MARTINI: Memory Access Traces to Detect Attacks

Yujun Qin
University of Michigan
qinyujun@umich.edu

Samuel Gonzalez
University of Michigan
samgonza@umich.edu

Kevin Angstadt
St. Lawrence University
kangstadt@stlawu.edu

Xiaowei Wang
University of Michigan
xiaoweiw@umich.edu

Stephanie Forrest
Arizona State University
stephanie.forrest@asu.edu

Reetuparna Das
University of Michigan
reetudas@umich.edu

Kevin Leach
University of Michigan
kjleach@umich.edu

Westley Weimer
University of Michigan
weimerw@umich.edu

ABSTRACT

Hardware architectural vulnerabilities, such as Spectre and Meltdown, are difficult or inefficient to mitigate in software. Although revised hardware designs may address some architectural vulnerabilities going forward, most current remedies increase execution time significantly. Techniques are needed to rapidly and efficiently detect these and other emerging threats.

We present an anomaly detector, MARTINI, that analyzes traces of memory accesses in real time to detect attacks. Our experimental evaluation shows that anomalies in these traces are strongly correlated with unauthorized program execution, including architectural side-channel attacks of multiple types. MARTINI consists of a finite automaton that models normal program behavior in terms of memory addresses that are read from, and written to, at runtime. The model uses a compact representation of *n*-grams, i.e., short sequences of memory accesses, which can be stored and processed efficiently. Once the system is trained on authorized behavior, it rapidly detects a variety of low-level anomalous behaviors and attacks not otherwise easily discernible at the software level.

MARTINI's implementation leverages recent advances in in-cache and in-memory automata for computation, and we present a hardware unit that repurposes a small portion of a last-level cache slice to monitor memory addresses. Our detector directly inspects the addresses of memory accesses, using the pre-constructed automaton to identify anomalies with high accuracy, negligible runtime overhead, and trivial increase in CPU chip area. We present analyses of expected hardware properties based on indicative cache and memory hierarchy simulations and empirical evaluations.

CCS CONCEPTS

• **Computer systems organization → Special purpose systems**; • **Security and privacy → Intrusion detection systems**.

KEYWORDS

intrusion detection, side-channel attacks, automata processing

ACM Reference Format:
Yujun Qin, Samuel Gonzalez, Kevin Angstadt, Xiaowei Wang, Stephanie Forrest, Reetuparna Das, Kevin Leach, and Westley Weimer. 2020. MARTINI: Memory Access Traces to Detect Attacks. In *2020 Cloud Computing Security Workshop (CCSW'20), November 9, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3411495.3421353

1 INTRODUCTION

Two trends point to the need for robust, low-overhead detection of novel attacks: (1) the advent of attacks that exploit architectural vulnerabilities, such as Spectre [24] or Meltdown [28], and (2) the widespread use of cloud and embedded systems intended to run sets of authorized programs but vulnerable to the injection of unauthorized code [10, 11, 26]. These problems, especially architectural vulnerabilities, are not easily and efficiently mitigated with software patches. Thus, there is a need for solutions that can be deployed with minimal modification to existing hardware, that impose minimal overhead on running software, and that generalize to detect novel attacks.

Runtime intrusion detection systems (IDS) are a well-studied approach for software vulnerabilities [27, 32, 42] and are deployed in industry [22, 53]. These systems typically monitor systems at high levels of abstraction (e.g., using system calls [18], network packets [43], filesystems [22], user-behaviors [36], architectural features [40, 63], or hardware performance counters [47]). Unfortunately, modern attacks such as Meltdown and Spectre leave no trace at these levels (and indeed are hidden by the processor's ISA) [27]. Other vulnerabilities may not be exposed with existing IDS techniques. Further, many host-based IDS approaches have high overhead arising from instrumentation (such as hooking I/O system calls or inspecting network packets). Thus, existing, software-level IDS are insufficient for defending against emerging classes of attacks. For Spectre and Meltdown, CPU speculation features can also be disabled, but the performance impact is high [31, 58] and doing so does not address other extant hardware vulnerabilities. Finally, novel microarchitecture redesigns are non-trivial and costly in terms of time and resources and may expose new—or overlook existing—hardware vulnerabilities.

CCSW'20, November 9, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8084-3/20/11…$15.00
https://doi.org/10.1145/3411495.3421353

There have been recent advances in processing-in-memory technology and, specifically, the development of memory-centric finite automata accelerator architectures such as the Cache Automaton [51], which admit low-overhead access to memory and require minimal architecural modification. Further, we observe that all programs manipulate memory and do so in characteristic ways (i.e., a program may be identified by its instruction stream). We hypothesize that (1) memory access patterns provide a *suitable abstraction* for identifying programs and detecting anomalous behavior, and (2) it is feasible to construct low-overhead anomaly detectors leveraging this memory access abstraction and in-memory computation.

In this work, we present initial evidence in support of these hypotheses. We describe Martini,[1] a low-overhead, hardware-assisted anomaly-intrusion detection system that detects anomalous and malicious program execution at the memory access level, including emerging hardware attacks. Martini can be deployed alongside existing IDS techniques to provide a greater level of system security. With Martini, authorized behavior is modeled with *dictionaries* that represent *n*-grams, or sliding windows, of short sequences of memory accesses, where each memory access is compressed into eight bits of information. Because Martini uses *n*-grams rather than complex pattern matching, once the dictionary is trained on indicative, authorized behavior, subsequent queries can be formulated in terms of finite automata inputs. Thus, Martini can be deployed in hardware with low overhead and latency by leveraging near-memory processing and in-cache computation. We develop a new functional unit with a custom data path that can be deployed in the processor core or last-level cache of modern CPUs, which admits real-time monitoring of memory accesses.

As an initial deployment scenario, we consider high-assurance whitelisting in which an operator provides a list of known, authorized programs—any other program or behavior is unauthorized (anomalous), and any such anomalous behavior should raise an alarm that terminates the program or logs the activity for later scrutiny by an operator. Such whitelisting facilities are useful in scenarios that require regulatory compliance, such as avionics software [44], health informatics [54], cloud infrastructure [16], and industrial control systems [38].

Threat Model: We assume an attacker can hijack control of userspace software by exploiting a vulnerability in an authorized program (i.e., executing an authorized program using malicious inputs) or by executing unauthorized programs. Martini is trained on indicative, authorized benign behavior, then exposed to subsequent behavior and asked to decide whether or not it is anomalous. Benign programs are known in advance (i.e., authorized), but their inputs are not. Generality thus requires few false alarms on untrained benign inputs and accurate detection of unauthorized scenarios, i.e., one that has high true positives and low false positives.

Our evaluation of Martini demonstrates an overall false positive rate of 4.4% with a true positive rate of 100% (area-under-curve = 0.9954) across a benchmark consisting of 16 Coreutils programs, the PARSEC 3 benchmark suite, and four recent CVEs that include Spectre and Meltdown proofs-of-concept. We consider more than 2,400 program traces and more than 13 billion individual memory accesses. We further evaluate the cost of our approach with a circuit simulation of the chip area required to deploy Martini. Based on a simulation using a 45nm process, we estimate that the area of our proposed address monitoring unit is only .015mm^2, a die area increase of less than 0.04% for a current-generation 8-core Xeon processor. These low overheads allow Martini to be deployed alongside existing IDS techniques providing defense in depth. Our empirical evaluation provides evidence in support of deploying processing-in-memory for security applications.

In summary, the primary contributions of this paper are:

- A model of system-level behavior based on sequences of memory accesses. The model captures microarchitectural phenomena, such as those exploited by recent hardware attacks, that are unobservable in current software-level IDS.
- Martini, an implementation that leverages this model to detect unauthorized program behavior, including architectural side-channel attacks, using dictionaries of *n*-grams of memory accesses.
- An empirical evaluation of Martini's classification accuracy on over 2,400 program traces from two large benchmark suites and four exploits, including Spectre and Meltdown proofs-of-concept. We find that Martini is able to classify intrusive activity with high accuracy and precision (AUC 0.9954) while requiring a very small chip area. Moreover, deploying Martini in hardware would enable classification without runtime overhead.
- A custom data path that leverages recent insights from high-performance automata processing to provide per-cycle monitoring of memory accesses. Our design increases total chip area by less than 0.04% on an 8-Core Xeon processor and demonstrates that low-overhead, hardware-level deployment of anomaly detectors is feasible with current design and manufacturing techniques.

2 BACKGROUND AND RELATED WORK

In this section, we describe (1) the memory access patterns of architectural side-channel attacks, (2) earlier work on intrusion detection and process identification, and (3) hardware acceleration via near-memory computation.

2.1 Architectural Side-Channel Attack Memory Access Patterns

Architectural side-channel attacks, such as Spectre and Meltdown, use cache timing to leak information in memory. Such attacks can exploit side effects of branch prediction and speculative execution to read or affect arbitrary memory locations. The key problem is that changes to the state of the cache persist even if the CPU discards instructions that are speculatively executed. Consequently, a malicious program can influence that state by executing a controlled sequence of memory accesses, then leverage its knowledge of the cache structure to either read or write arbitrary locations in memory that are cached by other programs. In addition to Spectre and Meltdown, other cache side-channel attacks include Foreshadow [55], Flush+Reload [64], Evict+Time [41], Prime+Probe [29], and Nailgun [37]. Since these attacks rely on hardware vulnerabilities, they are OS-independent and challenging to patch efficiently in software.

[1] Martini = Memory Address Representation To INfer Intrusions.

