M_r$ is in state $s_0$ when $\text{var}$ is absent from L1 cache and is in state $s_1$ when $\text{var}$ is present in L1 cache. The state transition logic for this sFSM is simple. When $\text{var}$ is accessed $M_r$ transitions to state $s_1$. When $\text{var}$ is flushed from cache via executing the flush instruction the sFSM transitions to state $s_0$. These transitions appear regardless of the current state of sFSM. This establishes the input alphabet $\Sigma_r$. For $M_r$, that is the set of architectural or MA actions within the scope of the subsystem $r$ that can affect MA states. In particular, $\sigma_{s_0} = \text{flush}(\text{var})$ and $\sigma_{s_1} = \text{access}_{\text{mem}}(\text{var})$. Then $\delta_r$ accepts symbols of this alphabet and triggers state transitions as previously described. This allows us to implement weird register, a basic 2-bit microarchitectural storage entity which uses CPU data cache for storage. We refer to this register as DC-WR for data cache weird register.

The state of the DC-WR is read by timing the number of CPU cycles it takes to access the chosen memory location. Please note that reading DC-WR register state is an invasive operation. It causes $M_r$ to transition to state $s_1$. Therefore we introduce an additional signal, $\sigma_r = \text{read}(r)$ for the corresponding sFSM. Processing the read instruction (passing $\sigma_r$) causes the same state transition as $\sigma_{s_0}$ previously defined but has the side-effect of storing the access time in a CPU register. The underlying mechanism of this timed memory load is as in [40]. We define $r$ to have a logic value of 0 when it is in state $s_0$ (not cached) and to have logic value 1 when it is in state $s_1$ (cached). It takes fewer CPU cycles to load var when it is in cache. Therefore we determine the logic value of $r$ by executing the read$(r)$ instruction which has the side-effect of placing that timing in an architecturally visible CPU register. If the load time is
greater than a certain threshold logic 0 is registered, otherwise it is logic 1.

L1 cache state is one of many computer subsystems that has microarchitectural resources that can be explicitly or implicitly manipulated into states that can be made architecturally visible by means similar to the DC-WR. We show some examples of WR that can be constructed using these other subsystems in this Table 1. In addition to utilizing MA sub-systems that have internal storage functionality, such as cache, WR can be implemented through modulating contention on MA resources. Examples of such WR include registers based on mul instructions and ROB listed in the table. Contention-based registers are more volatile and hold data only for certain number of cycles until the contention naturally disappears. Although such volatility deteriorate reliability, it contributes to the stealthiness of μWM.

The concept of WR can also be applied to formally analyze microarchitectural covert and side channels [59]. In the former case two entities construct a communication channel by writing and reading to and from a common WR. In the latter case a sensitive operation of a program belonging to a victim causes a state transition inside sFSM triggering a write to a WR. Then WR is read by an adversary. We believe that any microarchitectural covert or side channel can be abstracted as a WR and therefore can potentially support μWM execution.

In addition to basic data storage capabilities, WR have unique properties.

(1) Volatility: many states of microarchitectural entities are temporal in their nature and exist only for a short period of time. For example one can create a two bit WR from two states of a limited pipeline resource, such as multiplication unit which can be in two contention states, high and low. This register will hold its value for several cycles and then default to the value associated with low contention.

(2) State decoherence: reading WR’s value destroys its value since the reader needs to interact with the MA resource, for example by loading memory and measuring time. Note, that other normal system activity can also interfere with the corresponding MA element and destroy data in the WR. This property makes it challenging for a potential analyzer to observe μWM’s state and apply forensics techniques.

(3) Entanglement: many MA resources are connected to each other entanglement in non-obvious way. For example to assign a value to a data-cache based WR, some code needs to be executed, e.g. a mov instruction. That is, in turn, triggers activity in the instruction cache. Therefore interacting with one WR may affect another WR. While this can be viewed as a noise negatively affecting WR performance, this interference causes unique emergent properties to appear. We use this property later in the paper to construct WC.

(4) Variability: There are many different ways how a sFSM can be constructed from M for the same MA element. For example, one option is to view data cache as having only two states (when variable var is uncached or cached). At the same time, the same MA resource can be utilized to expand the number of states in Sz by using additional information such as L2 status, whether or not a certain cache set is filled or LRU sate [65]. In addition, mappings between Sz states and corresponding WR values is flexible and fully controlled by the attacker. As a result, monitoring μWM activity is challenging even for analyzer capable of observing MA activity.

### 3.2 Weird Gates (WG)

The Weird Gate abstraction builds on that of the WR. WG are basic elements of computation that exploit connection between different MA entities and their corresponding WR. A WG is a code construction that implicitly invokes an activity in MA components in which the state of one or more WR (input WR) conditionally changes the state of one or more WR (output WR) enabling performing computational logic. The WG we discuss in this paper can be viewed as implementation of logic gates such as AND, OR, and NAND. The WG abstraction includes more complex constructs that do not necessarily have 2 level logic output, and indeed we have experimentally verified operation of some such gates, but we choose leave description of such gates to the future work. While we do not describe its construction, among the WGs for which we provide experimental results in Section 6 are NAND gates. This suffices to demonstrate universality of WG as it is known that any arbitrary logic gate may be constructed using NAND gates.

