PerSpectron: Detecting Invariant Footprints of Microarchitectural Attacks with Perceptron

Samira Mirbagher-Ajorpaz
Computer Science and Engineering
Texas A&M University
College Station, USA
samiramir@tamu.edu

Gilles Pokam
Intel Labs
Santa Clara, USA
gilles.a.pokam@intel.com

Esmaeil Mohammadian-Koruyeh
Computer Science and Engineering
University of California, Riverside
Riverside, USA
emoha004@ucr.edu

Elba Garza
Computer Science and Engineering
Texas A&M University
College Station, USA
elba@tamu.edu

Nael Abu-Ghazaleh
Computer Science and Engineering
University of California, Riverside
Riverside, USA
naelag@ucr.edu

Daniel A. Jiménez
Computer Science and Engineering
Texas A&M University
College Station, USA
djimenez@acm.org

***Abstract*—Detecting microarchitectural attacks is critical given their proliferation in recent years. Many of these attacks exhibit intrinsic behaviors essential to the nature of their operation, such as creating contention or misspeculation. This study systematically investigates the microarchitectural footprints of hardware-based attacks and shows how they can be detected and classified using an efficient hardware predictor. We present a methodology to use correlated microarchitectural statistics to design a hardware-based neural predictor capable of detecting and classifying microarchitectural attacks before data is leaked. Once a potential attack is detected, it can be proactively mitigated by triggering appropriate countermeasures.**

Our hardware-based detector, *PerSpectron*, uses perceptron learning to identify and classify attacks. Perceptron-based prediction has been successfully used in branch prediction and other hardware-based applications. PerSpectron has minimal performance overhead. The statistics being monitored have similar overhead to already existing performance monitoring counters. Additionally, PerSpectron operates outside the processor's critical paths, offering security without added computation delay. Our system achieves a usable detection rate for detecting attacks such as SpectreV1, SpectreV2, SpectreRSB, Meltdown, breakingKSLR, Flush+Flush, Flush+Reload, Prime+Probe as well as cache-attack calibration programs. We also believe that the large number of diverse microarchitectural features offers both evasion resilience and interpretability—features not present in previous hardware security detectors. We detect these attacks early enough to avoid any data leakage, unlike previous work that triggers countermeasures only after data has been exposed.

***Index Terms*—microarchitectural attack defenses, secure architectures, anomaly detection**

I. Introduction

The number of known microarchitectural security vulnerabilities caused by speculative execution has increased sharply in recent years [1]. More recently, microarchitectural data sampling (MDS) vulnerabilities have been introduced: RIDL [2], Fallout [3] and LVI [4] exploit these vulnerabilities to leak data from internal CPU buffers. Also, the new CacheOut [5] attack adds the ability to select which data to leak from the L1 cache to the MDS-based attack.

Various components in a modern processor pipeline are susceptible to attacks [1], exposing many side-channels. A large body of work has been devoted to identifying each of these attacks, with industry responding with appropriate patches. However, this is a cat-and-mouse game as new attacks continue to appear at a regular cadence.

Prior work on detecting side-channel attacks relies on querying hardware performance counters via software to reveal malevolent behavior and prevent data from being leaked [6], [7], [8], [9], [10], [11], [12]. Unfortunately, relying on performance counters poses practical problems that can hurt detection and performance. First, the number of events available to performance counters is limited, *e.g.* on recent Intel processors, performance counters can only monitor up to 4 events at a time. Thus, detection techniques that require monitoring a rich feature space must multiplex counters, potentially degrading accuracy, providing opportunities for evasion, and missing opportunities to detect an attack. Second, because performance counters are accessible in software, an attacker may be able to access them and modulate its attack to evade detection. Third, the sampling rate of performance counters is low enough to allow an attacker to adjust the bandwidth of an attack to fit within the sampling interval of a software based detector.

We propose an alternative approach: a first line of defense that protects systems against broadly-defined microarchitectural attacks by leveraging a distinct *microarchitectural footprint* of each attack to detect and classify these attacks-in-progress. This technique enables countermeasures to be deployed proactively, before the attack can be successful. There is evidence that microarchitectural attacks are being used to hide other, more traditional attacks as well, so it is important for security solutions to be able to detect attacks in the speculative execution feature space [13].

Our hardware classifier can capture the signature of an attack in a much richer feature space, allowing higher classification accuracy. Security guarantees are stronger in systems that rely on microarchitectural statistics and are monitored in hardware. The detector cannot be disabled by software even if the kernel is compromised, which is important for threat scenarios where a compromised kernel may be attempting to compromise a secure enclave [14], [15]. Lastly, our hardware-based detector has a sampling frequency that makes it impossible for an attacker to time its attack to the sampling interval.

Our proposed approach to detecting microarchitectural attacks moves the detection of active attacks to hardware from software, allowing the predictor to efficiently use and monitor a large set of microarchitectural features that is not limited to the commit state and includes speculative instructions. There has been prior work on detecting malware in hardware [16], [17], [18], [19].

However those works were (1) not specific to microarchitectural attacks; and (2) primarily look at features related to committed state such as instruction mixes or memory access distribution. This is important because the signature for microarchitectural attacks is different than the signature of malware. We compare PerSpectron to these and other works in Section VII-B.

Designing such a system in hardware poses multiple challenges. First, we need a way to select a set of features to use as input to the detector. The features should be strongly correlated to known microarchitectural attacks. Identifying features that correlate well with an active microarchitectural attack is challenging, as it requires understanding the subtle interactions between various and plentiful pipeline stages across different processor components. It is also important that these features be discriminatory; otherwise, attackers may be able to evade them. Worse, normal programs may lead to false positives, reducing the effectiveness of the defense.

We propose a perceptron-based algorithm that streamlines the feature selection process. From the 1159 microarchitectural features available in our hardware simulator, our algorithm selects 106 features that show the strongest correlation to a range of microarchitectural attacks, including SpectreV1, SpectreRSB, Meltdown, breakingKSLR, Flush+Flush, Flush+Reload, Prime+Probe, and Calibration. The selected features are then used as inputs to a hardware predictor for microarchitectural attack detection and classification. The attacks that we examine are complex, yet identifiable.

The second challenge is designing the mechanism to predict and classify microarchitectural attacks rapidly and with both high precision and recall. The artificial neural networks that are used currently to recognize microarchitectural attacks in software typically use multiple layers of learned feature detectors that produce scalar outputs. By contrast, we use replicated perceptron detectors in hardware with overlapping features. We know that deep neural networks learn complicated features using multiple hidden layers. We show how replicated-perceptron detectors with a large set of low-level features can be used to detect complicated features related to microarchitectural attacks without hidden layers. We argue that this is a more promising way of dealing with variations in speculative attacks and cache attacks than the methods currently employed.

With any machine learning based detection solution, there are concerns about the ability of the attackers to evade detection by modifying their behavior to fool the detector [19]. We believe that the nature of microarchitectural attacks, and the fact that most of them are timing sensitive, substantially limit the ability of the attackers to evade detection. Moreover, we believe that solutions such as RHMD [19] where multiple detectors are used and invoked stochastically can substantially mitigate the threat of evasion. Another consideration with hardware detection is the threat of false positives: our detectors experience a very low false positive rate. Moreover, we view them as a first line of defense that simply alerts the operating system to enable the invocation of subsequent mechanisms to isolate a suspicious process or to monitor it more closely.

This paper makes the following contributions:

1) A novel use of perceptron learning to detect and classify a broad range of microarchitectural attacks. We introduce new replicated perceptron-based algorithms for the selection of highly correlated invariant features of microarchitectural attacks across, the different pipeline stages, most relevant for attack classification.
2) A comprehensive analysis of microarchitectural features that indicate footprints of an attack. We show using the proposed detection and classification algorithms, that valuable insights pertaining to the properties of microarchitectural attacks can be gleaned from a systematic analysis of the weights of the features in the perceptron design.
3) A hardware design of the proposed detection system and a fast classification system with low area and performance overhead. We provide proof that it is robust to even tweaked variations of considered attacks, including one of Flush+Flush, which prior work could not detect in hardware.

II. Background and Motivation

The following two subsections explain the attacks considered in our work. We categorize them as (1) cache side-channel attacks, which exploit timing differences caused by the lower latency of CPU caches (compared to physical memory), and (2) speculative attacks, which exploit microarchitectural vulnerabilities and leak the data using the above-described side-channel attacks.

In particular, for cache side-channel attacks, we examine Prime+Probe, Flush+Reload, and Flush+Flush attacks. For speculative (also known as transient execution) attacks, Spectre Variant1 (SpectreV1), Meltdown, and SpectreRSB. Our training set is limited to these six attacks.[1]

We then describe the perceptron, a small, single-layered learning network that is cheaply implementable in hardware and is the foundational computational structure of our work.