We consider the Meltdown vulnerability [28] as an indicative example and elaborate its memory access patterns briefly to motivate Martini's design decisions

At the core of Meltdown is an exploit to read out the value v stored at a given address A via a side-channel. First, Meltdown (illegally, but speculatively) reads from A the value v. Then, Meltdown (speculatively) reads from a legal address $f(v)$ which depends on that previously read value. However, these steps alone are inadequate to determine the exact value v: because the access to A is illegal, the speculation is carried out but not committed. In a second loop, each potentially accessed address (e.g., $f(0)$–$f(255)$) is legally loaded and timed before being flushed from the cache, which reveals which load was speculated and placed in the cache. Since the loaded address $f(v)$ was based on the (speculatively) loaded value v, this leaks v, the value at A. This exploit has a regular memory access pattern for every iteration and motivates our choice of n-grams of memory addresses to identify malicious behavior.

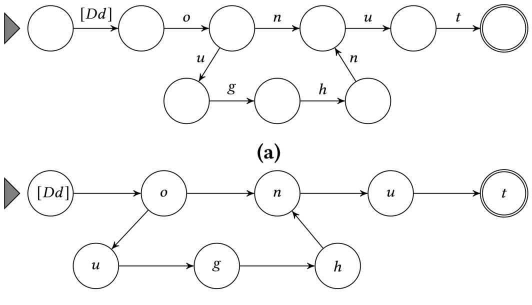
In addition, although the absolute page locations of addresses may vary from run to run (e.g., depending on the target address A as well as the location of Meltdown's region $f(0)$–$f(255)$), the relative differences and low-order bits are likely preserved across runs. As a result, we propose to abstract memory addresses: By using fewer bits to represent each access we can achieve low-overhead deployment, generality, and sensitivity to unauthorized behavior.

Memory access patterns have been studied in microarchitectural domains, e.g. for prefetching [33], but there is limited work that uses memory access patterns for detecting attacks. Moreira *et al.* [35] used Markov chains to represent program behavior from memory accesses, but their technique relies on expensive machine learning methods and are not suitable for our use case because they cannot be deployed in-hardware with low runtime overhead. Additionally, they focus on fault tolerance rather than malicious activity.

2.2 Runtime Intrusion Detection Systems

Broadly, an IDS is tasked with classifying a sequence of inputs as being normal or anomalous according to some model. IDS approaches can be categorized along several dimensions. Lazarevic *et al.* [25] focus on the network-based or host-based information analyzed. IDS can also be categorized according to the method used to specify normal behavior, e.g., statistics-based, pattern-based, rule-based, state-based or heuristics-based [27]. Yet another dimension focuses on signature-based vs. anomaly-based detection, where signature-based methods represent patterns of of known attacks and anomaly-based methods learn a model that can discriminate benign (normal) behavior from abnormal behavior that is correlated with malicious behavior. We selected an anomaly-detection approach, rather than a signature-based one, because it can be implemented efficiently, can detect zero-day attacks, and can be customized to particular operating environments. Additionally, signature-based approaches can be difficult to train and suffer from high false positive rates [49].

The closest IDS work to ours used n-grams of system calls to detect misbehaving Unix processes [17, 18, 48]. Other relevant work includes hardware-based malware detectors (HMD) and architectural side-channel detectors. An HMD monitors micro-architectural traces and raises alerts about anomalous behavior (e.g., [23]). HMDs can detect side-channel attacks that leave no system call traces and can potentially be secured against a compromised OS [65]. For example, Demme *et al.* used performance counters as the data source for an HMD [13], though there are concerns about using performance counters in this domain [12]. Similarly, Wei *et al.* proposed a power anomaly detection system for embedded systems which can detect side-channel attacks, including Spectre, with high accuracy [59]. However, this method targets embedded systems that run fixed jobs with consistent behavior. Leach *et al.* used hardware CPU features to detect attacks against cloud infrastructure [26], but their technique detected only certain classes of scheduler and resource usage attacks. We extend these previous HMDs to sequences of memory accesses in a host-, state-, anomaly-based approach for general workloads. This allows Martini to detect both conventional attacks and memory-based side-channel attacks accurately.



(a)

(b)

Figure 1: Two functionally equivalent finite automata, NFA (a) and homogeneous NFA (b), which both recognize four spellings of 'doughnut.' Starting states are indicated with an attached arrow and final states have a double edge. Embedding transition information inside each state admits efficient hardware implementation.

2.3 Finite Automata

We use *non-deterministic finite automata* (NFAs) to represent and monitor memory access patterns of executing programs. Formally, an NFA is a five-tuple, $(Q, \Sigma, Q_{start}, \delta, F)$, where Q is a finite set of states, Σ is a finite alphabet of input symbols, $Q_{start} \subseteq Q$ is the set of initial states, $\delta : 2^Q \times \Sigma \rightarrow 2^Q$ is a transition function encoding transfer of control between states based on an observed input symbol,[2] and $F \subseteq Q$ is a set of accepting states.

A *homogeneous NFA* specializes NFAs by restricting the transition function such that all incoming transitions to a given state occur on the same input symbol. NFAs and homogeneous NFAs have equivalent representative power [7] (Figure 1), and because all transitions into a given state occur on the same symbol(s), we can embed transition rules in the states. Following Dlugosch's *et al.* nomenclature, we refer to these combined state/transition rules as *state transition elements* (STEs) [14].

An NFA is typically said to have accepted its input if it is in an accepting state after the entire input has been processed. To support streaming of input data (memory accesses in Martini), we

[2] For architectural reasons, we choose a definition of NFAs without ε-transitions. In our formulation, an ε-transition is encoded by duplicating all incident transitions to a source state onto the target state [14].

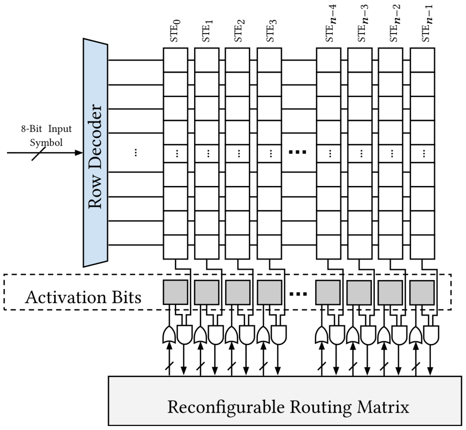


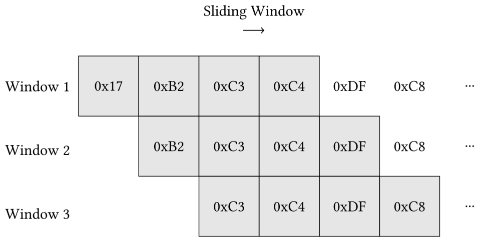
Figure 2: Cache Automaton (CA) architecture. STEs are stored in a memory array, and edges are encoded in a reconfigurable routing matrix. This architecture enables compact implementation of NFAs in hardware.

extend this definition to a notion of *reporting*: if at any point during input processing, an accepting state becomes active, a *report signal* is generated to indicate that the encoded pattern has been found (and is therefore authorized). If an NFA does *not* report on a given sequence of memory addresses, then that sequence is interpreted as unauthorized. Therefore, the NFAs represent authorized program behavior, with sequences of memory addresses as input.

2.4 Hardware acceleration and Near-Memory Computation

A recent body of work studies the acceleration of automata processing. There have been several efforts to develop memory-centric architectures for automata processing, such as Micron's D480 Automata Processor (AP) [14], Subramaniyan *et al.*'s Cache Automaton (CA) [51], Parallel Automata Processor [50], and Xie *et al.*'s REAPR [61]. These architectures enable fast, efficient implementations of automata-based computation [39, 56].

Because the CA is most similar to our approach, we describe our memory-centric automata processing with respect to it. Figure 2 depicts the CA architecture. SRAM arrays in the last-level cache (LLC) are repurposed in the CA to encode both STEs and transition rules. STEs are mapped to individual columns in the array. The NFA in Figure 1(b) requires eight columns to execute. To perform computation, 8-bit input symbols are fed through the row address lines of the array, and the row decoder drives a single row in the array (effectively implementing a one-hot encoding of the input symbol). An STE matches the input if a 1 is stored in the row associated with the input symbol. In a single clock cycle, all STEs simultaneously determine whether they match the input. STEs that match the input and are active (as determined by an additional one-bit register stored with each column) generate a transition signal, which is fed into a reconfigurable routing matrix to update the activation bit registers for the next cycle of computation. In brief, this construction implements a given NFA transition function in hardware.



Window 1	0x17	0xB2	0xC3	0xC4	0xDF	0xC8	···
Window 2		0xB2	0xC3	0xC4	0xDF	0xC8	···
Window 3			0xC3	0xC4	0xDF	0xC8	···

Figure 3: Example of a four-address fixed-width window. Here, an executing program accesses addresses 0x17, 0xB2, 0xC3, etc. Each window (*n*-gram) thus represents a local snapshot of accessed memory locations as the window slides across all memory accesses.

We leverage Subramaniyan *et al.*'s design for a novel sense-amplifier cycling technique, but demonstrate that NFAs for our application-specific deployment for monitoring memory accesses have a uniform and trivial topology that allows for significantly smaller interconnect than the one used in the CA.

3 TECHNICAL APPROACH

In this section, we describe Martini's design and implementation followed by a description of our proposed hardware architecture for monitoring memory accesses without impacting runtime overhead or classification performance.