#### 3.2.1 Weird AND Gate

One of the simplest WG we demonstrate in this paper is a gate that implements a logical AND operation which ANDs two input weird registers. In particular, we use registers implemented based on branch predictor and instruction cache as input. The gate’s pseudocode and operation flow-chart are presented in Figure 1. Please note that for demonstration purposes we combine gate code together with input WR assignment operation into a single function. However, in a real μWM these will be performed separately. The operation of the gate is based on the following set of observations. If a program contains a conditional branch instruction (if statement) depending from the previous branch history, the branch predictor can either correctly or incorrectly predict its direction. When the direction is incorrectly predicted, erroneous speculative execution is activated. The attacker can intentionally mistrain the branch in order to enable speculative executing the body of the if statement. However, instructions located inside the if statement’s body will be executed only if they are currently located in instruction cache (IC). This is because speculative execution caused by a branch misprediction has a limited duration known as a speculative window [39] resulting in a race condition. If instructions are not in IC, the speculative window is to narrow
and the execution is terminated before any changes are done to MA components. Note that during the speculative window, instructions from a predicted code branch are not permitted to alter the architectural state of the machine, e.g., the contents of RAM. However, they can change the MA state by, for instance, issuing cache accesses. This principal is used in Spectre attacks [13, 34], where the cache covert channel leaks sensitive data from within the erroneous speculative execution.

The first input weird register for the gate is a WR implemented using the branch predictor state associated with the gate’s if-then statement (line 11). We refer to it as BP-WR. In a carefully constructed code the BP can be trained into one of two states. In state 0 the BP will be trained to not speculatively execute a block of code. In state 1 it will execute the code. In the pseudocode from Figure 1 setting the BP-WR to state 1 is shown as train_bp_nt(). The NT stands for “Not Taken”; training the branch predictor to the “Not Taken” state causes the speculative block to be executed. train_bp_t() sets the BP-WR to 0 since the BP “Taken” state causes the speculative code to not be executed. Our Weird AND gate uses this BP-WR as one of its two inputs.

The second input WR to our AND is an instruction cache WR (IC-WR). The code for the speculative block containing the DC-WR access is either in the instruction cache or that code is not. Due to the limited duration of the speculative window if the code is not in the IC then it will not be executed. We consider the IC-WR to be in state 1 if the code is in cache, and in state 0 if it is not in cache (cflush(if_body) in the pseudocode). When we combine these two input WRs we see that the DC-WR memory access will only occur if both BP-WR is 1 and IC-WR is 1, the BP must attempt to speculatively execute the memory load, and the memory load instruction must be in instruction cache.

The output of our AND gate is in a DC-WR. In our construction the speculative block of code contains a DC-WR memory access operation (some chosen piece of memory is brought into the cache). If the BP-WR is in state 1 then the memory access operation will occur in speculative mode, and the state of the DC-WR will be set to 1 (in cache). We always set the DC-WR to 0 (flush the relevant memory location prior to gate execution. Therefore this combination of WRs perform an assignment operation. If BP-WR value is 1, DC-WR will be assigned to 1.

Note that the operation of this gate is architecturally invisible. While the inputs and outputs are visible, the actual AND logic makes no call to any kind of CPU AND instruction. The part of the WG that modifies the DC-WR state only occurs in speculative mode which has no architecturally visible effects. Note that for the sake of simplicity in Figure 1 the pseudocode contains two functions train_bp_t() and train_bp_nt() which perform direction branch predictor training. In the actual implementation this is achieved by repeatedly executing the branch instruction with the desired direction.

In the experiments discussed in Section 6 we demonstrate that this gate works with a high degree of accuracy.

3.2.2 Weird OR Gate. In this paper we also demonstrate a weird OR gate constructed using the same WRs. Figure 2 demonstrates the pseudocode of a simplified version of this gate. As the AND gate this gate uses the BP-WR and IC-WR registers as input. The gate’s functionality is implemented by placing two if statements into the code. The first statement (line 16) causes the store operation to be speculatively executed when the value of first input register WR-IC is set to 1 (when code at if_body is cached). Note that we always train BP to predict this branch as not-taken. The branch mistraining here is unconditional and serves only to create a window of speculative execution. As a result, in the first if statement the value of the second input register BP-WR is not used. However, its value is used in the second if statement (line 20) while the value of first input register is ignored (we do not flush its code). As a result the gate acts as a logical OR operation. The output WR is not set to 1 only when both of the input WR are 0. Otherwise after the gate activation, the output register is set to 1.

As with the previously explained weird AND gate, the operation of this WG is architecturally invisible. The inputs and outputs are visible, but the OR operation itself uses no CPU OR instruction and the essential parts of the gate operate only in speculative mode.

3.2.3 Other Weird Gates. In addition to aforementioned gates we also composed and studied other logical gates using similar MA mechanisms. The resulting set of gates provides universality property. This enables us to compute results of an arbitrary binary expression by using only two classes of actions, moving data from/to weird registers and activating weird gates. Later in this paper we demonstrate universality of this approach by implementing a SHA-1 algorithm within a μWM. In the experiments discussed in Section 6 we demonstrate that these gates work with a high degree of accuracy.