[1] For practical reasons, our data set does not include all known microarchitectural attacks (*e.g*, some variants of speculation attacks, side channels on other CPU structures, and RowHammer attacks); however, we believe PerSpectron can be trained to detect additional attacks.

A. Cache attacks

In a Prime+Probe attack [20], the attacker primes the cache by filling cache sets with its own code or data, then waits for a set time while the victim executes. Finally, the attacker probes the cache by executing and measuring the time to load each set of the primed data or code. If the victim has accessed some cache sets, it will have evicted some of the attacker's cache lines, which the attacker observes via increased access latencies for those lines.

In a Flush+Reload attack [21], the attacker and the victim share some pages in their address space, *e.g.* via shared libraries. First, the monitored data is flushed from the cache hierarchy. Second, the attacker waits to allow the victim time to access this data. Third, the attacker reloads the memory line, measuring the time to load it. If during the waiting phase the victim accesses the data, the line will be present in the cache and the reload operation will take a short time, exposing the victim's access behavior. If the victim has not accessed the memory line, the attacker will observe a longer access time.

Unlike the above attacks, a Flush+Flush attack [22] does not make any memory accesses and causes no cache misses from the attacker process. Instead, this attack relies on the execution time of the flush instruction: if an address is present in the cache, flushing takes more time. Flush+Flush attacks are stealthy [22], *i.e.* the spy process cannot be detected based on cache hits and misses detection mechanisms [22]. Because Flush+Flush conducts neither caches accesses nor causes misses, it cannot be detected by Cyclone [23], a state-of-the-art side channel detection solution.

B. Speculative attacks

Spectre Variant1 [24], or SpectreV1, exploits branch predictors used to predict the direction of conditional branches. The attacker mis-trains the branch predictor to bypass the boundary check of a data structure and speculatively access secret data out of bounds. As a result, the CPU speculatively brings indexing data into the cache that would not have been loaded otherwise. The attacker uses the secret data to index into its own address space and uses a cache side-channel (such as Flush+Reload) to leak the data.

The *Meltdown* attack [25] relies on the fact that permission checks are performed late in the execution pipeline, and a fault is generated only for a committed instruction. From the time an instruction is marked for permission exception at the beginning of the pipeline, to the time that an exception can be raised, if the attacker makes a (normally unauthorized) memory access, the secret data can be cached and leaked using a side-channel.

SpectreRSB [26] is another Spectre attack variant that exploits the return stack buffer (RSB), a predictor unit to predict the target of return instructions. In this attack, the attacker pollutes the RSB with an unmatched call/return pair, which redirects the speculative control flow of the program to a malicious Spectre-like setup.

C. Perceptron

We address software limitations by moving the detection of attacks to hardware, allowing the selection of features from a larger set of events without incurring performance overhead. Neural network models used in current work feature deep multi-layered networks (*e.g.* RNN) that are not easily amenable to hardware due to design and runtime complexity. We show that using a simple single-layered perceptron can provide a readily implementable solution. Perceptron learning has shown to be implementable in hardware for various applications including branch prediction, prefetching, replacement policies, and CPU adaptation [27]. Recent microarchitectures from Oracle [28], AMD (*e.g.* Bobcat, Jaguar, Piledriver, Zen, etc.), and Samsung [29], [30] are documented as featuring perceptron-based branch predictors.

We adapt the single-layer perceptron model used by prior work for detecting and classifying side-channel attacks. A perceptron is a vector of weights that records correlations between an input vector and a target value. It can be used to classify inputs into one of two classes [31]. We construct a quantized desired response $d(n)$ to a perceptron:

$$d(n) = \begin{cases} +1, & \text{if } x(n) \text{ belongs to malicious class.} \\ -1, & \text{if } x(n) \text{ belongs to benign class.} \end{cases}$$

The perceptron computes the weighted sum of the input patterns $x(n)$ and compares it to a threshold value. If the sum exceeds the threshold, the output of the perceptron is +1, otherwise, is it -1. The perceptron weights are updated after the desired outcome $d(n)$ of the predicted event is known. If the prediction was correct then the weights remain unchanged. Otherwise, the inputs are used to update the corresponding weights: $w(n+1) = w(n) + \mu[d(n) - y(n)]x(n)$ where μ is the learning-rate parameter and the difference $d(n) - y(n)$ is the error signal.

Perceptrons are a natural choice for building a microarchitectural attack detector because they can be efficiently implemented in hardware. Other forms of neural networks such as those trained by back-propagation, and other forms of machine learning such as decision trees, are less attractive because of excessive implementation costs.

III. Threat Model

The main goal of *PerSpectron* is to detect cache-based side channel attacks and speculative attacks and prevent the attacker from leaking the memory contents of a victim program that is otherwise not accessible to the attackers. In the attack scenario, we assume a strong adversary model where the attacker may execute any code with the privileges of a normal user. Our defense also applies for attacks that target secure enclaves, where the attacker may even have operating systems privileges and use them to attack the enclave using microarchitectural attacks, although the appropriate action upon the detection of an attack may be different (for example, we may tear down the enclave). Note that our detectors are specialized to microarchitectural attacks; traditional software vulnerabilities such as memory corruption bugs, or physical side channel attacks, are different threat models and out of scope for our defense.

IV. PerSpectron Overview

A. General Approach

In microarchitectural attacks, information about suspicious activity comes from different parts of the processor. We observe this for different attack variants, as well as attempts to evade detection. For example, in SpectreV1, the attacker trains the branch prediction unit, while other variants use the Branch Target Buffer (BTB) or the return stack buffer (RSB). Figure 1 shows that information on the activity of these attacks hops between input domains.

We can see that while SpectreRSB does not have a high *branchPredindirectMispredicted* value, it has a high *RASIncorrects* value. Meltdown cannot be predicted with a high *SquashedLoads* value, but it causes high *NonSpecInstrsAdded* to the IQ. Different variants of attacks present a problem similar to the *viewpoint* problem of image recognition that standard learning models struggle with [32], [33]. The same is true for cache attacks[2]. Each variant of an attack or evasion presents some common features and some unique compared to others, similar to the way an image does when it is rotated. We need to address this problem to detect different attack variants.

Importantly, common evasion techniques are unlikely to be effective against detecting distinct architectural footprints. For example, using polymorphism to produce different binaries, which can be effective against signature detection or features that look for specific instruction distributions, is unlikely to evade PerSpectron. Since microarchitectural attacks have distinctive time-sensitive microarchitectural footprints (*e.g.*.., misspeculation contention), evasion is difficult. For example, if the attack slows to the point where it cannot execute within the misspeculation window, it is no longer effective. Moreover, we believe that techniques for evasion-resilience [19] used by other hardware detectors can also be applied to PerSpectron to further improve its resilience to evasion. We will explore this idea in future work.

We know that information about attacks moves around the processor and shows up in different places similar to an image being rotated. We should detect these moving signatures in any component of the processor where they appear. Thus, we use the replicated features approach [34], an established solution for this type of problem. The idea is that if a feature vector was useful in detecting one target, it is likely that a similar feature detector with different positions in the input space can detect the replicated information, amplifying the signal and attenuating the noise.

Local features yield the well-known advantages of replicated features. Convolutional networks force the extraction of local features by restricting the receptive of hidden units to be local [35]. Pipeline components already exhibit this property. Table I shows a subset of replicated features from different pipeline stages that we found equally correlated in classifying attacks.

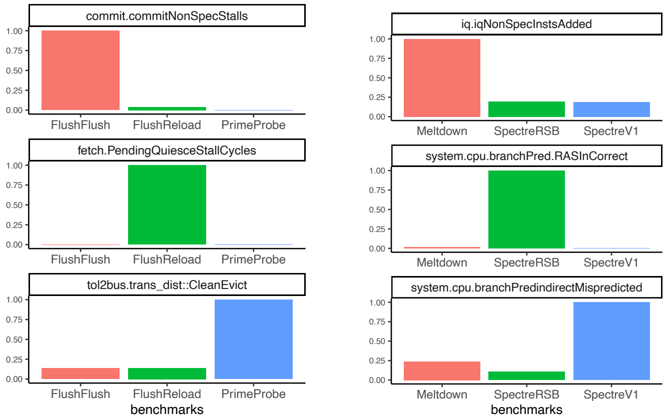

Fig. 1: Information hops between input dimensions. To detect different variants of attacks we need *viewpoint invariant features* and *replicated detectors*.

We do not disregard mutually correlated features of different types, but rather we use them to our advantage. Hinton [32] finds that using the entire space of possible instantiation parameters in the training set allowed the use of a simpler architecture, which could efficiently handle more complex images. In addition, no hidden layer was necessary—the mapping from the features' instantiation parameters to the object's instantiation parameters became linear.