3.1 The Memory Access Pattern Abstraction

Many abstractions have been proposed to compactly characterize program behavior, including the cadence of cache misses [63], program counters [45], taint tracking in I/O inputs [52], and hardware performance counters [12, 47]. Despite their ability to identify anomalous behavior, these models typically require intrusive instrumentation that degrades system performance. We aim for an abstraction of memory access patterns that is suitable for low-overhead, high-accuracy real-time monitoring.

Programs are represented as data stored in memory, and program execution proceeds by reading, modifying, and storing data in memory. Program behavior therefore partially manifests as a sequence of memory accesses produced during execution. These sequences are inherent to the underlying execution path and structure of program code. That is, alterations to the way in which a program processes information are revealed by its memory access patterns. For example, a calculator program that parses and interprets expression strings will generate distinct memory traces when multiplying vs. adding operands due to variations in the execution's control flow. By contrast, changing the operands (i.e., the numerical data) will typically not result in a change in the trace of memory addresses. The Martini design leverages this insight to characterize program execution as either benign or malicious (i.e., either authorized or unauthorized by the system operator) in a way that, ideally, generalizes to subsequent inputs.

Instead of considering a program's unique sequence of memory accesses as a whole, we propose a stream-based approach that can scale to arbitrary-size programs, observing a fixed-width window of

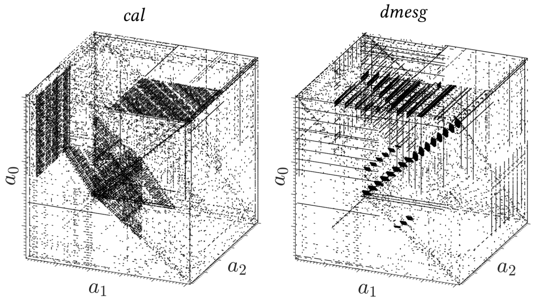
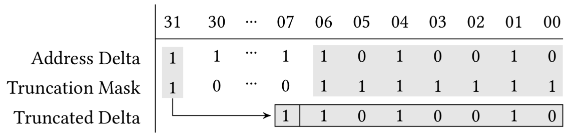


Figure 4: Visualization of n-gram representation for different programs. Sets of windows of size three are shown for memory accesses of *cal* and *dmesg*. Each point (a_0, a_1, a_2) in three-dimensional space represents a unique window recorded during the execution of the program, where a_n represents the n^{th} address in the window. The plots are structurally different between the two programs indicating significant differences in their behavior. We consider such windows in 8 dimensions.



	31	30	···	07	06	05	04	03	02	01	00
Address Delta	1	1	···	1	1	0	1	0	0	1	0
Truncation Mask	1	0	···	0	1	1	1	1	1	1	1
Truncated Delta				1	1	0	1	0	0	1	0

Figure 5: Example of address truncation. Memory address deltas are bitwise AND'd with a truncation mask and packed into 8-bit values. In practice, a sign bit and the seven least significant bits produce accurate results.

the n most recently accessed memory addresses. This window acts as a shift register, allowing MARTINI to observe a *sliding window* of memory addresses as program execution proceeds. This provides a localized, contextual view of a program's recent memory behavior that can be monitored during the execution of the program. Figure 3 provides an example of the construction of windows of width four from a stream of memory addresses.

Although any individual n-gram is likely shared across the execution of different programs, we hypothesize that the *set* of all windows for a given program provides a unique signature that is difficult to spoof. This is the intuition behind our approach—programs are characterized by the pattern of memory addresses accessed during execution. As an example, consider three-address windows: Figure 4 plots windows for the Linux utilities *cal* and *dmesg* in three-dimensional space, where each dimension represents one bit of the address. Each point represents a unique sequence of three memory addresses recorded during the execution of the program. The two plots are structurally dissimilar (e.g., the dense behavior on the "center-left," a_0–a_2 region, for *cal*), which shows how a simple comparison of fixed-width memory access window sets will differentiate the execution of different programs. We rigorously evaluate this hypothesis in Section 6.

3.2 Dictionaries of Program Behavior

Next, we extend the notion of memory access windows to (1) allow a system operator to define a collection of authorized programs and (2) compactly represent sets of valid windows.

Statically determining the exact execution path of a program is undecidable. Instead, we sample many indicative memory traces from each authorized program. We next construct a *dictionary* with all of the windows generated by this *training* set. The dictionary is an abstract model of authorized program behavior. New programs and traces may be added to a dictionary without retraining on existing traces. Similar to other IDS approaches, the quality of the model is determined by the extent to which the training set generalizes to all normal behaviors. In our experience, however, most programs have highly conserved execution patterns under benign inputs.

Two related challenges in offline learning of labeled training data include overfitting and model size [9, 19]. We require a solution that avoids overfitting (so that it will generalize to untrained benign program input data for high-assurance whitelisting) and that admits a compact representation (so that it can be efficiently deployed in hardware, such as in a Cache Automaton setting). To address these challenges, we propose three additional refinements:

3.2.1 Δ-Windows. Defensive techniques such as address space layout randomization (ASLR) randomize important memory locations of processes to harden systems against classes of exploits [46]. For MARTINI, the execution of identical processes could produce significantly different absolute memory traces. We generalize otherwise identical memory traces by storing the *distance between* consecutively accessed memory addresses rather than absolute locations. While absolute addresses can vary across executions, we hypothesize that these distances, or *deltas*, likely remain constant and generalize. We refer to windows of address deltas as Δ-*windows.*

3.2.2 Truncation. Δ-windows mitigate some of the risk of overfitting due to address randomization, but differences between physical and virtual addresses remain. We mitigate this by *truncating* the address delta values to b bits, excluding bits in the delta that may be specific to the physical page selected at runtime. This also significantly reduces the address space represented by our model, helping generalize the model and reduce overfitting. MARTINI supports general masking of the address deltas. Although a full parameter sweep falls outside the scope of this work, our experimentation showed that storing a sign bit and the seven least significant bits of the address deltas produces good results. An example of address delta truncation is given in Figure 5.

3.2.3 Compression. Simply truncating deltas (e.g., to 8 bits) improves the model, but is still insufficient for our needs. For example, for Δ-windows of length 8 containing 8-bit truncated deltas, there are $2^{8 \cdot 8} = 2^{64}$ unique values that could be stored in a dictionary. Even when storing fewer than half of these values, we found that a dictionary trained on a subset of Linux Coreutils contained approximately 40 million windows, which is several orders of magnitude larger than what MARTINI can efficiently support (Section 4).

To address this scalability challenge, we compress dictionaries using a method similar to earlier work on system calls [18]. In this scheme, the first element of a window is stored exactly, but each subsequent position is represented by the *unordered set* of all observed values at that offset from that starting element. For example, if the windows $\langle c, a, t\rangle$, $\langle c, o, w\rangle$, and $\langle d, o, g\rangle$ were observed, the compressed dictionary would store $\langle c, \{a, o\}, \{t, w\}\rangle$ and $\langle d, \{o\}, \{g\}\rangle$. Note that this compressed dictionary accepts the original three windows as well as "caw" and "cot"; the compression is not lossless. Additionally, "dog" is stored separately in the compressed dictionary because its first element is distinct. Thus, while the compression generalizes a dictionary, it also reduces the size, admitting efficient hardware implementation.

For windows of length k consisting of b-bit deltas, the number of possible values stored in the compressed dictionary reduces from $2^{k \cdot b}$ to $k \cdot 2^{b}$. Our empirical evaluation in Section 6 demonstrates that compressed dictionaries retain sufficient fidelity to detect unauthorized program execution, including difficult-to-observe hardware side-channel attacks.

3.3 Detecting Anomalous Program Execution

During the training phase, Martini records all of the memory traces associated with runs of authorized programs on indicative workloads. We consider fixed-width sliding windows of addresses from those traces, convert adjacent addresses to Δ-windows, truncate each delta to a smaller number of bits, and finally generate a compressed dictionary to store (an over-approximation of) the set of truncated Δ-windows associated with those programs and runs. Our experimental results show that such sets of abstracted memory addresses characterize program behavior in a way that is sensitive to the classes of anomalies in which we are interested.

After training, we determine whether a new sequence of memory accesses matches the model by converting incoming accesses to a truncated Δ-window and querying the dictionary for membership. If observations fall outside the dictionary, Martini flags the sequence as anomalous (and possibly malicious). We refer to these anomalous sequences as *mismatches*. A *mismatch counter* c, initialized to zero, increments by one whenever Martini detects a mismatch. The mismatch counter is multiplied by a *decay* coefficient d ($0 \leqslant d < 1$) every N windows to retain local context and eventually forgive past mismatches. The *mismatch rate* r ($0 \leqslant r < 1$) is defined as $r = (1 - d)c/N$. An alarm triggers when the mismatch rate exceeds a predefined threshold t. Briefly, the mismatch rate reflects the concentration of mismatches at any point in time. This allows the system to tolerate some false positives, while still responding to legitimate deviations. t controls the sensitivity of the system and allows Martini to be configured to optimize the tradeoff between false- and true-positives in different settings. Proper tuning of thresholds has been demonstrated to mitigate many false positives [48]. We evaluate mismatch rate thresholds in Section 6.