4 WEIRD CIRCUITS

WGs described in Section 3.2 enable a basic framework for constructing WMs. A computation is first presented as a binary circuit (or expression) and then divided into a sequence of individual register and gate operations. Such model of operation requires outputs of each gate to be read from the output register into the architectural state of the program before it can be sent to the next gate’s input. For the WR implemented using data cache, reading the intermediate state is done by measuring the latency to access the corresponding memory location via the rdtscp instruction. Then the state is written into a WR that is used as input for the next gate. There are several disadvantages to this approach. WR reading and writing operations require a considerable number of additional
instructions executed, causing a slowdown. Moreover, μWM composed in such a way have limited obfuscation properties. Since the intermediate state of μWM is stored in the architectural memory, an advanced analyzer may be able to detect malicious patterns in state transitions of the architecturally-visible program or inside the program’s memory.

Both of these limitations can be addressed by performing contiguous computation within MA state instead of using architectural state to enable the dataflow between weird gates. The goal is to enable contiguous ensembles of WGs implement more complex binary functions than individual gates without saving the intermediate state in architecturally-visible memory. We refer to such ensembles as weird circuits (WC). In WC, data is copied into the MA layer only once and then a series of WG are activated in such a way that the output of one gate serves as input for another gate. The intermediate data is stored only inside WR for the whole time of WC activation.

To describe how WC are formed and operate consider a minimal WC consisting of two and gates transferring the state within a single MA entity appears possible. However, as indicated by prior work [51], the execution is not repeated immediately. The pipeline continues to execute instructions even after the fault. This introduces a new source of erroneous speculative execution which we utilize for WC construction. Please note that MA side-effects from this speculative execution are not rolled-back upon leaving the TSX code. There are many conditions that will cause a TSX transaction to abort such as page faults and divide-by-zero operations. The most common use of the transactions is to avoid using locks. During a transaction the CPU maintains a log of all memory accessed, and if the memory is accessed by another process the CPU aborts the transaction. In some programs this permits optimistic execution of critical sections of computation without need of locks. In our implementations when we want to cause transaction aborts we simply divide a number by 0. This is a stand-in for more subtle (obfuscatable) abort mechanisms that we plan to develop in the future.

The second property cannot be fulfilled due to the use of WR of different types and the lack of hardware interfaces to transfer the state between separate MA entities. First, we consider a case when we need to assign the value of c which is a DC-WR to another weird register b, which is a BP-WR. In this case, we need to conditionally train the BPU depending only on the state of another MA entity, the data cache. Unfortunately there is no simple way to achieve this since it would require performing training of the BPU from within speculative execution. At the same time, transferring the state within a single MA entity appears possible. Suppose we have two DC-WR e and f implemented by variables d0 and d1 correspondingly. By storing the address of d1 in d0 we can implement a basic WR assign functionality (e = f). It is done by simply dereferencing the pointer (*d0) while in speculative execution. Under the race condition, variable d1 will be accessed only if d0 is cached enabling the conditional assign operation.

To overcome these challenges we need to implement a new WG mechanism that does not rely on BPU mistraining and uses WR of the same type for all input and output gates. While alternative implementations are possible, for this paper we will focus on the series WC we have implemented based on Intel Transactional Synchronization Extensions (TSX) technology. Introduced in Haswell microarchitecture, TSX provides CPU-level transactional memory operations. TSX introduces a set of instructions XBEGIN, XEND, and XABORT. When a running program issues the XBEGIN instruction the CPU enters a transactional mode in which operations are executed until either the XEND instruction is encountered or an error condition occurs. When the CPU encounters an XEND instruction with no error then all effects from execution (such as memory reads and writes) are committed and become visible on the architectural level. If an error or fault occurs during the transaction then the executed code is rolled back such that there are no architecturally visible effects from that code and the CPU continues execution at an address specified as an argument to the XBEGIN instruction which is typically a fault handler.

However, as indicated by prior work [51], the execution is not stalled immediately. The pipeline continues to execute instructions even after the fault. This introduces a new source of erroneous speculative execution which we utilize for WG construction. Please note that MA side-effects from this speculative execution are not rolled-back upon leaving the TSX code. There are many conditions that will cause a TSX transaction to abort such as page faults and divide-by-zero operations. The most common use of the transactions is to avoid using locks. During a transaction the CPU maintains a log of all memory accessed, and if the memory is accessed by another process the CPU aborts the transaction. In some programs this permits optimistic execution of critical sections of computation without need of locks. In our implementations when we want to cause transaction aborts we simply divide a number by 0. This is a stand-in for more subtle (obfuscatable) abort mechanisms that we plan to develop in the future.