Our solution to this viewpoint problem starts with extracting a large, redundant set of features across each pipeline stage that are invariant under transformation and evasion similar to the invariant feature approach. Ullman [36] observed that a large set of invariant features is sufficient to solve the problem. By using the entire space of possible features in the training set we can efficiently detect multiple variations of attacks and evasions without needing to represent the relationships between features directly because other features capture them. PerSpectron applies the same method in a single layer. Using detailed microarchitectural features in hardware enables us to apply the invariant feature approach.

An example of an invariant feature is the effect of misses and stalls in the Fetch stage. The squashed cycles in each stage, all the ROB, IQ, and Register full events, undone maps in the Rename stage, and memory order violation in the IEW stage propagate back to the Fetch stage. The relationship between these events' Fetch is not a simple cumulative function in an out-of-order processor. However, features such as *fetch.MiscStallCycle* capture the relationship. By including overlapping and redundant features, one feature tells us how other features are related. This solves the domain-hopping problem caused by different variants of microarchitectural attacks and also makes our detector more difficult to evade.

[2] While Flush+Flush does not cause high *PendingQuiesceStallCycles*, it causes abnormal *commitNonSpecStalls*. Also, while Flush+Reload does not cause abnormal *commitNonSpecStalls*, it causes an abnormal number of *PendingQuiesceStallCycles*. While Prime+Probe does not cause abnormally low *tol2bus.transdist.CleanEvict* or *PendingQuiesceStallCycles* values, it causes abnormal *tol2bus.transdist.CleanEvict* values(See Figure 1).

B. Feature selection

In this section, we explain extracting the invariant features and building the replicated perceptron detectors.

The first step of the selection procedure derives, for each of the 1159 features, a measure of mutual information. We use the Pearson correlation coefficient, a number between -1 and 1 that indicates the extent to which two variables are linearly related. We then create a correlation matrix and, based on the values in it, we group features $f_1 \dots f_m$ that are closely correlated into group $g_1 \dots g_c$, $c = 17$ is number of components. Features i and j are closely correlated if the value $|c_{i,j}|$ in the matrix is above a threshold, 0.98. The members of 4 out of 53 corresponding correlation group with $|c_{i,j}|$ above 0.98 are presented in Table I.

The next step is to decorrelate the features within a component, still keep one from every group, g_i. This process seeks to maximize the information about the class contained in each component and minimize the correlation within a component. There was no statistically significant decrease in validation accuracy when we started discarding mutually correlated features within a group, but the accuracy on the validation set decreases significantly when we discard mutually correlated features of different components.

The third step is to select a bank of n features from each component of pipeline. We sort $F_1 \dots F_n$ in each component by the mutual information about the class *benign* or *suspicious*. We then use a greedy iterative selection process, initialized by selecting the f_1 with the highest correlation with the class. Features are added one per each component until the gain in accuracy is small or until a limit on the features m is reached. This method enforces the replication of invariant features.

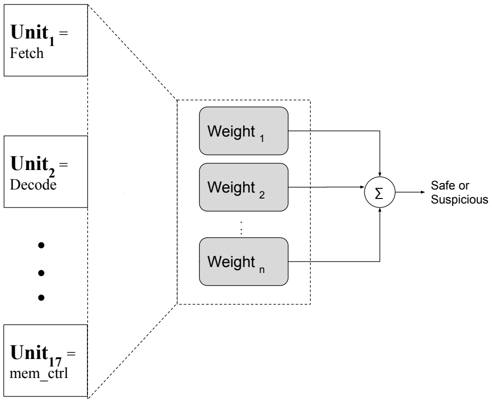

Fig. 2: Architecture of PerSpectron

C. Hardware Design

We simplify our input values into a hardware-friendly representation. In particular, we design a k-sparse binary feature to represent values of microarchitectural statistics, where a k-sparse binary feature has k `1`s in the vector and `0`s elsewhere. Specifically, the k-sparse binary feature vector is represented as $x \in \{0,1\}^u$, where u is the total number of features and the t^{th} entry x_t is a 0/1 indicator, denoting whether the t^{th} scaled statistic is less than 0.5 or higher than 0.5. Scaled statistic t is the value of the t^{th} feature divided by the corresponding maximum value for a given sampling point. Maximum values are stored in a two-dimensional matrix M, with u rows and s columns. The value on the i_{th} row and j_{th} column is the maximum value for the counter i at the execution point j.

The example shown partially in Figure 1 is represented as a vector of inputs that when normalized to the maximum value of the counter would generate input vectors, which in this case are distinctive signatures.

[f1 = ReadResp, f2 = commitNonSpecStalls, f3 = PendingQuiesceStallCycles, f4 = CleanEvict]

$$\begin{aligned} suspicious &:< 0,1,0,0 > \\ suspicious &:< 1,0,1,0 > \\ suspicious &:< 0,0,0,1 > \\ safe &:< 1,1,0,0 > \end{aligned}$$

The above vectors show the one-hot representation and k-sparse binary feature for three attacks and one safe program (example shown in Figure1). We see that regardless of the order and the position of each feature, the k-sparse representation for four program are distinctive, and only contain 0/1 values, thereby simplifying the prediction problem.

D. Training and prediction

We train our perceptron using the 106 features that we selected from offline training. The result of this training is a vector of 106 weights that will be used for building one final perceptron in hardware for prediction (Figure 2).

E. Computing the perceptron output

Since `0` and `1` are the only possible input values to the perceptron, multiplication is not necessary to compute the dot product. Instead, we simply add the weight when the input bit is `1` and subtract when the input bit is `-1`. Furthermore, only the sign bit of the result is needed to make a prediction. This allows the other bits of the output to be computed slowly without waiting for a final value.

F. Performance Overhead

PerSpectron has negligible performance impact since the sampling and prediction happens in hardware and in parallel to the execution. The inference step is not on any critical performance path for the processor. The inclusion of any additional hardware counters not already present in the microarchitecture is also of little concern for overhead, as microprocessor design usually includes hundreds of counters in various parts of the chip as a means to debug and verify processors.

Previous work adapting perceptron learning to branch prediction was done under very tight timing constraints, requiring complex, high-speed arithmetic logic. For example, the original perceptron branch prediction paper suggests an expensive Wallace-tree adder [37] taking the same area and energy as a large integer multiplier circuit. However, for our purposes

group 1	group 2	group 3	group 4
commit.SquashedInsts	commit.SquashedInsts	commit.NonSpecStalls	branchPred.condIncorrect
lsq.squashedStores	iew.lsq.forwLoads	l2.ReadSharedReq_misses	commit.op_class_0::No_OpClass
iew.memOrderViolationEvents	dtlb.rdMisses	rename.serializingInsts	iew.iewExecSquashedInsts
fetch.SquashCycles	iew.SquashCycles	membus.trans_dist::ReadSharedReq	iew.lsq.thread0.ignoredResponses
iew.lsq.forwLoads	fetch.SquashCycles	l2.ReadSharedReq_mshr_miss_latency	iq.iqSquashedInstsIssued
decode.SquashCycles	iq.SquashedNonSpecRemoved	dcache.ReadReq_mshr_miss_latency	iew.iewDispSquashedInsts
iq.SquashedInstsExamined	decode.SquashCycles	l2.ReadSharedReq_accesses	branchPred.RASInCorrect
lsq.squashedLoads	iew.memOrderViolationEvents	l2.ReadSharedReq_miss_latency	iq.fu_full::FloatMemWrite
iew.SquashCycles	fetch.IcacheSquashes	dcache.ReadReq_mshr_misses	commit.op_class_0::FloatAdd
iew.BlockCycles	lsq.squashedLoads	tol2bus.trans_dist::ReadSharedReq	fetch.PendingQuiesceStallCycles
memDep.conflictingStores	iq.SquashedInstsExamined	commit.membars	iew.lsq.thread0.rescheduledLoads
dtb.rdMisses	iew.BlockCycles	dcache.ReadReq_misses	commit.branchMispredicts
iq.SquashedNonSpecRemoved	memDep.conflictingLoads	l2.ReadSharedReq_mshr_misses	branchPredindirectMispredicted
rename.SquashCycles	iq.fu_full::MemRead	membus.trans_dist::ReadResp	commit.op_class_0::SimdCvt
memDep.conflictingLoads	lsq.squashedStores	rename.serializeStallCycles	iq.fu_full::IntAlu
rename.UndoneMaps	rename.UndoneMaps	mem_ctrls.selfRefreshEnergy	iew.branchMispredicts
fetch.IcacheSquashes	memDep.conflictingStores	iq.NonSpecInstsAdded	iew.predictedNotTakenIncorrect
iq.SquashedOperandsExamined	rename.SquashCycles	tol2bus.trans_dist::ReadResp	tol2bus.snoop_filter.tot_requests

TABLE I: Subset of highly correlated features. Features are ranked by correlations between the true class labels and the feature value from top to bottom and left to right. Features in each group have high mutual correlation but belong to different pipeline components and are used to construct *Replicated Perceptron detectors.*

the timing constraints are much more relaxed. The central computation of perceptron-based inference, the dot-product computation, can be performed using a very modest circuit that sequentially adds partial products. This generates a result in a number of cycles proportional to the number of inputs, *i.e.* on the order of 100 cycles for our perceptron. Since the inputs are binary, the circuit simply adds or subtracts weights corresponding to inputs with the value of 1 or 0, respectively. Such a circuit would take negligible area and energy and deliver a prediction in a very timely manner compared with the sampling interval.