This work focuses on detecting anomalies, but what happens after an anomaly is reported? Here are two reasonable responses: (1) the alarm signal could trigger the OS to terminate the process or (2) the memory system could delay completion of the memory transaction. Termination would require careful implementation to avoid denial-of-service and livelock issues. A delay-based approach can be effective in some OS settings because users can often tolerate an occasional small delay but attackers typically cannot [48]. For some versions of Spectre and Meltdown, additional delays would explicitly defeat relevant timing-based calculations (Section 2.1). We view either of these scenarios as plausible but leave the development of a robust demonstration for future work.

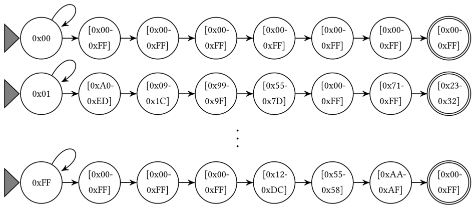


Figure 6: Homogeneous NFA representation of a dictionary. The NFA consists of 256 connected components, each containing a chain of eight STEs. The initial STE for each component matches a unique value; all subsequent STEs match a set of possible values. Training determines the values within.

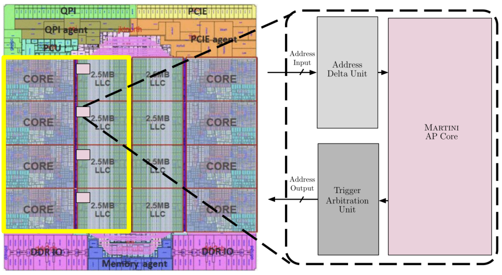


Figure 7: High-level architectural design of Martini: an 8-core Xeon processor, each with private L1 and L2 caches and shared 2.5MB Last-Level Cache (LLC) slice with embedded Martini processor, and a block diagram of the Martini processor (shown in pink). Note that regions are not to scale.

4 MARTINI ARCHITECTURAL DESIGN

Having detailed Martini's IDS design, we now describe its microarchitectural design and efficient implementation. First, we present the homogeneous finite automaton compressed dictionary representation. Then, we describe our proposed architecture for monitoring memory accesses, which is embedded in the last-level cache (LLC) of the CPU.

4.1 From Dictionaries to Automata

We represent compressed dictionaries as homogeneous NFAs (Section 2.3) to facilitate hardware implementation and execution. A separate automaton, or *connected component* [51], is created for each Δ-window in the trained dictionary. Because there is a single entry in the compressed dictionary for each unique address delta that begins a window, b-bit deltas translate to 2^{b} connected components.

Within a given automaton, we allocate one STE for each window offset, which forms a linear chain. The symbol match conditions are taken directly from the compressed dictionary. Additionally, each initial STE contains a self-loop to account for the sliding window comparison. The last state in each chain (equivalent to the final position in the window) generates a report signal if activated. In

Martini Figure 6 illustrates the automata layout. When a new dictionary is trained, the overall automata topology is unchanged; only the symbols within individual STEs change. This insight allows us to simplify hardware-level routing to save space in silicon.

The automata input is the sequence of truncated memory address deltas generated by program execution, and reports are generated for every input Δ-window that matches the encoded dictionary.

4.2 Martini Address Monitor

To support real-time monitoring of memory accesses, we embed Martini in the last-level cache (LLC) region of the CPU. Figure 7 shows an enterprise 8-core Intel Xeon-E5 processor. The Xeon family of processors typically includes 8–16 slices of LLC (one slice per core) [6, 8, 20]. In our prototype, each processor core is allocated a dedicated Martini unit (the pink rectangles within each private cache in the Figure). Our Martini unit consists of three components: the Address Delta Unit, Automata Processing (AP) Core, and Trigger Arbitration Unit.

The Address Delta Unit snoops the memory address lines of the core and calculates the truncated delta between two consecutive addresses (as described in Section 3.2). In its simplest form, this unit performs two's complement arithmetic on 8-bit values; however, a more sophisticated unit could support dynamically masking and truncating address deltas.

The generated address deltas are then fed into the AP core, described in the next subsection. This core executes the automata computation, producing triggers when a window of address deltas is not found in the loaded dictionary.

The Trigger Arbitration Unit tracks triggers and generates an alarm signal when a pre-defined threshold is exceeded. Our prototype implementation consists of two counters. The first counter tracks the number of windows processed, while the second counter tracks the number of triggers produced by the AP core. Whenever the window counter reaches its threshold, the trigger counter is shifted to decay the value and favor local context. If the trigger counter reaches its threshold (i.e., the mismatch rate from Section 3.3), the Trigger Arbitration Unit produces a signal indicating an anomaly, which is handled by the OS or memory system.

4.3 Automata Processing Core

The AP Core is responsible for taking an input Δ-window and determining whether it is present in the dictionary. We next describe our prototype design for implementing Martini in a current-generation Intel Xeon CPU, which represents a novel, system-level integration of automata processing.

Our AP Core follows much of the implementation of the Cache Automaton [51]; however, we make several application-specific modifications to reduce space overhead and improve performance. Each 2.5 MB slice of LLC in the Xeon processor is organized into 20 ways, each of which is subdivided into five 32 kB banks. Four of these banks constitute data arrays, while the fifth is used for storing cache state [6, 8, 20]. Internally, the banks used for data arrays are made up of four 8 kB (256 × 256) SRAM arrays. We repurpose these SRAM arrays to perform automata computation. A single SRAM array can accommodate 256 STEs, meaning that to accommodate all 2048 STEs of the compressed dictionary, eight arrays—two banks—are repurposed for the AP Core.

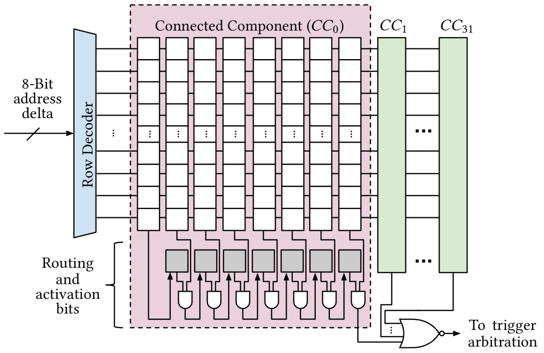


Figure 8: Specialized Martini AP architecture. Routing of activation signals is simplified because connected components in the automata consist of chains of eight STEs (shown in the dashed region). A single 256 × 256 SRAM Array contains 32 connected connected components. We require eight arrays (256 connected components). Masked address deltas are fed as input to the row decoders, and outputs from each connected component are fed to trigger arbitration.

Figure 8 depicts the repurposed SRAM array. As described in Section 2.4, each column encodes the input matching rule for an STE following the state-match design of previous memory-centric AP models [14, 51]. The row decoder converts the current address delta to a 256-bit one-hot encoding. The homogeneity property of the automata ensures that STEs can be represented by a single column of SRAM. Each STE also has a corresponding activation bit. An STE must both match the input symbol and also be active to generate a transition signal. One exception is the initial STE in each connected component: this STE is always active (every cycle is also the start of a new sliding window).

In general-purpose automata processing, a second SRAM array is used to support a reconfigurable routing matrix for transition signals. For Martini, this is not needed; the topology of the automata is fixed, consisting of chains of eight STEs. This allows for static routing in which the transition signal from the previous STE feeds into the activation bit register of the next STE, resulting in a more compact design. The transition signal out of the last STE in each connected component chain feeds into a NOR gate, which aggregates signals from all of the connected components and produces a trigger for the Trigger Arbitration Unit.

4.4 System Integration

Compressed automata dictionaries are (1) placed and routed for hardware resources and (2) stored as a bitmap containing STE input match symbols and the thresholds for the Trigger Arbitration Unit. At runtime, the OS loads the bitmap into the monitoring unit using standard load instructions and Intel Cache Allocation Technology [21]. Anomaly alarms trigger a hardware interrupt, allowing the OS to implement custom mitigation strategies. The configuration overheads are small (roughly equivalent to loading 2 kB of data into the LCC) and typically only occur once. The unit needs to be reconfigured only when loading a new dictionary.

5 EXPERIMENTAL METHODOLOGY

Although we have not fabricated the custom data path described in Section 4, we evaluate the viability of collecting and using memory traces from program executions, and describe our approach for simulating the Address Delta, AP and Trigger Arbitration units.

5.1 Recording Memory Traces

We built two helper tools to collect memory access traces of target programs. First, we leverage an extension to QEMU [4] called PANDA (the Platform for Architecture-Neutral Dynamic Analysis) [15]. Since QEMU is a full-system emulator, this approach has the advantage that we can instrument every instruction executed by the guest system without perturbing its behavior. Second, we used Intel's Pin tool [30] to collect memory traces of userspace programs. In contrast to the QEMU-based approach, Pin can collect memory traces much more quickly, where faithful modeling of the cache hierarchy is necessary. However, the primary disadvantage to Pin is the need to statically modify a target binary, potentially changing memory addresses that are accessed at runtime.

These PANDA and Pin instrumentations are used only in the simulation evaluation to establish a ground truth; they are not part of our proposed deployment. We use both approaches to collect memory traces of a suite of benchmark programs. While Pin instrumentation modifies the software under test, we observed a difference of less than 1%.

5.2 Building and Testing Dictionaries

We next construct a dictionary from the recorded memory traces by applying the refinements described in Section 3.2. First, we calculate the differences between consecutive memory accesses in the traces. Next, we slide a fixed-width window across this data to form 8-delta-long Δ-windows while simultaneously truncating each value to 8 bits. Finally, we construct a dictionary by creating sets of address deltas for each window offset.