Figure 2: Pseudocode of an OR WG (a) and its workflow (b)
The TSX mechanism is well suited for construction of WC intended for stealthy operation such as use in our weird obfuscation system. We create a window of speculative execution simply by including a divide-by-zero error in each TSX block. In our experiments we observed that the transaction blocks exhibit a longer and more stable window of speculative execution comparing to when we use BPU mistraining. At the same time multiple TSX based WG can be strung in a row such that they compose a more complex WC that performs calculations in a serial fashion with no architecturally visible intermediate results. They also make it impossible for standard debugging techniques to be used for observing the operation of TSX based WC. A requirement of the transaction interface is that no part of the transaction become architecturally visible unless the entire transaction completes with no other thread accessing memory used in the transaction. If an external debugger were to be able to observe what was happening in a transaction block that would by definition be a side-effect and would cause an abort. The debugger would see the XBEGIN instruction, then the next instruction would be the beginning of the abort handler. There are certain ways that Intel Processor Trace (PT) technology can be used to help debug the insides of transaction blocks, but they require modification of the program which is outside our security assumptions.

Our TSX-based weird gates are based on the observation that the duration of speculative execution occurring inside a TSX code block upon a fault is limited. This crats a race condition. Assume a code sequence consisting of three instructions i1; i2; i3; i4 that are executed inside a TSX transaction block and instruction i1 triggering a fault. In this case, whether or not instructions i2-14 have a chance to execute and alter the MA state depends from their performance. For instance, if instructions do not have any memory dependencies, their execution time is low and they are likely to be executed. Otherwise if they require data from RAM, their execution time is unlikely to fit inside the speculative window provided by the faulty instruction i1. This phenomenon creates a basic primitive needed to construct a gate. Additionally, we can introduce dependencies between instructions by grouping them using arithmetical operations. Assume:

```
i2: mov d0, %r1;
i3: add d1, %r1;
i4: mov (%r1), %r2;
```

In this code sequence, the last instruction will be able to issue a store request and modify the MA state only if variables d0 and d1 are both cached. This effectively creates an AND weird gate with input and output registers being DC-WR.

Figure 4 contains pseudocode for a sample TSX WC that we use for testing and which simultaneously calculates $Q_0 \leftarrow A \land B$ and $Q_1 \leftarrow A \lor B$ in a single WC based on the principle that cache status of operands to addition will determine whether the addition will be performed. This WC is a component of the weird obfuscation system implementation described in Section 5.1. In this pseudocode d0...d4 are variables that implement DC-WR. Absence from cache representing logic 0 and presence in cache is logic 1. Line 3 initializes all the DC-WR to logic 0 by flushing the memory to which they point. In lines 4 and 5 the architecturally visible inputs A and B are read into d0 and d1. Line 8 is the first line of the actual WC. It is an illegal operation that aborts the transaction rendering the operation of the gate architecturally invisible. After the abort speculative execution continues to the transaction end. The Weird Gate in this the weird circuit consists of the assignment weird gate in which DC-WR d3 will be set to logic 1 if DC-WR d0 is logic 1. It operates as follows: On line 9 if DC-WR d0 is logic 1 then *d0 will be in cache and the addition will be evaluated. *d0 holds value 0, and ADDR(d3) is the constant value of the address pointed to by d3 which is determined at compile time. Therefore the result of the addition is (0 + ADDR(d3)) and when the result of the addition is dereferenced *d3 will be loaded into memory setting the logic value of DC-WR d3 to 1. If *d0 is not in cache then the addition will not be performed, *d3 will not be loaded into memory, and the DC-WR value of d3 will remain logic 0. The second WG on line 10 is similar to an assignment WG. It differs only in that the destination WR for the assignment (d2) is not initialized to logic 0 prior to the assignment. Therefore if d1 is logic 1 d2 will be set to logic 1, but if d1 is 0 the value of d2 will remain unchanged. The combination of those two WG result in a TSX-based weird OR circuit. The third WG in the WC is a TSX based weird AND gate. The addition expression on line 11 will only be evaluated if both d0 and d1 are set to logic 1 which forms an AND using the same principles used to build the OR. After the WC has executed we read the WR values into visible memory. We want to avoid having the rdtscp instructions visible to an observer because they are frequently associated with Spectre style attacks. We therefore perform the timed memory load inside a TSX transaction. An adversary attempting to observe the process of reading the WR will cause that transaction to abort which destroys the value of the WRs and leaves the architecturally visible outputs Q0 and Q1 set to 0. Intentionally causing such aborts to disrupt malware weird circuits hidden in a legitimate program is an interesting line of future work. It will, however, interfere with proper execution of the legitimate program if done in a naïve way.