G. Using PerSpectron within the System

Once the detector identifies an attack, it is natural to ask how this information will be used within an end to end strategy for defending the system. Because any low-level machine learning detector can experience false positives, we are hesitant to suggest termination of a process that is suspected to be malicious. Rather, we use PerSpectron as a low-level detector that provides information to the system to trigger further mitigations. We discuss two possible classes of mitigations next.

1) Hardware Mitigations

We envision our technique being deployed with the ability to update the neural weights using a vendor distributed patch reflecting training with the most recent known classes of attacks. The weights are small and few so the overhead of distributing and applying the patch would be modest.

The output of a perceptron before being thresholded can be used as a confidence measurement that can also be passed along to the operating system should an attack be suspected. Based on the category of the attack and the confidence, the technique uses a low-cost solution. For example, for mitigating cache attacks, the technique could apply changes to replacement policies to make set eviction harder or enable index randomization in the LLC like Qureshi's CEASER [38], or making unsafe loads invisible in the cache hierarchy [39], [40] to mitigate the attack only when suspicious activity was predicted with high confidence. For mitigating branch predictor attacks, when an attack is suspected, we can inject noise into the branch predictor, or invoke load fences [41] so that it occasionally reverses its taken/not-taken prediction. Increasing the frequency of the noise increases the time for an attack to succeed and decreases performance, however.

2) Software Mitigations

In order to generalize to other classes of attacks, the weights of replicated detectors can be adjusted to cover an arbitrary group of attacks provided that we are exposed to them during training. The detectors are "software defined," meaning there is a secure update process to configure them. They are also replicated per pipeline component. So, we can add classifiers for additional attacks using the same infrastructure if new attacks emerge.

V. Methodology

We use the gem5 [42] cycle-level simulator. Table II gives the parameters of the simulated architecture. We used the FANN C library [43] to implement PerSpectron to work with gem5. We used the scikit-learn python library to measure the performance of other machine learning models, including *K*-nearest neighbors, Decision Tree, Logistic Regression, and Neural Network.

Data: For speculative attacks, we simulate SpectreV1,SpectreV2, SpectreRSB, Meltdown, breakingKSLR (based on Meltdown) and CacheOut attack. For cache attacks, we run Flush+Flush, Flush+Reload, and Prime+Probe along with their respective profiling-calibration phases. Calibration programs threshold distinguishing cache hits and cache misses based on different cache attack techniques [20], [21], [22].

For benign programs, we run individual SPEC CPU 2006 applications [44]. The workloads include C compression programs modified to do most work in memory (rather than I/O), optimization scheduling, Ethernet network simulator, high-rank artificial intelligence programs, discrete event simulation, gene sequence protein analysis, the A* algorithm, and more.

We have written an interface to gem5 that dumps statistics once every 10K, 50K, and 100K instructions, and samples all event counters for each program. Using this tool, we collect multi-dimensional time-series traces of applications.

Statistics: We examined 1159 microarchitectural counters, including features related to cycle accounting and micro-operation (μop) flow, stall decomposition, precise memory access events, latency events, precise branch events, core memory access events, and other core and uncore events. For each event (if applicable) we measure total number, cycles, rate, average and distribution. For each counter, we maintain a maximum possible value for each sampling simulation point. Each statistic is normalized over the maximum value of that counter.

We include events related to stalls and differentiate between types of stalls and among various points in the pipeline. We differentiate between read and write latency, and what came from the master ports and was forwarded to the slave (requests and responses). Likewise, what came from the slave and was forwarded to the master. We also examined features related to energy consumption in different microarchitectural units and pipeline stages. Also tracked are complete power state machine statistics such as Active, Idle, Active Power-Down, Precharge Power-Down, and Self-Refresh values.

Architecture
X86 O3CPU 1 core Single Thread at 2.0GHz
Core
Tournament branch predictor 16 RAS entries, 4096 BTB entries LQEntries=32, SQEntries=32, ROBEntries=192 fetch/dispatch/issue/commit width=8 numPhysIntRegs=256,numPhysFloatRegs=256
L1 I-Cache
32KB, 64B line, 4-way
L1 D-Cache
64KB, 64B line, 8-way
Shared L2 cache

2MB bank, 64B line, 8-way,
mshrs=20, tgtsPerMshr=12, writeBuffers=8
tagLatency=20, dataLatency=20, responseLatency=20

TABLE II: Parameters of simulated architecture

Training and Validation: We train the perceptrons for 1000 epochs, or until the training error falls below 0.4. We used 3-fold stratified splitting with randomization. The class split for each fold for 100K setting is as follows:
Class: [benign malicious]
train - [4820, 1500] — test - [2420, 740]

VI. Security Analysis

We structure the security analysis of PerSpectron around three questions: (1) How well does PersPectron detect the different attacks? We seek to show that it can indeed reliably detect all the classes of attacks it was exposed to during training; (2) Can attackers evade detection by modifying the attacks? We consider different evasion strategies and discover that PerSpectron is robust against evasion attempts; and (3) Could PerSpectron generalize to detect novel cache based or speculation attacks? We show using examples of new attacks (that were not part of the training set) that PerSpectron is indeed able to detect them, likely indicating that it is detecting the basic signatures of these attacks in the microarchitecture. This is an encouraging sign that the approach can help protect against zero day microarchitectural attacks (novel attacks not seen before) even before we include them in the training set.

A. Resilience to evasion

It is natural to expect that an attacker would attempt to change their attack to avoid detection using evasive attacks. In general, this is the problem of adversarial attacks against machine learning, which has been considered even against hardware detectors [19]. Although solutions such as RHMD can also be applied to our detector, we believe that the nature of microarchitectural attacks makes it difficult for attackers to evade detection without interfering with the attack, meaning that the attack features are strong discriminators that cannot be hidden without disabling the attack.

To support this hypothesis, we investigate the robustness of PerSpectron to two evasion techniques in this section: (1) Resilience to polymporphic evasion (where the attacker attempts to produce different binaries implementing the attack); and (2) Resilience to bandwidth reduction mimicry: a common evasion strategy is to slow down the rate of the attack to weaken the microarchitectural signal.

1) Polymorphic evasion

We generated variants of the attack PoC code-snippets used in this study to investigate whether our predictor is resilient to such variations. This is a typical strategy used by malware to evade signature based detectors [45], [46]. PerSpectron was not exposed to any of the modified attacks: they were not used in any of the feature selection phases or final model training. We considered the following transformations:

- Moving the leak to a function that cannot be inlined.
- Add a left shift by one on the index.
- Use `x` as the initial value in a `for()` loop.
- Check the bounds with an `AND` mask, rather than `<`.
- Compare against the last-known good value.
- Use a separate value to communicate the safety check.
- Leak a comparison result in which the attacker is assumed to provide both `x` and `k`.
- Make the index the sum of two input parameters.
- Do the safety check into an inline function.
- Invert the low bits of `x`.
- Use `memcmp()` to read the memory for the leak.
- Pass a pointer to the length.

Figure 3 shows a stacked line chart of perceptron output pre-threshold versus the number of instructions simulated for different versions of a Spectre attack with the same leakage frequency. All variations were detected and triggered the

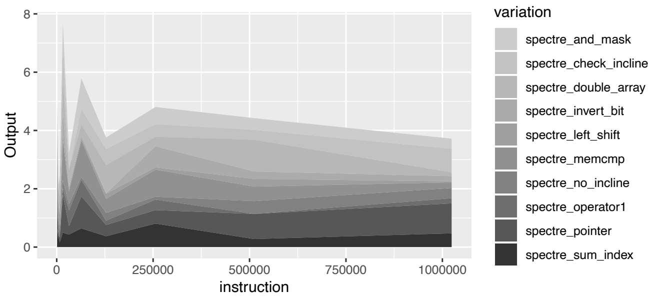

Fig. 3: Perceptron output vs. number of instructions for Spectre variations with same leakage frequency. All variants were flagged as suspicious at the same sampling interval.

perceptron output as suspicious at the same sampling interval, showing that the variations do not cause failure in the detection.