We use MNRL to generate automata from a compressed dictionary. MNRL is an open-source state machine representation language and language API intended for large-scale automata processing applications [2, 3]. To simulate the execution of our proposed accelerator architecture, we use *VASim*, a cycle-accurate simulator for automata processing architectures [57]. We extend the simulation to support the operations of the Address Delta and Trigger Arbitration units. In our evaluation, we loaded memory traces from the testing set into *VASim* and processed the data using the compressed dictionary NFA, producing a list of generated alarms.

5.3 Benchmarks

We use multiple software benchmarks as indicative examples of both benign and malicious behavior. Table 1 shows these programs aggregated into one of three benchmark suites based on general behavior. In total, we collect and test on over 2,400 program traces and over 13 billion memory accesses.

The *Coreutils Subset* features a subset of 16 of the Linux *coreutils* programs, commonly used as benchmarks (e.g., [34, 60]). Programs in this suite represent a range of benign applications; we executed each with a variety of command-line arguments to gather memory access traces. The *PARSEC* benchmark [5] is composed of larger multithreaded programs that are designed to simulate a diverse set of highly parallelizable programs (e.g., ray tracing, fluid simulation, video compression, etc.). We use these programs as a test set for evaluating whether Martini can successfully detect execution that is not part of a trained dictionary. The *Security* benchmarks include representative malicious behavior: Spectre, Meltdown and two recent CVEs associated with Linux programs: `dnstracer` and `objdump`. The additional "Detect" column indicates whether Martini can successfully detect the execution of these CVEs using a dictionary of indicative benign programs (see Section 6).

Table 1: Summary of benchmarks used in the experiments.

Coreutils Subset Suite

Program	Version	Traces	Average Trace Length
cal	N/A	269	307,994
cat	coreutils 8.25	227	133,667
cp	coreutils 8.25	175	274,652
date	coreutils 8.25	29	102,393
diff	coreutils 3.3	50	266,856
dmesg	util-linux 2.27.1	50	7,077,967
dnstracer	1.8.1	100	107,111
du	coreutils 8.25	257	6,498,869
grep	2.25	266	429,125
ls	coreutils 8.25	260	884,068
objdump	Binutils 2.26.1	150	1,066,007
ps	procps-ng 3.3.10	30	2,112,550
readelf	Binutils 2.26.1	50	8,911,505
sed	4.2.2	50	370,897
tar	1.28	232	458,187
uname	coreutils 8.25	57	93,281

PARSEC Suite

Program	Version	Traces	Average Trace Length
blackscholes	3.0	1	233,020,782
bodytrack	3.0	1	614,815,076
canneal	3.0	1	946,425,872
dedup	3.0	1	1,005,640,971
facesim	3.0	1	232,921,684
ferret	3.0	1	766,278,169
fluidanimate	3.0	1	640,500,257
freqmine	3.0	1	1,287,177,742
raytrace	3.0	1	1,005,358,609
streamcluster	3.0	1	473,597,488
swaptions	3.0	1	786,879,131
vips	3.0	1	1,596,912,331
x264	3.0	1	233,132,151

Security Suite

Program	Version	CVE	Traces	Avg. Trace Length	Detect
Meltdown	N/A	N/A	64	13,361,880	✓
Spectre	N/A	N/A	52	3,614,351	✓
dnstracer	1.8.1	2017-9430	52	227,616	✓
objdump	Binutils 2.26	2018-6323	50	814,034	✓

We trained dictionaries in our evaluation using a random 60% of the corresponding program's traces. For example, a dictionary containing `diff` would be trained using 30 of its traces, randomly selected.

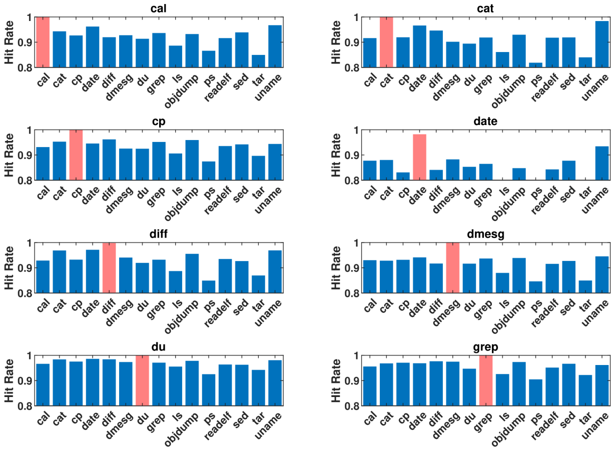


Figure 9: Comparison of memory traces between pairs of Linux utilities. We trained a dictionary using each utility, then measured the ratio of similarities between additional traces of that utility and traces of the other utilities. Red bars compare the trained utility to a subsequent execution of that same utility. Note that the red bar is approximately one for all utilities while all blue bars are less than one, demonstrating that memory traces can effectively identify programs.

6 EVALUATION

In this section, we present an empirical evaluation of Martini and our proposed memory sequence abstraction. We frame our evaluation around the following four research questions to validate design assumptions and investigate system-level hypotheses:

RQ1. Can our proposed memory sequence model identify, and differentiate, program executions?

RQ2. What are the effects of compressing dictionaries?

RQ3. Can Martini distinguish malicious from benign inputs?

RQ4. Can Martini detect unauthorized and malicious programs, including Spectre and Meltdown?

6.1 RQ1. Differentiating Programs

We collected traces of memory accesses for each program in the Coreutils Subset (Table 1) and constructed a dictionary for each utility using 8-access windows from the traces (see Section 3.2). Then, we used execution traces from the other utilities and measured the fraction of tested 8-access windows that are found in the trained dictionary. This experiment measures sequences of memory accesses that are the same between a trained dictionary and some subsequent test program execution. For example, we expect that a dictionary constructed from `ls` will match a high number of memory accesses in a subsequent run of `ls` on different arguments, but will match a low number of accesses from a trace of `cat`, an entirely different program.

Figure 9 summarizes our findings, showing bar graphs for each utility (the remainder show similar results and are elided for space). The x-axis shows the testing program and the y-axis shows the "hit rate," or fraction of 8-access sequences in the testing program trace that matches the dictionary. We gain confidence in our assumptions if (1) testing and training on the same program shows a high hit rate (i.e., the red bar is near 1.0), and (2) testing and training on separate programs shows a low hit rate. The figure shows clear separation between these two measurements, which establishes that we can set a threshold to distinguish between different programs, based only on 8-access sequences of memory accesses.

6.2 RQ2. Effects of Dictionary Compression

Next, we evaluate the effectiveness of dictionary compression (Section 3.2.3). Recall that (1) the primary goal for the compression is minimizing the chip area required for implementation, and (2) we hypothesize that we can compress the model without significantly increasing collisions of Δ-windows, which would cause false negatives. To study this question, we compare Martini's accuracy at classifying authorized vs. unauthorized program behavior, both for uncompressed and compressed dictionaries.

We built compressed and uncompressed dictionaries from traces of 12 of the Coreutils Subset programs. We then used traces from the four CVE proofs-of-concept and the four held-out Coreutils Subset to determine whether those programs would trigger alarms. In this setup, traces from the in-dictionary programs *should not* trigger alarms, and traces from the out-of-dictionary programs *should.* We measured true- and false-positive and -negative data (results are detailed in Section 6.4). We found that the uncompressed dictionary yielded an AUC of 0.9995, while the compressed dictionary yielded an AUC of 0.9928, a trivial decrease in performance. Thus, we conclude that compressing dictionaries does not significantly influence classification performance.

6.3 RQ3. Malicious vs. Benign Inputs

Having established that Martini separates benign from anomalous behavior (e.g., `cal` from `objdump`) and validated that compression is effective, we consider malicious program inputs (e.g., `objdump` normal operation vs. an `objdump` exploit). CVE-2018-6323 describes an unsigned integer overflow in the `elf_object_p` function of `objdump` that can be triggered when it reads a specially crafted ELF file. We generated a training dictionary by running `objdump` over 34 different ELF files. We evaluated with respect to three traces each of 16 benign ELF inputs and traces of the malicious ELF.

We used these to evaluate the detector's performance as a function of the threshold t chosen (see Section 3.3). We say that a trace *passes* if it does not trigger any alerts. Figure 10 plots the pass rate of the held out benign `objdump` dictionary (i.e., true negatives, shown in solid blue) and the held out malicious CVE (i.e., false negatives, shown in dashed red), both as a function of threshold. Note that interval size and decay are configurable parameters discussed in Section 3.3. Thresholds from 0.01 to 0.05 catch *all* malicious behavior with *no* false positives. **This is one of three principal whole-system results:** Martini distinguishes malicious from benign program behavior with accuracy and precision.

6.4 RQ4. Detecting Unauthorized and Malicious Programs

We also investigate Martini's ability to detect unusual or malicious programs, such as unauthorized programs or exploits and attacks.

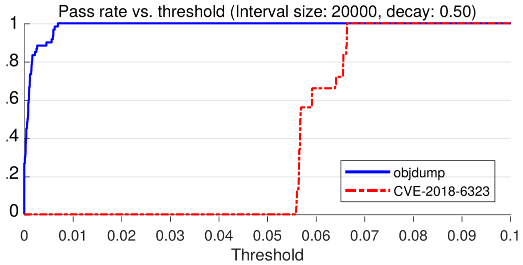


Figure 10: Martini performance on `objdump` CVE-2018-6263, as a function of threshold for benign and malicious inputs. We trained on traces from benign `objdump` inputs; the vulnerability was exploited using the malicious input, and Martini classified the runs as benign or malicious. True negatives are shown in solid blue and false negatives in dashed red. Note that thresholds of 0.01–0.05 allow all benign runs to pass, while raising alarms on 100% of malicious runs.