**Figure 3: Pseudocode for TSX weird circuit that computes**

\[Q_0 \leftarrow A \land B, Q_1 \leftarrow A \lor B\]

```
<table>
<thead>
<tr>
<th>Weird Gate</th>
<th>Iterations</th>
<th>Execution Time (s)</th>
<th>Executions / Second</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>81</td>
<td>0.42</td>
<td>1M</td>
<td>98%</td>
</tr>
<tr>
<td>OR</td>
<td>766</td>
<td>0.591</td>
<td>1M</td>
<td>98.5%</td>
</tr>
<tr>
<td>NAND</td>
<td>766</td>
<td>0.591</td>
<td>1M</td>
<td>98.5%</td>
</tr>
<tr>
<td>NOT</td>
<td>766</td>
<td>0.591</td>
<td>1M</td>
<td>98.5%</td>
</tr>
<tr>
<td>TSX_AND</td>
<td>100%</td>
<td>12,345</td>
<td>1M</td>
<td>98%</td>
</tr>
<tr>
<td>TSX_OR</td>
<td>100%</td>
<td>15</td>
<td>1M</td>
<td>98%</td>
</tr>
<tr>
<td>TSX_XOR</td>
<td>100%</td>
<td>15</td>
<td>1M</td>
<td>98%</td>
</tr>
<tr>
<td>TSX.Assign</td>
<td>100%</td>
<td>15</td>
<td>1M</td>
<td>98%</td>
</tr>
<tr>
<td>TSX_ADDON</td>
<td>100%</td>
<td>15</td>
<td>1M</td>
<td>98%</td>
</tr>
<tr>
<td>TSX_XOR</td>
<td>100%</td>
<td>15</td>
<td>1M</td>
<td>98%</td>
</tr>
</tbody>
</table>

### 4.1 TSX-Based Weird XOR

The security of the weird obfuscation scheme discussed in Section 5.1 derives from the properties of a one-time pad (OTP). As with all OTP schemes each bit of the ciphertext must be XORd with each bit of the pad to recover each bit of the plaintext. As discussed in the previous subsection multiple TSX based WC can be strung in a row such that they perform more complex calculations in a serial fashion with no architecturally visible intermediate results. The TSX_XOR, our TSX based implementation of XOR that is used by our weird obfuscation scheme, demonstrates a multi-step calculation with no visible intermediates. It performs XOR using the TSX_AND.OR gate from the previous subsection together with a TSX-based NOT WG and an additional TSX-based AND gate.

Evaluation of TSX-XOR accuracy is discussed in Section 6.4.

### 4.2 Gate Performance Estimate

Naturally occurring system noise caused by timing variations and various MA conflicts expectedly causes a reduction in WG stability. Reliability can be improved by introducing redundancy by repeatedly activating the same gate and using a voting mechanism to choose the gate’s output. Such redundancy affects overall performance of $\mu$VM. To estimate performance of WGs we carried out an experiment in which we executed various non-TSX and TSX based gates in large series and measured the overall performance and accuracy. Results are demonstrated in Table 2. Please note that in this work we have not made significant attempts to optimize the gates for performance and shown results represent data from rather naive implementation. We believe the performance can be significantly improved by carefully optimizing WG code, deeply studying the MA effects and designing better error detection or correction mechanisms. We leave this research direction for the future work.

### 5 APPLICATIONS OF WEIRD CIRCUITS

In previous sections we explored design and implementation of simple weird circuits that demonstrate the ability to create functionally complete microarchitecturally invisible boolean circuits. In this section we will first demonstrate weird obfuscation, a malware obfuscation system that uses more complex WC. We will then examine in greater depth the multi-gate TSX-based weird XOR circuit used by the weird obfuscation system. Finally we will demonstrate an implementation of SHA-1 that uses weird circuits.

#### 5.1 Weird Obfuscation System

In this section we describe the operation of our weird obfuscation (WO) system and how we use this system to obfuscate malware code such that its passive operation is invisible to an observer with full power to view the architectural state of a system until the code receives a ping with a special trigger value in the body. When the trigger is received the malware decrypts and executes its payload. We will demonstrate the use of our WO system to conceal and then execute a payload that exfiltrates a shadow password file, and a payload that creates a reverse shell.

In our scenario we are in the roll of an attacker who has managed to get an advance persistent threat (APT), such as a trojan horse, installed onto a computer running inside an adversary’s network. Our adversary, the defender, has the ability to view all architectural state of the infected computer. Our adversary has the power to run our infected program in a debugger or other dynamic analysis tools. We hope this work will inspire future work for development of static analysis tools that will detect and characterize $\mu$VM in programs such as our APT, but as discussed in Sections 1 and 7 those tools do not yet exist. We therefore do not give our adversary abilities granted by those tools.

When constructing our APT we first take the payload, choose a random 128 bit AES key, encrypt the payload with that key and store the encrypted payload in the structure shown in Figure 4 a) starting at bit 324. We then place a specially crafted jmp instruction at bit 164 followed by the AES key. We then create a random one-time-pad of length 160 bits. We then XOR each bit of the pad against the bits of the memory structure starting at bit 164. This has the effect of “encrypting” the jmp instruction and the AES key against the one-time-pad. The 160 bit one-time-pad will later be used as the trigger value that will cause our malware to enter its active phase. We complete the preparation by filling the first 160 bits of the structure with random data followed by an illegal divide by zero instruction, then copying the entire structure into the body of a TSX block.

Our APT is malware hiding in a program that receives pings. During operation every time it receives a ping each ping body payload is used as an XOR key to transform the memory labeled...
xor-encrypted JMP/AES key in Figure 4 and overwrite the bits labeled random. Bits 32-160 are then used as an AES key to decrypt the payload at the end of the memory region. Finally, the entire region is mmap’d and executed inside a TSX block as shown in Figure 5.1. If the secret key in the ping body was correct there a jmp instruction leading to goto target_function appears at bits 0-32 that will begin execution of the decrypted payload. Because our XOR and AES keys were correct in this case, the payload will execute properly and open a reverse shell to the attacker.