2) Bandwidth reduction evasion

A typical microarchitectural attack encompasses a priming phase used to place the system into an initial state, *e.g.* flushing the cache, a speculation phase that exposes a victim's data, a disclosure gadget phase that readies the data for transmission, and a final disclosure primitive phase that leaks the data to the adversary. We seek to detect suspicious activities before the disclosure phase begins so that appropriate countermeasures are launched on time to prevent leaking the first byte of data.

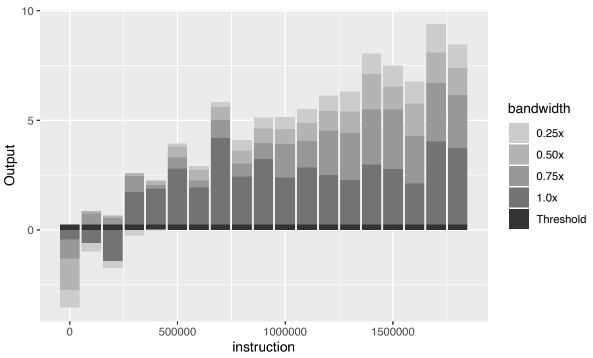

Fig. 4: Perceptron output pre-threshold vs. number of instructions for various SpectreV1 bandwidths.

Figure 4 shows perceptron output pre-threshold versus the number of instructions simulated for various SpectreV1 bandwidths. We reduced the bandwidth of Spectre by 75%, 50%, and 25% over the original attack code by injecting safe source code into the program. The safe code was injected after the disclosure primitive phase as well as before the priming phase, in proportion to the ratio of bandwidth reduction. Our safe code does not tamper with branch history or the secret cache line.

Figure 4 shows that even though the unmodified Spectre (1.0x) saturates the output faster, all lower bandwidth versions stay above the cutoff threshold after the first complete phase of the attack, showing that PerSpectron detects the suspicious activity. Further reducing the bandwidth to less than 25% leads to no leakage of data. While PerSpectron continues to detect an attack at lower bandwidth, data is not leaked even after many tries within 2,000,000,000 executed instructions.

Bandwidth evasion is based on timing the completion of all attack atomic tasks to fit within the sampling interval. It is a problem with the low sampling frequency of the software detector. Li and Gaudiot [47] demonstrated that performance counter-based detectors can be evaded by changing the bandwidth of the attack so that it runs inside the 100 ms sampling interval of the detector. They identified three ”Atomic Tasks” that, if interrupted, will disable the attack: (1) Flushing cache lines - 10 μs (2) Mistraining branch predictor - 13 μs (3) Attempting to infer the secret byte that is loaded into cache - 38 μs. The authors concluded that putting the attack to sleep after all three tasks were completed is the optimum evasion strategy for an attack. This means that the attack runs for 61 μs before being put to sleep, which allows Spectre to run inside the 100 ms sampling interval of the detector. Our sampling interval is 3 μs, which gives 20 sampling intervals within the 61 μs run time it takes to complete all three tasks, making PerSpectron resistant to this evasion strategy. The authors acknowledged that future work should be done on a dedicated hardware detector to reduce performance overhead. Therefore evasion is made more difficult by decreasing the sampling interval to below the run time of essential tasks of the attack.

B. Generalization to Other Attacks

When evaluating trained models at development time, our goal was to estimate the generalization accuracy, *i.e.* the predictive performance of a model on future (unseen) attacks. Thus, we use K-fold cross-validation (CV) to characterize the distribution of possible model behaviors on workloads not present in the training set.

We start by splitting our dataset into three parts: a training set for model fitting, a validation set for model selection, and a test set for the final evaluation of the selected model. We merge the training and validation sets (excluding the test set) after model selection. At every fold, we remove all the samples belonging to attacks in the test set and hence they were not used for model selection. Therefore, our cross validation confidence interval of (0.9979±0.0065) simulates new data that the model has not seen before. Thus, we are naturally evaluating the generalization of the detection to attacks not part of the training or validation set.

We also explored whether PerSpectron can generalize to newly discovered attacks within the category of microarchitectural and speculative execution attacks. Specifically, we consider the CacheOut attack, which was disclosed after PerSpectron was developed [5]. CacheOut attacks Read from Line Fill Buffer, a feature that PerSpectron identified as invariant prior to disclosure of CacheOut. CacheOut (94% true-positive detection) demonstrates PerSpectron's robustness on detecting advanced MDS attacks. We also held out SpectreV2 from any of the independent training set of 3-cross-validation and were able to detect it (91% true-positive) providing evidence of robustness in detecting new variations.

Both CacheOut and SpectreV2 transmit values via a Flush+Reload channel. To assure that PerSpectron is detecting both recovery and transmission portion of attacks, we used different cache channels for the attacks in the test set and the training set of the each CV fold (see Table III).

Note that PerSpectron does not rely on a cache channel for detection; it relies on replicated detectors and invariant features that monitor contention in all CPU components i.e., buffers and ports. Our CV results confirm that there are distinct signatures for these attacks that should hold across their different variations, and by extension to newly discovered ones, which shows our detector to be generalized to more variations.

C. Attack Coverage

Table IV shows that DT-CART [48] with MAP [16] features was not able to detect Prime+Probe and SpectreV1 with a bandwidth smaller than 0.75x post leakage. DT-CART using PerSpectron features was able to detect Prime+Probe but still was not able to detect attacks with bandwidth lower than ($< 0.75x$). KNN was unable to detect calibration-ff, calibration-pp, and Prime+Probe even after leakage. Interestingly KNN could detect polymorphic attacks using PerSpectron features. Logistic Regression with MAP features suffered from high false positives, and could not detect Prime+Probe and polymorphic attacks until post leakage. Perceptron, with the large feature set of 1159, had false negatives on Prime+Probe pre-leakage. A neural network with MAP features suffered from false positives on memory and interrupt intensive workloads when they were dropped out of the training set. Neural network with PerSpectron features had fewer false positives and could detect 11 out of 12 polymorphic attacks. *PerSpectron* could detect all of the attacks before they leaked with only false negatives on initial and pre-leakage samples. This includes the Prime+Probe as well as Flush+Flush attacks[3].

VII. Experimental Evaluation

This section details our experimental results exploring different aspects of the behavior of PerSpectron. Specifically, we first study the impact of the sampling granularity, discovering that prediction accuracy improves up to a sampling rate of 10K instructions. We also compare PerSpectron to other models and feature sets and see that it achieves substantially higher accuracy than these models for the types of attacks we consider. Finally, we explore the individual feature behavior across the pipeline to demonstrate both the interpretability of the behavior of the detector and how the signatures manifest in different pipeline stages, supporting a hypothesis that PerSpectron is both generalizable and difficult to evade.

A. Sampling granularity and threshold

We tested different sampling granularities and threshold values. We present the receiver operating characteristic (ROC) curves in Figure 5 to show the relationship between true positive and false positive rate. The figure shows ROC curves under three different sampling intervals: 10K, 50K and 100K instructions. We can see the 10K interval is better than the 50K and 100K.

We found that a threshold of 0.25 is the best with an area under the curve. PerSpectron has high sensitivity and specificity (AUC of 0.9949) to atomic tasks and benign code. The prediction statistics are as follows: mean accuracy is 0.9979, standard deviation is 0.0015 and accuracy range of (0.9914, 1.0000).

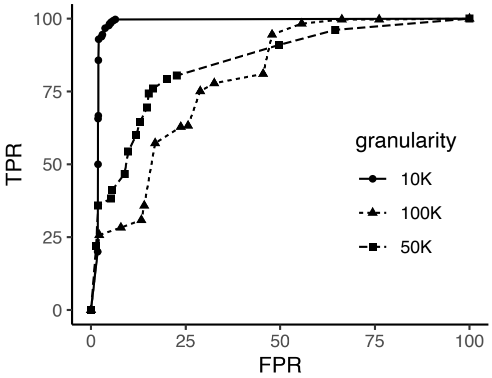

Fig. 5: ROC curve for different sampling granularities. A 10K interval is better than 50K and 100K intervals.

B. Comparison with other models and feature sets

To compare different models' performance, we specifically used standard K-fold cross-validation (CV). Standard CV is the unbiased estimate of prediction error, which is the expected loss on future instances. More precisely, our CV is defined as follows. Let there be $N = |D|$ data cases in the training set. Denote the data in the k'th test fold by D_k and all the other data by D_{-k} (Table III). At every fold, we remove all samples belonging to attacks and benign programs in the test set so they are not used for training the model, analytically removing the effect of the i'th training case. Benign programs on these fold are chosen so the class proportions are roughly equal in each fold.