In this evaluation, we train a dictionary with 60% of all traces of 12 of our Coreutils Subset programs. The resulting model is then simultaneously subjected to four types of testing traces: (1) trained programs, (2) untrained Linux utilities (i.e., the remaining held-out Coreutils), (3) exploits of trained programs, and (4) Spectre and Meltdown proofs-of-concept. In our use case, only trained programs are considered normal; all the others are anomalous and cause Martini to raise an alarm. We can detect anomalous behaviors with a high true positive rate and a low false positive rate.

We are particularly interested in detecting emerging hardware side-channel attacks, such as Spectre and Meltdown. Therefore, we first consider these two attacks, in isolation, before presenting our full system results. Figure 11 shows the pass rates for both Spectre and Meltdown using the dictionary described in this subsection. For comparison, we also show the pass rate of `objdump`, an indicative, authorized program. The horizontal separation between the blue and the two red curves allows a generous range of thresholds that would detect *all* malicious executions with *no* false positive alarms on executions of `objdump`. **This is our second principal whole-system result:** sequences of memory accesses provide a suitable abstraction for detecting emerging hardware side-channel attacks.

There are several reasons this result is significant. First, the vulnerabilities exploited by both Spectre and Meltdown are *not exposed* by the processor's ISA, yet our memory sequence abstraction detects the anomalous behavior. Second, our dictionary-based model is trained using only benign, authorized programs, meaning that exploits need not be known *a priori* to be detected by Martini. Third, our approach, while intended to detect emerging hardware side-channel attacks, does not leverage any exploit-specific insights. Instead, we leverage a general model of memory access patterns. As such, we expect that the results presented here will generalize to other hardware-based exploits as they become available.

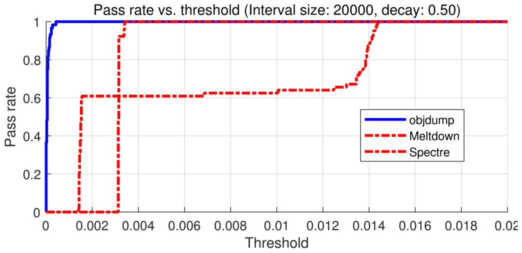


Figure 11: Comparison of pass rates between Spectre and Meltdown and an indicative, authorized program (Objdump). We show one authorized application for comparison and present full-system results separately. The horizontal separation (up and left) between the benign and malicious programs demonstrates the range of thresholds that can detect both attacks while allowing benign executions.

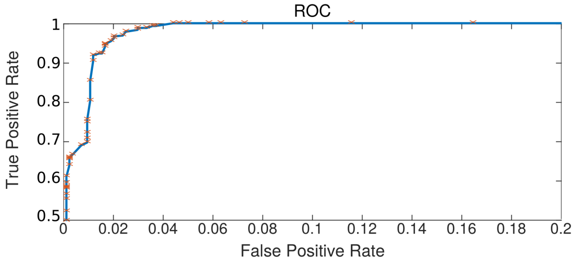


Figure 12: Experimental Martini results for tested anomalies. The ROC curve reports data for all the benchmarks. The blue line reports the average detection rate across all data points using the 12 Coreutils dictionary with 2400 program traces. AUC=0.9954.

Next, we present results using all of the benchmark applications. Figure 12 shows the receiver operating characteristic (ROC) curve of the results. The curve plots the true-positive and false-positive rates parametrically as a function of the threshold: each point on the curve represents a different threshold that can be chosen and thus a different tradeoff in the space. A common metric to evaluate such figures is the Area Under the ROC Curve (AUC); an effective classifier that with high true positives and low false positives has a high AUC. **This is our third principal whole-system result:** when trained on Linux utilities, Martini distinguishes them from other utilities, and *all* of the other benchmark security exploits and side-channel attacks powerfully, with an area-under-curve (AUC) of 0.9954. At 100% true positive rate, our best-performing configuration has a false-positive rate of 4.4%.

We use PARSEC traces as indicative long-running processes that could be considered anomalous with respect to the Coreutils dictionary. We test if Martini detects anomalies *early*, regardless of the trace length (i.e., detection soon after an attack, rather than minutes later). We split PARSEC traces into blocks with 2.5 million memory accesses each and assigned a reasonable threshold. The result is shown in Figure 13. The x-axis of each graph shows execution time (in block index units) and the y-axis shows the mismatch rate. The programs generally fall into two categories. Some (e.g., `bodytrack` and `vips`), consistently trigger alarms. Others, (e.g., `fluidanimate` and `streamcluster`), trigger alarms more sporadically, likely due to coincidental overlap with address sequences in the dictionary. Across all benchmarks, we correctly identified anomalous execution within an average of 85,000 memory references (minimum 20,000, maximum 360,000, stdev 48,400).

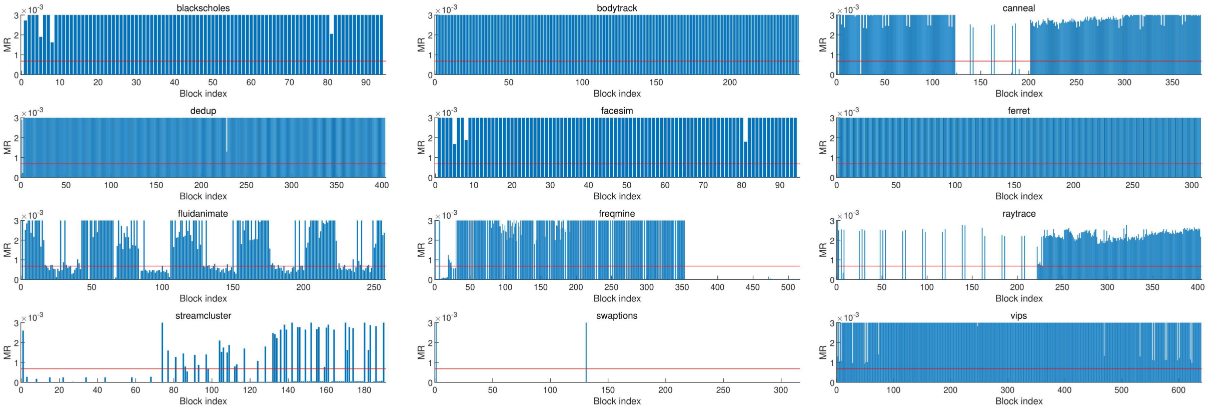


Figure 13: Detection of unauthorized program execution. We trained a dictionary using the Coreutils subset of 16 Linux utilities. Then, we ran each of the PARSEC benchmarks to determine if Martini could correctly identify the anomalous executions. The x-axis of each graph represents execution time in terms of *blocks* of 250 million memory accesses each, and the y-axis shows the mismatch rate.The horizontal line represents a configurable threshold (set to 0.00068 in the figure). These plots suggest that the NFA approach, when combined with a simple counter for mismatches between memory access sequences and the trained dictionary, detects anomalous execution within an average of 85,000 memory references. Using this simple thresholding approach, Martini correctly identifies all benchmarks as anomalous early in their execution.

Table 2: Martini Runtime Overhead.

	Full Cache		Reduced Cache		
Program	Runtime (ms)	Std. Dev.	Runtime (ms)	Std. Dev.	**Change**
blackscholes	152.7	4.7	154.8	3.7	1.42%
bodytrack	457.2	25.5	455.2	30.1	-0.43%
canneal	2774.9	24.3	2809.4	26.1	1.24%
dedup	3236.1	25.3	3242.9	33.8	0.21%
facesim	11.4	0.6	11.5	0.5	0.92%
ferret	467.9	18.4	463.2	12.2	-1.02%
fluidanimate	3.1	0.3	3.1	0.3	-2.00%
freqmine	1053.8	78.5	1042.6	71.5	-1.07%
raytrace	2798.6	10.8	2794.5	11.2	-0.15%
streamcluster	35375.0	8428.7	32491.0	8015.4	-8.15%
swaptions	3.7	0.5	3.8	0.4	0.45%
vips	259.0	10.1	259.7	10.1	0.27%
x264	718.9	8.9	717.2	9.8	-0.23%

For this benchmark suite, anomalous behavior is detected almost immediately (i.e., the blue bar crosses the read line very far to the left on each subgraph). Second, the overall performance is quite high: 91.87% true positive rate and 2.39% false positive rate.

7 ARCHITECTURAL EVALUATION

Having established in the previous section that sequences of memory abstraction provide a suitable abstraction for detecting emerging hardware exploits, we next evaluate Martini's ability to provide low-overhead runtime monitoring. In this section, we present an evaluation of the runtime performance, chip area, and energy consumption of our proposed hardware unit.

7.1 System Performance Impact

Because computation in Martini is decoupled from cache operations, performance impacts are predominantly the result of reduced LLC capacity. We evaluate this impact using the *PARSEC* benchmark suite. We collected runtime performance metrics using a server running Ubuntu 16.04 with 192 GB of RAM and two Intel Xeon Platinum 8275CL CPUs, each with 36 cores running at 3 GHz. Each processor has 36 MB of LLC, subdivided into eleven cache ways. We execute each benchmark twenty times, recording wall clock execution time. Then, using Intel Cache Allocation Technology [21], we reduce the number of cache ways from eleven to ten and execute each benchmark an additional twenty times. One cache way exceeds the resources required by our proposed design for each processor core; the results here present an upper bound for the runtime overhead. Aggregate results are presented in Table 2.