During the silent phase, before the attack is triggered, the affected machine may receive many pings. When a received ping does not contain the trigger value, first 160 bits of the TSX block will contain a bad AES key resulted in garbage values in the decrypted payload, and no jmp instruction. Instead of properly jumping over the contents of the AES key and divide-by-zero instruction, this incorrectly-decrypted region is executed as-is. This will generally cause a near-immediate fault, but is guaranteed to fault by the time it reaches the tmp = tmp/0 instruction at bits 160-164. This fault is then rolled-back since it is inside a TSX block, and the program continues to wait for the next ping trigger.

All critical parts of this APT operate in TSX blocks which are not directly observable by a debugger. In addition, the one-time-pad “decryption” of the AES key is performed by a TSX-based XOR WG that has no architecturally visible intermediate values. The analyzer will not see any part of the payload until the trigger has been successful and the payload is already running.

Execution of the logic gates underlying the TSX-based XOR, as discussed in Section 4.1, is not 100% accurate and is not guaranteed to always return a correct result. Practically this means that each trigger must be evaluated by the APT multiple times. We chose this evaluation multiple to be 10. In our implementation the APT is able to process pings in real-time with inter-arrival times up to 500ms. We see in Figure 6 and Table 3 the distribution of pings in 100 experiments required to successfully decrypt and execute the reverse shell malware payload. A median of 6 pings demonstrates successful XOR transform of the 160 bits shown in Figure 4 in ~60 XOR attempts. Detailed performance of this operation is examined in Section 6.

### 5.2 SHA-1 Implementation

We chose a hashing algorithm to be an illustrative high level algorithm with which to demonstrate partially architecturally visible pWM for a number of reasons. A cryptographic hashing function provides a challenging case for pWM which have components with less than 100% accuracy. If a single bit of a pre-image is changed, a good cryptographic hash function should provide a completely different hash. In the same way single bit errors that occur during hash computation are magnified which makes SHA-1 a challenging test case for what can be done with pWM. While SHA-1 is no longer considered a secure hash function, it is still an algorithm at a level of complexity suitable for initial demonstration of of pWM fitness for complex computational tasks. Another reason we chose a hashing algorithm is that our WC version can be used to replace the hash function in the malware obfuscation system due to Sharif et al. [56]. This provides an additional malware obfuscation system based on pWM.

Referring to our SHA-1 implementation “partially architecturally visible” implies that while many interim values are stored in architecturally visible memory all of the actual SHA-1 computation is performed by pWM. For example, when the algorithm requires adding two numbers, no CPU add instructions are executed. The implementation performs the addition using a full adder constructed from two discrete weird XOR gates and a composed weird AND_AND_OR gate. During execution the output of the weird XOR gates is temporarily stored in memory as is the output of the weird AND_AND_OR gate. With the parameters that we chose for experiments as described in Section 6.5.2, 41.9% of the intermediate results were architecturally visible.

As discussed in previous sections many of our weird gates have a high degree of accuracy, but in very long runs errors do occur. Our implementation of SHA-1 requires more than 100 000 executions of several different kinds of WG and we determined that the accuracy of the WG was insufficient. Our SHA-1 implementation improves on the accuracy of single gate executions by performing multiple redundant executions. In the first step it performs each WG execution $s$ times, storing the timing values in each element of an array $T$ of length $s$. The algorithm then converts the median value in $T$ into a logic value which is then stored in array $L$ of length $n$. The algorithm then uses a best-k-of-n voting scheme such that whatever logic value appears more than $k$ times in array $L$ is chosen as the final output. Parameters $s$, $k$, and $n$ are specified in a configuration file.

The described system supports a chosen tradeoff between execution time, visibility (number of intermediate values exposed in RAM), and accuracy. We ran a series of 10 experiments in which each experiment consisted of an execution of the SHA-1 implementation. For these experiments we chose parameters $s=18$, $k=3$, and $n=5$ which are conservative values favoring accuracy over speed. Each of those experiments produced a correct hash. Table 4 shows

**Table 3: Triggers required for successful wm_apt transform**

<table>
<thead>
<tr>
<th>Triggers</th>
<th>Min</th>
<th>Q1</th>
<th>Med</th>
<th>Q3</th>
<th>Max</th>
<th>Std Dev</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>12</td>
<td>60</td>
<td>11.19</td>
</tr>
</tbody>
</table>

![Figure 6: Histogram of wm_apt triggers yielding successful transform](image-url)
the correct / incorrect gate executions from a representative experiment in which “Correct After Median” indicates that the median timing value from $T$ was in the correct range for the expected output of the WG, and “Correct After Vote” indicates that the result of the $k$-of-$n$ vote was the correct logic value. Each execution took around 26 minutes. The experiment shown in Table 4 took 26:30. Please note that we have made no attempt to find parameter values that produce near 100% successful SHA-1 hashes at faster speeds.