Our experiments with multi-way classification achieves a near-perfect F1-score on the training set. However, due to the limited number of attacks per category, excluding test and training set for K-fold-CV was impractical for multi-way classification. Thus, in this section we report the accuracy of binary classification comparing with previous works. Table III shows which attacks were chosen for training and which attacks are excluded.[4]

We investigated the features used in previous works that had success using Logistic Regression for detecting malware. Table IV shows Logistic Regression trained with features similar to MAP [16], achieving 75.5% accuracy in detecting

[3] Note that Perspectron does not detect occurrence of *leakage*. PerSpectron detects suspicious activity related to microarchitectural attacks that leads to leakage. Some variations don't leak in the simulation environment.

[4] We excluded CacheOut from all the folds to further stress the CV results—especially to compare the robustness of models on detecting advanced MDS attacks.

k'th test fold	D_k	D_{-k}
1	spectreRSB, spectreV2, cacheOut, breakingKSLR, prime+probe	meltdown, spectreV1, flush+flush, flush+reload
2	spectreV1, spectreV2, cacheOut, flush+reload	meltdown, spectreRSB, breakingKSLR, flush+flush, prime+probe
3	spectreV2, cacheOut, meltdown, breakingKSLR, flush+flush	spectreV1, spectreRSB, flush+reload, prime+probe

TABLE III: Estimating the risk using cross validation. At each fold, we excluded one version of each categories of attacks, trying to predict the attack using a model trained on data that does not contain that attack.

microarchitectural attacks while majority labeling yielded 74.4% accuracy. These results show that the signatures of microarchitectural attacks are different than signatures of malware attacks. In addition, memory, branch, and interrupt intensive benign workloads such as h264ref, povray, gcc, sjeng, and gobmk were shown as false positive using the features from previous work. We also tried Logistic Regression using the 106 highly correlated features we extracted via the PerSpectron algorithm. The accuracy increased to 89.7%, showing that using a highly correlated feature set improves detection accuracy. The K-nearest neighbor algorithm showed a high mean accuracy of 94.8% with the best $k = 3$ and PerSpectron features. However, KNN is not suitable for implementation in hardware due to its high overhead and classification latency.

Table IV shows that the decision tree with CART [48] algorithm has 87.1% accuracy with MAP [16] features and 90.5% accuracy with the 106 highly correlated PerSpectron features. Decision tree, on the other hand, leads to hard decisions in different regions of input space, causing poor performance on classifying attacks that were unseen during the training. Lastly, CART was unable to correctly classify low bandwidth attacks.

We also investigated neural networks using the features from previous works [16], yielding a low accuracy of 80.2%. We noticed that the values of features kept constant during the different phases of attacks show that the previous work did not use features suitable for microarchitectural attack detection. The neural network, when trained with 106 features selected with the PerSpectron algorithm, yields an accuracy of 98.2%. However, a deep neural network has high hardware overhead and classification latency, making it unsuitable for implementation in hardware.

Finally, PerSpectron gave the highest accuracy with 99.7% and a tight confidence interval. PerSpectron does not suffer from a greediness problem similar to decision trees. Unlike the single perceptron, miss-classification by one component can be recovered by the replicated feature detector from the other component of the pipeline.

C. Interpretation through feature analysis

An important aspect of our model is its interpretability. In perceptron based detectors, when features in an active program carry high positive weights, suspicious activity is indicated, whereas a negative weight indicates a benign program. Thus, the weights attached to each feature potentially convey insightful information related to how each feature correlates to the output giving us a detailed understanding of the footprint of these microarchitectural attacks in our feature space. This interpretability contrasts with deep neural network models whose features are automatically generated from the data, and therefore resist interpretability. In the following, we take advantage of this visibility to explore the footprint of the attacks in groups of our features related to different components of the architecture.

Rename unit features: We found that *CommittedMaps* is an invariant feature that contributes highly to different classes of the suspicious activity including speculative and cache attacks. The Committed Map table holds the speculative mappings from ISA registers to physical registers. If enabled, this allows a single-cycle reset of the pipeline during flushes and exceptions of any kind. On every branch, the Rename Map tables are used to allow single-cycle recovery on a branch misprediction. Otherwise, pipeline flushes require multiple cycles to "unwind" the ROB to write back in the rename state at the commit point, one ROB row per cycle. This functionality explains why *CommittedMaps* is invariant to different versions of transient execution attacks. Polymorphic evasions that uses contention on the Register File were detected preleakage after including *CommittedMaps* features even by the simplest models *i.e.* (DT-CART) which shows this feature is highly informative.

The other invariant features from the rename stage are *serializeStallCycles*, *LQFullEvents* and *tempSerializingInsts*. We also observe that Flush+Flush-type attacks have an unusually high number of temporary serializing instructions in the rename unit which also manifest in the commit unit as a high number of stalls.

Memory controller features: One feature that we found to be both strongly correlated to attacks and invariant against a range of them is the *bytesReadWrQ* selected from Memory Controller. The *bytesReadWrQ* tracks the number of DRAM read requests that are serviced by the write queue. We observe that the access rate to the write queue is abnormally high in all attacks early on before the completion of the first iteration of attacks. Including this feature reduces the false positive for memory intensive workloads. This feature illustrates the fact that most attacks attempt to read data freshly evicted from the caches while constructing a covert channel inside the tight speculative window at a rate much higher than typical even in memory intensive workloads.

Another two invariant features from *mem_ctrl* were *bytesPerActivate* with large positive weights, and *wrPerTurnAround* with lowest negative weight. Feature *bytesPerActivate* is the number of accessed bytes per row activation in DRAM and *wrPerTurnAround* is the number of writes before turning the bus around for reads. We noticed that including these two features reduces the number of false positives and remain discriminatory for reduced bandwidth attacks in all the models.

Model	**DT-CART***	**DT-CART**	**Logistic Regression***	**Perceptron**	**KNN**	**NN***	**NN**	**PerSpectron**
Feature	**MAP**	**PerSpectron**	**MAP**	**1159 features**	**PerSpectron**	**MAP**	**PerSpectron**	**PerSpectron**
Mean Accuracy	0.8718	0.9058	0.7594	0.8974	0.9487	0.8026	0.9822	0.9979
95.00% Confidence	0.8718 ± 0.1005	0.9058 ± 0.0120	0.7594 ± 0.0018	0.8974 ± 0.2030	0.9487 ± 0.1435	0.8026 ± 0.0026	0.9822 ± 0.0089	0.9979 ± 0.0065
False Positives $> 20 samples$	dealII,gcc, gobmk,bzip2	dealII, gcc	h264ref,povray,gcc sjeng,gobmk	gobmk,dealII	sjeng,gobmk	dealII,povray bzip2,gobmk	gobmk,dealII	gobmk
False Negatives $post/preleakage$	prime+probe, spectre$< 0.75x$ post leakage	spectre$< 0.75x$, polymorphic post leakage	prime+probe, spectre$< 0.75x$, polymorphic post leakage	prime+probe, spectre$< 0.75x$ pre leakage	calibration-ff/pp* post leakage	prime+probe spectre$< 0.75x$ pre leakage	not post leakage	not post leakage
Hardware Complexity	low	low	low	low	high	high	high	low

**pp: prime+probe*

TABLE IV: ML Model and Feature Set comparison

Another invariant feature is the *selfRefreshEnergy* from *mem_ctrl*. The system's DRAM will enter a low power self-refresh mode when it is not being actively utilized. Many variations cause high volumes of memory accesses to non-cache resident arrays (for example, due to Flush operations common in Flush+X attack variants). Including *selfRefreshEnergy* decreases the false positive rate by 2% and makes the predictor more resilient to polymorphic evasions[5].

L2 bus features: We found that features *ReadCleanReq*, *WritebackClean* and *readsharedreq* remain discriminative across a range of attacks and evasive variations. Essentially, these events capture cache evictions, but do not focus on the mechanism (whether using Flush or cache priming to cause the eviction of a cache line). Our results show previous detections were not able to detect prime+probe correctly without causing an increase in false positives. The reason is prime+probe behaves similar to the large memory footprint applications when monitored by high-level features. We were able to detect prime+probe by including *ReadCleanReq*, *readsharedreq* and *WritebackClean* without causing an increase in false positives.