In general, we find that the runtime overhead of our proposed hardware is negligible. In the worst case (blackscholes), we observed a 1.42% increase. The largest change in performance (streamcluster) actually ran 8.15% *faster* with the reduced cache size. We hypothesize that this performance gain is caused by improved cache data alignment. However, any observed performance gain or loss is negligible: using a Wilcoxon signed-rank test, we are unable to find a significant difference in the average execution times for the full and reduced cache configurations ($p = 0.1536$).

7.2 Area Overhead

Next, we study the feasibility of deploying Martini in real silicon by considering a current-generation Intel Xeon CPU. We estimate the area overhead for in-memory automata processing accelerators [1, 51]. We model the area overhead of the proposed AP core with IBM 45 nm `soi12s0` cell library and Synopsys Design Compiler. The total area is 0.016mm^2. For comparison, an 8-core Intel Xeon E5 processor has a die size of 354mm^2 [6] in a 22nm manufacturing process. Thus, our proposed architectural changes would increase the overall die size by less than 0.04%. The synthesis results also shows that all the additional circuits achieves 4GHz frequency after technology scaling to 22nm, maintaining existing frequencies.

7.3 Energy Consumption

The energy required to read out 256 bits from an SRAM array is estimated to be 13.6pJ [1], for the Xeon CPU used by our AP Core. The SRAM arrays operate at 4GHz. As there are eight arrays used in an AP Core, the peak power of our AP Core is estimated at 0.435W. The peripherals of AP Core, the Address Delta Unit, and the Trigger Arbitration Unit together consumes 0.035W of power based on the synthesis results with IBM 45nm cell library. In total, these sum to 0.470W, which is far below the TDP of the Xeon E5 processor core (160 W). Therefore, the proposed architecture does not incur any significant power overhead to the system.

8 DISCUSSION

At a high level, Martini captures program behavior by observing memory access patterns under normal operation and then looking for unusual patterns in subsequent executions. Unauthorized programs and malicious use of authorized programs, including architecture side-channel attacks, both manipulate data differently to accomplish their goals. Defeating this approach would require that a program: (1) mimic the memory behavior of the trained program(s), and (2) manipulate data differently enough to attack the system. Although we do not offer an impossibility proof of such mimicry, we argue that it would be very difficult to accomplish without inventing entirely new exploit strategies. We hypothesize that Martini will apply to subsequent side-channel attacks based on memory accesses or memory timings. Inasmuch as a program accesses memory, both for data and for instructions, this paradigm of detection applies broadly. For these same reasons, efforts to exfiltrate a dictionary from our proposed monitoring unit in the CPU would be difficult without detection. Our evaluation section provides evidence that simple memory traces can be used to characterize normal and anomalous program behavior accurately and precisely (i.e., with high true positives/negatives and low false positives/negatives) and with low runtime overhead (i.e., at native speed if deployed in hardware). This novel combination of microarchitectural insights and anomaly detection provides a useful basis for efficiently detecting architectural side-channel attacks such as Spectre and Meltdown as well as recent CVEs. In this section, we discuss applications and limitations of our technique.

8.1 Applications of Martini

Our approach is motivated by deployment in hardware for low-latency, low-overhead detection of anomalous execution. We demonstrated how Martini detects Spectre and Meltdown—recent architectural side-channel attacks. When such vulnerabilities are discovered, there may be a significant delay between developing software- and hardware-based fixes. Currently proposed hardware fixes incur significant performance overheads (21–72% slowdown in one study [62]), and architectural changes often address a single vulnerability type, leaving the system vulnerable to other attacks. Martini provides a method for detecting unusual patterns of execution, which can protect the system until hardware redesigns are deployed. In brief, we provide a low chip area, low overhead approach to detecting general anomalous execution.

8.2 Limitations of Martini

We consider a high-assurance scenario with a set of authorized programs—any execution not in that approved set is considered anomalous. A more general use-case would focus on malicious execution. Martini captures such behavior for the cases we tested (correctly classifying malicious CVE behavior vs. benign behavior), but increased generality would require overly large, uncompressed dictionaries and violate our chip area constraints. With current technology, our approach is effective in high-assurance, safety-critical scenarios such as avionics or industrial control software.

In addition, after building a dictionary of authorized execution, our approach requires the selection of an appropriate mismatch rate threshold to rapidly detect subsequent anomalous execution. Our evaluation demonstrates that is is possible to determine such a threshold. However, in practice, Martini would require an internal testing or validation set of memory traces to determine the threshold. A system administrator could perform threshold selection as part of a deployment of Martini. Finally, the dictionary and threshold can be updated quickly as Martini uses an SRAM cache in its implementation, which is quick to rewrite and update.

Martini does not directly protect legacy hardware, but our proposed technique requires minimal modification to current hardware designs, protecting future processors and allowing for drop-in replacements. Automata processing architectures have also been embedded in DRAM [14], which suggests that our approach could be implemented in the outer layers of the memory hierarchy. While limiting memory accesses visible to the monitoring unit (i.e., cache-based side-channel attacks would no longer be observable), a DRAM unit could protect legacy systems against a subset of vulnerabilities.

9 CONCLUSIONS

Architectural side-channel attacks such as Spectre and Meltdown potentially impact billions of devices. There is a need for techniques that efficiently and precisely identify when such attacks occur. In this paper, we present Martini, a hardware-assisted approach for leveraging memory accesses of programs to classify behavior as authorized or anomalous. Our proposed implementation strategy uses NFAs appropriate for in-cache computation. We demonstrated that Martini accurately and precisely classifies benign and malicious program execution using a suite of Coreutils programs, PARSEC benchmarks, and four CVEs. Martini demonstrates the important role in-memory computation can play in the development of new detection and mitigation strategies for emerging attacks.

ACKNOWLEDGMENTS

This work was funded, in part, by: AFRL (FA8750-19-1-0501); DARPA (FA8750-19C-0003, HR001119S0089-AMP-FP-029, N6600120C4020); the NSF (CCF 1763918, CCF 1763674, CCF 1908633, IOS 2029696); and the Santa Fe Institute.

REFERENCES

[1] Kevin Angstadt, Arun Subramaniyan, Elaheh Sadredini, Reza Rahimi, Kevin Skadron, Westley Weimer, and Reetuparna Das. 2018. ASPEN: A Scalable in-SRAM Architecture for Pushdown Automata. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture* (Fukuoka, Japan) *(MICRO-51)*. IEEE Press, Piscataway, NJ, USA, 921–932. https://doi.org/10.1109/MICRO.2018.00079

[2] K. Angstadt, J. Wadden, V. Dang, T. Xie, D. Kramp, W. Weimer, M. Stan, and K. Skadron. 2018. MNCaRT: An Open-Source, Multi-Architecture Automata-Processing Research and Execution Ecosystem. *IEEE Computer Architecture Letters* 17, 1 (Jan 2018), 84–87. https://doi.org/10.1109/LCA.2017.2780105

[3] Kevin Angstadt, Jack Wadden, Westley Weimer, and Kevin Skadron. 2017. *MNRL and MNCaRT: An Open-Source, Multi-Architecture State Machine Research and Execution Ecosystem*. Technical Report CS2017-01. University of Virginia.

[4] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41. 46.

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (Toronto, Ontario, Canada) *(PACT '08)*. ACM, New York, NY, USA, 72–81. https://doi.org/10.1145/1454115.1454128

[6] William J. Bowhill, Blaine A. Stackhouse, Nevine Nassif, Zibing Yang, Arvind Raghavan, Oscar Mendoza, Charles Morganti, Chris Houghton, Dan Krueger, Olivier Franza, Jayen Desai, Jason Crop, Brian Brock, Dave Bradley, Chris Bostak, Sal Bhimji, and Matt Becker. 2016. The Xeon® Processor E5-2600 v3: a 22 nm 18-Core Product Family. *J. Solid-State Circuits* 51, 1 (2016), 92–104. https://doi.org/10.1109/JSSC.2015.2472598

[7] Pascal Caron and Djelloul Ziadi. 2000. Characterization of Glushkov automata. *Theoretical Computer Science* 233, 1 (2000), 75–90.

[8] Wei Chen, Szu-Liang Chen, Siufu Chiu, Raghuraman Ganesan, Venkata Lukka, Wei Wing Mar, and Stefan Rusu. 2013. A 22nm 2.5 MB slice on-die L3 cache for the next generation Xeon® processor. In *Symposium on VLSI Technology*. C132–C133.

[9] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2017. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282* (2017).

[10] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. 2018. Inception: System-Wide Security Testing of Real-World Embedded Systems Software. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 309–326. https://www.usenix.org/conference/usenixsecurity18/presentation/corteggiani

[11] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. 2014. A Large-scale Analysis of the Security of Embedded Firmwares. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (San Diego, CA) *(SEC'14)*. USENIX Association, Berkeley, CA, USA, 95–110. http://dl.acm.org/citation.cfm?id=2671225.2671232

[12] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. 2019. SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[13] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. 2013. On the feasibility of online malware detection with performance counters. *ACM SIGARCH Computer Architecture News* 41, 3 (2013), 559–570.

[14] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. 2014. An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 12 (2014), 3088–3098. https://doi.org/10.1109/TPDS.2014.8

[15] Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee. 2013. Tappan zee (north) bridge: mining memory accesses for introspection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 839–850.