## 6 EXPERIMENTAL METHODOLOGY AND EVALUATION

### 6.1 Setup for Weird Gate and Weird Circuit Experiments

We used 2 different computers for the experiments discussed in this paper. We performed all experiments that involved TSX-based WC on a laptop fitted with an Intel Dual-Core i7-6600U CPU running Ubuntu 18.04.4 LTS with a 5.4.0-42 kernel. We configured grub such that physical CPU 1 was isolated and the dynamic frequency scaling was disabled to permit us to manually set the CPU frequency to 2.30 GHz for all experiments. We configured our experiments to always use the 3rd logical CPU core for $\mu$WM execution. We performed our non-TSX based WC experiments on a machine equipped with i5-8259U 2.3GHz CPU. The setup was otherwise the same as for the TSX-based WG experiments.

While this configuration improves the overall $\mu$WM stability, it is not critical and practical $\mu$WM computation can be achieved on a system with the default configuration. CPU frequency scaling and other processes being executed on the hypercore within the same CPU core introduce the most invasive effects. However, it is fairly easy to reduce such effects by executing an infinite loop and tying it to the matching hypercore. This 1) brings CPU frequency to the maximum supported value, 2) does not allow other processes to be scheduled on that core. In addition, WR demonstrated in the paper are implemented using only local core resources. As a result, activity on other cores does not have a drastic effect.

### 6.2 Evaluation Framework

One of the challenges of working with WGs is their stability when composed into WCs. Successful execution of each WG depends on microarchitectural state that is not generally obvious to a developer. To combat this, we developed a framework we call skelly that abstracts away the need to understand the state of the microarchitecture to build WCs. This framework acts as a static library that provides basic logic functions such as int and(int a, int b) ; . Our SHA-1 implementation (see Section 5.2) was built using the skelly framework.

Aside from reliability improvement features such as the $k$-of-$n$ vote described in the SHA-1 implementation, the main feature of skelly is its mechanisms to maintain WG functionality even as WGs are added. For example, we found that branch predictor and instruction cache WGs have a strong dependence on code alignment such that the clflush operates on the correct cache line. We identify and map in skelly a dedicated portion of memory at cache-aligned addresses for each WG that depends on this alignment and initialize it at run time. This has allowed us to implement AND, OR, XOR, NOT, AND-AND-OR, and a variety of convenience functions such as a full adder, 32-bit left shift/rotate, as well as 32-bit versions of all logical primitives.
6.3 Evaluation of Branch Predictor and Instruction Cache Based Weird Gates

We evaluated our branch predictor/ instruction cache based gates, such as AND and OR, using the skelly framework with system conditions outlined in Section 6.1. skelly was compiled with a flag allowing architecturally visible verification of WG outputs to report on purposes. In evaluating AND & OR we performed 320,000 operations per gate type using sets of 2 randomized input provided by rand(). We show the accuracy statistics in Table 5, and timing KDEs for each in Figure 7 and Figure 8 that show the logic level boundary.

6.4 Evaluation of TSX-Based Weird Circuits

We evaluated our TSX-based gates using an optimized version of the skelly framework with additional code alignment to improve TSX gate stability. Each gate was exercised with 64,000 operations. We show in Table 8 the mean accuracy and number of unrecovered TSX aborts across 4 TSX-series gate types and present delay measurements from TSX–AND–OR and TSX–OR gates in Tables 6 and 7.

6.5 Setup for Weird Circuit Application Experiments

6.5.1 Weird Obfuscation. The wm_apt framework as described in Section 5.1 was built with the modified skelly framework described in Section 6.4. We performed the following experiment 100 times to generate the distribution shown in Figure 6. In the first terminal wm_apt is executed. In a second, continuous pings are sent with ping localhost -p $XOR_SECRET -i 0.5. In a third, a netcat session is running awaiting the reverse shell from the wm_apt with the following commands: kill ping && exit. Upon successfully decoding the malicious payload, the reverse shell terminates ping and the ping count is recorded.

6.5.2 Evaluation of the SHA-1 Implementation. The $\mu$WM computing SHA-1 was built using the interfaces provided by the skelly framework. For evaluation a test fixed plaintext string was provided to the executable 10 times for hashing using system conditions outlined in Section 6.1. In our test run each of the ten hashes executed successfully, with discrete and logical operation counts and accuracies outlined in Table 4. As a convenience we have allowed skelly framework to abort when an incorrect logical operation is detected, and to compare the hash output to a reference SHA-1 implementation. During standard operation, however, skelly does not provide this verification.

7 RELATED WORK

The work presented in this paper touches on three areas of research: Weird Machines, microarchitectural side-effect based computing and software obfuscation. We discuss related work in all of these.

Dullien [20] was first to provide a formal model for the notion of a Weird Machine. Previously several works demonstrated possibility of WM construction using architectural-level artifacts in existing machines. For instance, Shapiro et al [55] described an ELF (Executable and Linking Format) Weird Machine that is present within the Linux runtime loader (RTLD). The authors showed how computations defined using a formal language can be used to drive its operation. Bangert et al. [4] described a Weird Machine present within the IA32 architecture’s interrupt handling and memory translation tables and how it can be used to perform arbitrary computations. These works show the use of unexpected (but specified) computation capabilities within the architectural layer of the target CPU. Both WMs are observable and therefore can be mitigated by an analyzer constructed following CPU specifications.