Fetch unit features: The fetch unit contains highly correlated features that are invariant to variations of stalls, flush, miss and traps used in the polymorphic attacks. The reason is because stalling a pipeline involves stopping not only the pipe stage experiencing *i.e.* the data dependency, but previous stages as well. And therefore such signals propagated backward to the fetch stage. Features *PendingQuiesceStallCycles*, *IcacheSquashes*, *MiscStallCycles* and *PendingTrapStallCycles* are mutually decorrelated in the fetch unit but they have high correlation with stalls and traps in other components, *i.e. commit.NonSpecStalls*, *lsq.thread0.rescheduledLoads* and *dcache.blocked:no_mshrs*, constructing a perfect set for replicated detectors as explained in Section IV. This combination particularly enabled us to classify orthogonal speculative attack techniques [46] (*e.g.*, speculative read in spectre vs. page fault trap in Meltdown) with perceptron in a linearly separable space.

We believe that the interpretability of our features provides confidence that they are strong discriminators that are difficult to evade. It also provides confidence that the large number of features across the whole pipeline capture different aspects of attack signatures, supporting generalization across different attacks, provided they manifest in one or more of the components of the feature space.

VIII. Related Work

We discuss the related work in two categories: (1) Detection-based solutions which attempt to monitor behavior and identify attacks, and (2) other defenses against microarchitectural attacks. We focus primarily on solutions that are rooted in hardware and do not discuss software-only solutions.

A. Detection based solutions

Prior work [6], [7], [8], [9], [10], [11], [12] has used hardware performance counters to detect cache-based side-channel attacks. These approaches are based on the assumption that all cache attacks cause unusual cache access patterns. To detect the attacks, they use cache related counters such as L1miss, L3hit, L1D and L1I accesses and others. These counters are highly biased toward the cache access patterns and are unable to detect attacks like Flush+Flush, which do not access the cache. It would be easy for an attacker to evade such approaches since they use only a few simple counters.

FortuneTeller [50] uses Recurrent Neural Networks and hardware performance counters at the OS-level to predict Meltdown, Spectre, RowHammer and ZombieLoad. They tested 36 counters with their selection method. They used 3 counters to profile for anomaly detection, as the desktop processors Intel Core i5 and i7 are limited to 3 programmable counters. The three selected counters are *L1InstMiss*, *L1InstHit*, *LLCMiss*. The overall performance degradation of FortuneTeller is 24.88% for $10\mu s$ sampling rate [50].

Similar to our approach, prior work has explore the idea of using a hardware detector to detect malicious software [16], [18]. Unlike PerSpectron, these classifiers target general malware, which are software attacks with a footprint that should manifest in the committed instructions of the program. In contrast, our goal is to detect microarchitectural attacks that are inherently exploiting hardware. As a result, the signature of these attacks is reflected in the behavior of the processor pipeline overall, and not only in the features present in the committed stage of the pipeline. Thus, we focus on features related directly to these microarchitectural attacks and view our solution to be solving a different problem and looking at different feature spaces. CC-Hunter [51] detects contention-based CPU side channels by detecting unsual contention patterns. In a similar vein, ReplayConfusion [52] records a running application, then replays it on a different cache configuration to detect

[5] This is an interesting observation because flush operations are also essential for Rowhammer attacks [49]. We have not simulated the Rowhammer attack due to the limitations of simulator, but this result is promising with respect to the generalization of PerSpectron to other micro-architectural attacks that share concepts with the attacks we considered in our feature set.

whether contention patterns are sensitive to the configuration of the cache (indicating an attack). GPUGuard applies a similar strategy of detecting microarchitectural attacks using a decision tree classifier and then invoking defenses. Sadly, it is limited to the context of GPUs, where these attacks are still limited [53].

While some studies have explored some high level features such as cache hits/misses and branch missprediction, they do not consider individualized features in the pipeline stage. PerSpectron's feature space is much more detailed because signature of microarchitectural attacks is more complex and different than the signature of malware.

B. Other Defenses

A number of solutions have been proposed to protect against micro-architectural attacks including side- and covert-channels on caches, as well as transient execution attacks such as Spectre and Meltdown.

1) Defenses against Cache-based Side channels

In this section, we review proposed work on defending against side channel attacks on shared cache hierarchies. NoMo [54] proposes dynamic cache partitioning; it controls the number of ways for each application to interfere with the attacker's ability to prime the cache. CATalyst [55] is another partitioning technique that partitions the cache into security sensitive and a non-secure regions. RPCache [56] and CEASER [57] try to prevent the attacker from evicting target lines from caches by randomizing the mapping of addresses to cache sets. TimeWarp [58] and FuzzyTime [59] add noise to the system clock to prevent attacks that measure cache access latency and execution time. RIC [60] exploits the inclusion property of the LLC to prevent LLC side-channel attacks. This approach prevents the attacker from replacing the victim's data from the local core caches, thus preventing the leakage of secret data.

2) Defenses against transient execution attacks

To defend against SpectreV1, Intel and AMD proposed to use serializing instructions [61], [62] near the branch, eliminating the performance gain of speculation execution. A low overhead hardware solution to mitigate SpectreV1 attacks is Context Sensitive Fencing [41]. In this approach, serializing instructions are injected dynamically to the target of the branches based on the run time conditions or taint analysis engine.

To mitigate Spectre-BTB and SpectreRSB attacks, Intel and AMD [61], [63] include new instructions (i.e, IBRS) in that control speculation around indirect branches. Experiments show that these instructions impose high overhead. Retpoline [64] is a software mitigation technique against a Spectre-BTB attack that converts indirect branches to a sequence of direct call and return and uses an infinite loop or a serializing instruction to prevent the CPU from speculating on the target of an indirect branches.

Several hardware-based techniques are proposed to mitigate different types of speculative execution attacks. DAWG [65] propose partitioning the cache at the cache way granularity to provide isolation between protection domains, which can prevent leakage through cache-based side channel attacks. SafeSpec [39], InvisiSpec [40] and CleanupSpec [66] hide transient microarchitectural side effects from the committed state of the processor, unless they are committed. NDA [67] and SpecShield [68] prevent the propagation of potential secrets to the speculative instructions that may leak the secrets by forming a side channel. STT [69] propose using taint tracking to prevent the forwarding of the sensitive data to speculative instructions.

SpecCFI [70] proposed to use hardware-level control-flow integrity to reduce the range of possible indirect call targets to a small set, and then protecting this small set of call sites using the known lfence-based technique to mitigate all 8 variants of Spectre. In contrast to these efforts, PersPectron uses a different approach of detecting the footprint of these microarchitectural attacks, enabling rapid reaction to them, rather than on rearchitecting the processor to interfere with these attacks.

We should note that several of the detection and defense mechanisms discussed either impose high performance overhead to the system and/or only mitigate a specific variant of the attacks. We believe that PerSpectron offers an inexpensive, always-on solution that can provide wide coverage against microarchitectural attacks. We also believe that there are interesting possibilities when combining PerSpectron with other defenses, for example to raise the alarm to activate other defenses to avoid their overhead unless an attack is suspected.

IX. Conclusion and Future Work

This paper proposes a detection mechanism for a broad range of micro-architecture side-channel attacks in hardware. Previous approaches achieve low accuracy in detecting unseen variations. Software-only methods are also prone to bandwidth evasion. PerSpectron design uses two techniques from Vision (in ML); *replicated detectors* and *invariant feature selection*, leveraging a large set of features available in hardware from all components of the processor including ports, buffers, buses, and uncore.

PerSpectron mapped a non-linearly separable problem to linear, which was then separable by perceptron and is implementable in hardware. The result is competitive with a more complex deepNN that is not easily implementable in hardware. The use of machine learning gives an opportunity to cover other attacks in the future by expanding the feature set and updating the perceptron weights.

Many features used in PerSpectron are invariant to the nature of the attack and are difficult to evade: for example, speculation attacks exploit mispredictions or exceptions. The large number of features potentially make prior evasion attacks prohibitively expensive [19]. In the future, we hope to evaluate and, if necessary improve the security of PerSpectron against adversarial ML attacks, for example by exploring approaches that randomize the feature subset being used over time similar to the RHMD defense [19].

Acknowledgment

We thank Jeffrey N. Collins for his help and comments during the drafting of this paper. This research was supported by National Science Foundation grants CNS-1938064, CCF-1912617, and generous gifts from Intel Corporation.

References

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

[2] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *S&P*, May 2019.

[3] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking data on meltdown-resistant cpus," ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 769–784. [Online]. Available: https://doi.org/10.1145/3319535.3363219

[4] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *41th IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[5] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "CacheOut: Leaking data on Intel CPUs via cache evictions," https://cacheoutattack.com/, 2020.

[6] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Applied Soft Computing*, vol. 49, pp. 1162–1174, 2016.

[7] T. Zhang, Y. Zhang, and R. B. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," September 2016 2016.