[16] European Commission. [n.d.]. https://ec.europa.eu/info/law/law-topic/data-protection_en.

[17] S. Forrest, S. Hofmeyr, and A. Somayaji. 2008. The Evolution of System-Call Monitoring. In *ACSAC '08: Procc. of the 2008 Annual Computer Security Applications Conf.* IEEE Computer Society, Washington, DC, USA, 418–430. https://doi.org/10.1109/ACSAC.2008.54 Invited paper for Classic Papers session.

[18] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. 1996. A Sense of Self for Unix Processes. In *Proceedings of the IEEE Symposium on Security and Privacy (SP '96)*. IEEE Computer Society, Washington, DC, USA, 120–. http://dl.acm.org/citation.cfm?id=525080.884258

[19] Douglas M. Hawkins. 2004. The Problem of Overfitting. *Journal of Chemical Information and Computer Sciences* 44, 1 (2004), 1–12. https://doi.org/10.1021/ci0342472 arXiv:https://doi.org/10.1021/ci0342472 PMID: 14741005.

[20] Min Huang, Moty Mehalel, Ramesh Arvapalli, and Songnian He. 2013. An Energy Efficient 32-nm 20-MB Shared On-Die L3 Cache for Intel® Xeon® Processor E5 Family. *J. Solid-State Circuits* 48, 8 (2013), 1954–1962. https://doi.org/10.1109/JSSC.2013.2258815

[21] Intel. 2017. Cache Allocation Technology. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology

[22] Ionx. [n.d.]. Verisys. https://www.ionx.co.uk/.

[23] Mikhail Kazdagli, Vijay Janapa Reddi, and Mohit Tiwari. 2016. Quantifying and improving the efficiency of hardware-based mobile malware detectors. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 37.

[24] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[25] Aleksandar Lazarevic, Vipin Kumar, and Jaideep Srivastava. 2005. Intrusion detection: A survey. In *Managing Cyber Threats*. Springer, 19–78.

[26] Kevin Leach, Fengwei Zhang, and Westley Weimer. 2017. Scotch: Combining software guard extensions and system management mode to monitor cloud resource usage. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 403–424.

[27] Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, and Kuang-Yuan Tung. 2013. Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications* 36, 1 (2013), 16 – 24. https://doi.org/10.1016/j.jnca.2012.09.004

[28] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[29] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 605–622.

[30] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. ACM, 190–200.

[31] Sergey Maximov. 2018. Performance Implications of the Meltdown and Spectre Fixes. https://www.virtuozzo.com/connect/details/blog/view/performance-implications-of-the-meltdown-and-spectre-fixes.html.

[32] S. Momeni Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan. 2019. HOLMES: Real-Time APT Detection through Correlation of Suspicious Information Flows. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 447–462. https://doi.org/10.1109/SP.2019.00026

[33] Sparsh Mittal. 2016. A survey of recent prefetching techniques for processor caches. *ACM Computing Surveys (CSUR)* 49, 2 (2016), 35.

[34] Martin Monperrus. 2018. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 17.

[35] Francis B Moreira, Matthias Diener, Philippe OA Navaux, and Israel Koren. 2017. Data mining the memory access stream to detect anomalous application behavior. In *Proceedings of the Computing Frontiers Conference*. ACM, 45–52.

[36] Rahul Murmuria, Angelos Stavrou, Daniel Barbará, and Dan Fleck. 2015. Continuous Authentication on Mobile Devices Using Power Consumption, Touch Gestures and Physical Movement of Users. In *Research in Attacks, Intrusions, and Defenses - 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015, Proceedings*. 405–424. https://doi.org/10.1007/978-3-319-26362-5_19

[37] Zhenyu Ning and Fengwei Zhang. 2019. Understanding the security of ARM debugging features. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19)*.

[38] North American Electric Reliability Corporation. [n.d.]. Compliance and Enforcement. https://www.nerc.com/pa/comp/Pages/default.aspx.

[39] Marziyeh Nourian, Xiang Wang, Xiaodong Yu, Wu-chun Feng, and Michela Becchi. 2017. Demystifying Automata Processing: GPUs, FPGAs, or Micron's AP?. In *Proceedings of the International Conference on Supercomputing* (Chicago, Illinois) *(ICS '17)*. ACM, New York, NY, USA, Article 1, 11 pages. https://doi.org/10.1145/3079079.3079100

[40] Stefano Ortolani, Cristiano Giuffrida, and Bruno Crispo. 2011. KLIMAX: Profiling Memory Write Patterns to Detect Keystroke-Harvesting Malware. In *Recent Advances in Intrusion Detection*, Robin Sommer, Davide Balzarotti, and Gregor Maier (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 81–100.

[41] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Cryptographers' track at the RSA conference*. Springer, 1–20.

[42] Thomas Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David Eyers, Jean Bacon, and Margo Seltzer. 2018. Runtime Analysis of Whole-System Provenance. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. ACM, New York, NY, USA, 1601–1616. https://doi.org/10.1145/3243734.3243776

[43] Martin Roesch. 1999. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX Conference on System Administration* (Seattle,

Washington) *(LISA '99)*. USENIX Association, Berkeley, CA, USA, 229–238. http://dl.acm.org/citation.cfm?id=1039834.1039864

[44] RTCA, Inc. 1992. DO-178B: Software Considerations in Airborne Systems and Equipment Certification.

[45] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati. 2001. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy* (Oakland).

[46] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the Effectiveness of Address-space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security* (Washington DC, USA) *(CCS '04)*. ACM, New York, NY, USA, 298–307. https://doi.org/10.1145/1030083.1030124

[47] Baljit Singh, Dmitry Evtyushkin, Jesse Elwell, Ryan Riley, and Iliano Cervesato. 2017. On the Detection of Kernel-Level Rootkits Using Hardware Performance Counters. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (Abu Dhabi, United Arab Emirates) *(ASIA CCS '17)*. ACM, New York, NY, USA, 483–493. https://doi.org/10.1145/3052973.3052999

[48] Anil Somayaji and Stephanie Forrest. 2000. Automated Response Using System-Call Delays. In *Proceedings of the 9th USENIX Security Symposium*. Denver, CO.

[49] Yingbo Song, Michael E. Locasto, Angelos Stavrou, Angelos D. Keromytis, and Salvatore J. Stolfo. 2007. On the Infeasibility of Modeling Polymorphic Shellcode. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '07)*. Association for Computing Machinery, New York, NY, USA, 541–551. https://doi.org/10.1145/1315245.1315312

[50] Arun Subramaniyan and Reetuparna Das. 2017. Parallel Automata Processor. In *International Symposium on Computer Architecture*. ACM, New York, NY, USA, 600–612. https://doi.org/10.1145/3079856.3080207

[51] Arun Subramaniyan, Jingcheng Wang, Ezhil R. M. Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. 2017. Cache Automaton. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50)*. ACM, New York, NY, USA, 259–272. https://doi.org/10.1145/3123939.3123986

[52] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. 2004. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA, USA) *(ASPLOS XI)*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/1024393.1024404

[53] Tenable. [n.d.]. Nessus. https://www.tenable.com/products/nessus/nessus-professional.

[54] U.S. Department of Health and Human Services. [n.d.]. Summary of the HIPAA Security Rule. https://www.hhs.gov/hipaa/for-professionals/security/index.html.

[55] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 991–1008.

[56] J. Wadden, V. Dang, N. Brunelle, T. Tracy II, D. Guo, E. Sadredini, K. Wang, C. Bo, G. Robins, M. Stan, and K. Skadron. 2016. ANMLzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *International Symposium on Workload Characterization (IISWC '16)*. 1–12. https://doi.org/10.1109/IISWC.2016.7581271

[57] Jack Wadden and Kevin Skadron. 2016. *VASim: An Open Virtual Automata Simulator for Automata Processing Application and Architecture Research.* Technical Report CS2016-03. University of Virginia.

[58] Steven Walton. 2018. Patched Desktop PC: Meltdown and Spectre Benchmarked. https://www.techspot.com/article/1556-meltdown-and-spectre-cpu-performance-windows/page4.html.

[59] Shijia Wei, Aydin Aysu, Michael Orshansky, Andreas Gerstlauer, and Mohit Tiwari. 2019. Using Power-Anomalies to Counter Evasive Micro-Architectural Attacks in Embedded Systems. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 111–120.

[60] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[61] T. Xie, V. Dang, J. Wadden, K. Skadron, and M. Stan. 2017. REAPR: Reconfigurable engine for automata processing. In *27th International Conference on Field Programmable Logic and Applications (FPL '17)*. 1–8. https://doi.org/10.23919/FPL.2017.8056759

[62] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *The 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-51)*. 428–441. https://doi.org/10.1109/MICRO.2018.00042

[63] Mengjia Yan, Yasser Shalabi, and Josep Torrellas. 2016. ReplayConfusion: Detecting Cache-based Covert Channel Attacks Using Record and Replay. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO-49)*. IEEE Press, Piscataway, NJ, USA, Article 39, 14 pages. http://dl.acm.org/citation.cfm?id=3195638.3195685

[64] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 719–732.

[65] F. Zhang, K. Leach, A. Stavrou, H. Wang, and K. Sun. 2015. Using Hardware Features for Increased Debugging Transparency. In *2015 IEEE Symposium on Security and Privacy*. 55–69. https://doi.org/10.1109/SP.2015.11