Szepes et al. [58] provides a detailed survey of microarchitectural side and covert channel attacks and outlines the space for MA artifacts that can be used to construct $\mu$WM. Kocher et al. [34] showed how branch prediction state can be manipulated to force victim programs to leak arbitrary data via a cache based covert channel. Lipp et al. [38] leveraged delayed exception handling within the CPU pipeline to leak kernel space data into user space. More recently line fill buffers [60] and write transient forwarding of store buffer values [42] were exploited to leak data without address space restrictions. The existing corpora of work on attacks using microarchitectural side-effects typically focuses on data leaks using various microarchitectural structures rather than attempt to formalize the computation. Wampler et al. [61] demonstrates the feasibility of concealing computations using speculative execution. The computations are performed by executing instruction sequences that cannot be executed during normal operation of the program but can occur in transient execution mode due to the CPU’s mis-speculations. Traditional reference monitors cannot detect this attack since they ignore never-executed code paths. However recently developed speculative execution code trace detection tools [28, 62] can be used to detect such types of execution. In contrast, in our work the malicious code is not represented by regular ISA instructions and to compare the hash output to a reference SHA-1 implementation.
in irregular behavior similar to that observed during side channel attacks, existing tools may be applied to detect \(\mu\)WM presence. Such tools typically use hardware-based performance monitoring to detect abnormal patterns in software. However, they usually are only used for monitoring applications handling sensitive data and not designed for full-system monitoring [43, 69] which is needed for \(\mu\)WM detection in a running system. While [43] can detect malicious speculative execution patterns, the analyzer needs to be trained for specific types of malicious code sequences. In this paper we discussed how \(\mu\)WM phenomenon is not tied to a specific microarchitectural entity. An advanced attacker may be able to bypass the detection by discovering and using alternative mechanisms inside CPU microarchitecture. This makes construction of an universal \(\mu\)WM detector challenging or even impossible. Several recent works [33, 48] proposed to mitigate speculative execution attacks by limiting speculation or by hiding effects from speculative execution. Such protections require significantly changing the CPU pipeline.

Recently microarchitectural protections against speculative executed attacks were introduced that relay on CPU microcode to suppress dangerous speculative execution. While some of our gates that rely on branch prediction mistraining may be affected, this happens only under the most conservative configuration. Such configuration is likely to cause drastic performance degradation. Moreover, gates implemented using the TSX technology appear unaffected.

The approach we use to obfuscate the logic bomb used to trigger the APT described in this paper is based loosely on the approach followed in [56]. Here the logic bomb is hidden by use of a cryptographically secure hash function to hide the value of the input trigger. Such an approach is susceptible to brute force attacks since the code used to decode the input trigger can be executed in arbitrary environments consisting of CPUs with diverse microarchitectural characteristics. Our use of microarchitectural side-effects to decode the input trigger makes such attacks harder to accomplish since the decoding will work only in specific microarchitectural environments. Baldoni et al. [3] discusses the use of symbolic execution techniques to identify backdoor inputs. Symbolic execution operates at the level of programmatic abstraction which is even higher than the architectural layer which in turn is above the microarchitectural layer. For true brute force analysis multiple instances of virtualized environments need to be created to explore multiple paths in a program; this leads to more impacts since the code used to decode the input trigger can be executed in arbitrary environments consisting of CPUs with diverse microarchitectural characteristics. Bulazel et al. [10] discusses dynamic monitoring of malware execution using various types of instrumentation, generally either in-system or out-of-system. Most monitoring techniques result in timing overhead except possibly bare-metal analysis. Bare-metal analysis is used to counter virtualization-resistant malware but provides less insight into malware behavior and results in challenges in scalability. Fratantonio et al. [27] discusses the use of static analysis to detect suspicious predicates that guard sensitive functions; this approach cannot apply to WCs. Schrittwienser et al. [50] deals with binary code analysis. It classifies analysis goals, obfuscation techniques, analysis techniques and then assesses the security of code obfuscation techniques. They discuss dynamic analysis which trades off coverage against cost, microarchitectural behavior exhibited only under very specific circumstances may never be observable owing to scaling requirements for coverage.

8 CONCLUSIONS

We have introduced the concept of \(\mu\)WMs: a methodology for harnessing the computing capability provided via the unspecified aspects of CPU microarchitectures. We have described a framework for programmatically storing and operating on microarchitectural state as WRs and WGs respectively. We have shown techniques for composing primitive operations into more complex WCs. Our approach for manipulation of microarchitectural state has been shown to be applicable to diverse microarchitectural structures. The use of our approach for the creation of an obfuscated, microarchitectural sensitive logic bomb as well as for the implementation of a reasonably complex cryptographic algorithm SHA-1 shows its flexibility for diverse applications. Obfuscation of code functionality, in particular, has been shown to be an important area of application for \(\mu\)WMs. We believe that our work merely uncovers the tip of the iceberg: we believe that \(\mu\)WMs will have strong applications in both offensive and defensive adversarial scenarios in the future.

REFERENCES