[8] F. Herath. These are not your grand daddy's cpu performance counters - cpu hardware performance counters for security. [Online]. Available: https://www.blackhat.com/docs/us-15/materials/us-15-Herath-These-Are-Not-Your-Grand-Daddys-CPU-Performance-Counters-CPU-Hardware-Performance-Counters-For-Security.pdf

[9] M. Payer, "Hexpads: a platform to detect stealth attacks," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2016, pp. 138–154.

[10] X. Wang, C. Konstantinou, M. Maniatakos, and R. Karri, "Confirm: Detecting firmware modifications in embedded systems using hardware performance counters," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2015, pp. 544–551.

[11] Y. Zhang and M. K. Reiter, "D&#252;ppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud," in *Proceedings of the 2013 ACM SIGSAC conference on Computer &#38; communications security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 827–838. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516741

[12] M. Mushtaq, A. Akram, M. K. Bhatti, V. Lapotre, and G. Gogniat, "Cache-Based Side-Channel Intrusion Detection using Hardware Performance Counters," in *CryptArchi 2018 - 16th International Workshops on Cryptographic architectures embedded in logic devices*, Lorient, France, June 2018. [Online]. Available: https://hal.archives-ouvertes.fr/cel-01824512

[13] J. Wampler, I. Martiny, and E. Wustrow, "Exspectre: Hiding malware in speculative execution." in *Proc. NDSS*, 2019.

[14] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre attacks: Stealing intel secrets from sgx enclaves via speculative execution," *arXiv preprint arXiv:1802.09085*, 2018.

[15] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2421–2434.

[16] M. Ozsoy, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Malware-aware processors: A framework for efficient online malware detection," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 651–661.

[17] K. N. Khasawneh, M. Ozsoy, C. Donovick, N. Abu-Ghazaleh, and D. Ponomarev, "Ensemble learning for low-level hardware-supported malware detection," in *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, ser. RAID 2015. Berlin, Heidelberg: Springer-Verlag, 2015, p. 3–25. [Online]. Available: https://doi.org/10.1007/978-3-319-26362-5_1

[18] M. Kazdagli, V. J. Reddi, and M. Tiwari, "Quantifying and improving the efficiency of hardware-based mobile malware detectors," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[19] K. N. Khasawneh, N. Abu-Ghazaleh, D. Ponomarev, and L. Yu, "Rhmd: Evasion-resilient hardware malware detectors," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 315–327. [Online]. Available: https://doi.org/10.1145/3123939.3123972

[20] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 605–622.

[21] Y. Yarom and K. Falkner, "Flush+ reload: a high resolution, low noise, l3 cache side-channel attack," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 719–732.

[22] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds. Cham: Springer International Publishing, 2016, pp. 279–299.

[23] A. Harris, S. Wei, P. Sahu, P. Kumar, T. Austin, and M. Tiwari, "Cyclone: Detecting contention-based cache information leaks through cyclic interference," 10 2019, pp. 57–72.

[24] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *IEEE Symposium on Security and Privacy (Oakland)*, 2019.

[25] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *USENIX Security Symposium (Security)*, 2018.

[26] E. Koruyeh, K. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2018.

[27] S. J. Tarsa, R. B. R. Chowdhury, J. Sebot, G. Chinya, J. Gaur, K. Sankaranarayanan, C.-K. Lin, R. Chappell, R. Singhal, and H. Wang, "Post-silicon cpu adaptation made practical using machine learning," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 14–26. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322267

[28] M. Shah, R. Golla, G. Grohoski, P. Jordan, J. Barreh, J. Brooks, M. Greenberg, G. Levinsky, M. Luttrell, C. Olson, Z. Samoail, M. Smittle, and T. Ziaja, "Sparc t4: A dynamically threaded server-on-a-chip," *IEEE Micro*, vol. 32, no. 2, pp. 8–19, 2012.

[29] B. Burgess, "Samsung's exynos-m1 cpu," in *Hot Chips: A Symposium on High Performance Chips*, August 2016.

[30] J. Rupley, "Samsung's exynos-m3 cpu," in *Hot Chips: A Symposium on High Performance Chips*, August 2018.

[31] F. Rosenblatt, "Perceptron simulation experiments," *Proceedings of the IRE*, vol. 48, no. 3, pp. 301–309, March 1960.

[32] G. E. Hinton and K. J. Lang, "Shape recognition and illusory conjunctions." in *IJCAI*, 1985, pp. 252–259.

[33] G. E. Hinton, S. Sabour, and N. Frosst, "Matrix capsules with EM routing," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=HJWLfGWRb

[34] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio, "Object recognition with gradient-based learning," in *Shape, contour and grouping in computer vision*. Springer, 1999, pp. 319–345.

[35] Y. LeCun, Y. Bengio *et al.*, "Convolutional networks for images, speech, and time series," *The handbook of brain theory and neural networks*, vol. 3361, no. 10, p. 1995, 1995.

[36] S. Ullman and E. Bart, "Recognition invariance obtained by extended and invariant features," *Neural Networks*, vol. 17, no. 5-6, pp. 833–848, 2004.

[37] D. A. Jimenez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, Jan 2001, pp. 197–206.

[38] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2018, pp. 775–787.

[39] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Safespec: Banishing the spectre of a meltdown

with leakage-free speculation," in *Design Automation Conference (DAC)*, 2019.

[40] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[41] M. Taram, A. Venkat, and D. Tullsen, "Context-sensitive fencing: Securing speculative execution via microcode customization," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[42] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, August 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[43] S. Nissen, "Implementation of a fast artificial neural network library (fann)," Department of Computer Science University of Copenhagen (DIKU), Tech. Rep., 2003, http://fann.sf.net.

[44] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, September 2006. [Online]. Available: http://doi.acm.org/10.1145/1186736.1186737

[45] "Spectre mitigations in microsoft's c/c++ compiler," https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html, accessed: 2019-09-30.

[46] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1–19.

[47] C. Li and J. Gaudiot, "Challenges in detecting an "evasive spectre"," *IEEE Computer Architecture Letters*, no. 01, pp. 18–21, feb 5555.

[48] J. R. Quinlan, "Induction of decision trees," *MACH. LEARN*, vol. 1, pp. 81–106, 1986.

[49] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 361–372.

[50] B. Gulmezoglu, A. Moghimi, T. Eisenbarth, and B. Sunar, "Fortuneteller: Predicting microarchitectural attacks via unsupervised deep learning," 2019.

[51] J. Chen and G. Venkataramani, "Cc-hunter: Uncovering covert timing channels on shared processor hardware," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 216–228.

[52] M. Yan, Y. Shalabi, and J. Torrellas, "Replayconfusion: detecting cache-based covert channel attacks using record and replay," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–14.

[53] Q. Xu, H. Naghibijouybari, S. Wang, N. Abu-Ghazaleh, and M. Annavaram, "Gpuguard: mitigating contention based side and covert channel attacks on gpus," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 497–509.

[54] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, pp. 1–21, 2012.

[55] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *2016 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 2016, pp. 406–418.

[56] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 34th annual international symposium on Computer architecture*, 2007, pp. 494–505.

[57] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 775–787.

[58] R. Martin, J. Demme, and S. Sethumadhavan, "Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2012, pp. 118–129.

[59] W.-M. Hu, "Reducing timing channels with fuzzy time," *Journal of computer security*, vol. 1, no. 3-4, pp. 233–254, 1992.

[60] M. Kayaalp, K. N. Khasawneh, H. A. Esfeden, J. Elwell, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, "Ric: Relaxed inclusion caches for mitigating llc side-channel attacks," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2017, pp. 1–6.

[61] I. ADVANCED MICRO DEVICES, "Software techniques for managing speculation on amd processors," https://developer.amd.com/wp-content/resources/90343-B_SoftwareTechniquesforManagingSpeculation_WP_7-18Update_FNL.pdf, 2018.

[62] ARM, "Cache speculative side-channels," https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.

[63] Intel, "Intel analysis of speculative execution side channels," https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf, 2018.

[64] P. Turner, "Retpoline: a software construct for preventing branch-target-injection," https://support.google.com/faqs/answer/7625886, 2018.

[65] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 974–987.

[66] G. Saileshwar and M. K. Qureshi, "Cleanupspec: An" undo" approach to safe speculation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 73–86.

[67] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "Nda: Preventing speculative execution attacks at their source," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 572–586.

[68] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, "Specshield: Shielding speculative data from microarchitectural covert channels," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2019, pp. 151–164.

[69] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative taint tracking (stt) a comprehensive protection for speculatively accessed data," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 954–968.

[70] E. M. Koruyeh, S. H. A. Shirazi, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Speccfi: Mitigating spectre attacks using cfi informed speculation," 2019.