EVAX: Towards a Practical, Pro-active & Adaptive Architecture for High Performance & Security

Samira Mirbagher Ajorpaz
North Carolina State University, USA
smirbag@ncsu.edu

Daniel Moghimi
University of California San Diego, USA
danielm@ucsd.edu

Jeffrey Neal Collins
Independent Author, USA
collinsje91@gmail.com

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

Nael Abu-Ghazaleh
University of California Riverside, USA
naelag@ucr.edu

Dean Tullsen
University of California San Diego, USA
tullsen@cs.ucsd.edu

***Abstract*—This paper provides an end-to-end solution to defend against known microarchitectural attacks such as speculative execution attacks, fault-injection attacks, covert and side channel attacks, and unknown or evasive versions of these attacks. Current defenses are attack specific and can have unacceptably high performance overhead. We propose an approach that reduces the overhead of state-of-art defenses by over 95%, by applying defenses only when attacks are detected. Many current proposed mitigations are not practical for deployment; for example, InvisiSpec has 27% overhead and Fencing has 74% overhead while protecting against only Spectre attacks. Other mitigations carry similar performance penalties. We reduce the overhead for InvisiSpec to 1.26% and for Fencing to 3.45% offering *performance and security* for not only spectre attacks but other known transient attacks as well, including the dangerous class of LVI and Rowhammer attacks, as well as covering a large set of future evasive and zero-day attacks.**

Critical to our approach is an accurate detector that is not fooled by evasive attacks and that can generalize to novel zero-day attacks. We use a novel Generative framework, Evasion Vaccination (EVAX) for training ML models and engineering new security-centric performance counters. EVAX significantly increases sensitivity to detect and classify attacks in time for mitigation to be deployed with low false positives (4 FPs in every 1M instructions in our experiments). Such performance enables efficient and timely mitigations, enabling the processor to automatically switch between performance and security as needed.

***Keywords*-Hardware Security, Side channel, Generative Adversarial Networks, Automatic Attack Sample Generation, Adversarial Machine Learning Attacks, Automated Hardware Performance Counter Engineering, Microarchitectural Attack Detection, Linearized Neural Network, ML Interpretability, Zero Day Attack Defense.**

I. INTRODUCTION

The cost of Global Cybercrime is growing at a rate of 15% per year. The average cost of a data breach in 2020 was $3.86 million with malicious Attacks being responsible for 52% of data breaches [3]. More than 100 Zettabytes of data is distributed across clouds [3]. In 2018 with the disclosure of *transient* (e.g., Meltdown [59] and Spectre [54]) attacks, we learned that most processors currently in use are insecure.

Microarchitectural attacks can be extremely dangerous. For example, transient attacks can compromise all secret data on a system: they can leak arbitrary memory locations from kernel or victim memory. Other attacks such as RowHammer go beyond data confidentiality and break the integrity of programs by inducing controlled memory errors in DRAM cells [50] [38]. Most microarchitectural attacks can not be fully mitigated in software. For example, studies have shown that all currently deployed Rowhammer mitigations are still vulnerable to attack [31] [21].

Microarchitectural attacks continue to emerge at a breath-taking pace. Even worse, newly developed automated attack generation tools can discover new vulnerabilities, challenging existing defenses [34] [66] [30] [89]. Many of these generated attacks have no mitigation [89], *e.g.* FlushConflict breaks even the newest Intel Ice Lake and Comet Lake micro-architectures which are hardened against Spectre and Meltdown.

Transient attacks, to pick an example class of microarchitectural attacks, target the foundations of hardware optimization. Mitigating them fully leads to a significant trade off between performance and security; e.g., 21% - 72% performance overhead for the state-of-the-art HW mitigation for Spectre-type attacks (InvisiSpec) and 74% - 204% for Fencing [90]. The overhead for mitigating LVI attack is over 900% [87]. The number of known security vulnerabilities caused by speculative execution has increased sharply in recent years [15]. New variants of these *transient execution attacks* undermine system security boundaries [14], [15], [66], [79], [86], [87], [88]. Other variations exploit other microarchitectural optimizations and their shared hardware components [7], [8], [10], [26], [27], [35], [44], [46], [60], [65], [70], [71], [74], [75], [78]. Defenses proposed to date [22], [52], [56], [77], [90] only target specific attack variants or protect individual microarchitectural components. Complete protection requires multiple mitigations that are active concurrently, imposing additive performance overhead

on the processor. This paper proposes an ML-assisted architecture that can detect attacks and apply targeted mitigations only as needed significantly reducing their performance overhead.

Currently, the average time to identify a data breach is 280 days [3]. The average cost savings of containing a breach in less than 200 days vs. more than 200 days is $1.12 million [3]. The propose detection architecture will be part of a proactive and an adaptive end-to-end solution for security closing the gap in detection and eliminating the significant cost of reactive security response.

Detection of microarchitectural attacks poses several challenges. Attacks can have short signatures. For example, transient attacks trigger transient execution (using branch prediction or out-of-order execution). During the transient window, they complete their attack and leak secret data to the attacker. Access to a high number of hardware performance counters (HPCs) monitoring transient events (uncommitted instructions) is essential. Lightning fast Sampling and classification is essential for mitigations to be applied prior to leakage. Finally, detection can be improved by engineering new security-centric HPCs to detect new attacks. Software detectors fail because the number of events that can be monitored from software without multiplexing counters is limited (up to 4). Computation is on the critical path, limiting the sampling frequency and leaving software detectors vulnerable to bandwidth evasion [58], [62]. On the other hand, hardware detectors are promising due to significantly higher sampling frequency and computations being conducted outside the critical path [62]. We show that prior work fails to detect evasive attacks and many novel attacks, due to insufficient training data and dependence on current HPCs which were designed for tuning performance. We need to develop new security-centric HPCs that expose features that promote accurate and rapid detection.

Perceptron-based detectors have been shown to provide classification outputs within the transient window [62] without computational delay. PerSpectron was able to negate bandwidth evasion due to its high sampling frequency on the order of 100 nsec to 1μ-sec [62]. However, PerSpectron is limited in detecting zero-day attacks and is susceptible to evasive attacks generated by new *fuzzing technologies* and AML attacks [16], [30], [34], [48], [66], [89].

Attackers exploit pockets of adversarial space between the data fitted by the learning agent and the theoretical distribution space to fool ML (see Figure 1) . A large class of vulnerabilities arise from the fundamental problems of imperfect learning: (1) Ineffectiveness of classifier models to contribute to the adversarial space; and (2) Flawed adversarial space identification as a result of an imperfect training data set [19]. We need automatic attack sample generation tools to provide a broad range of possible permutations/variants of microarchitectural attacks to train our classifier.

Prior works have proposed automatic attack code generation techniques that attempt to create new attacks [30], [34], [66], [89]. These transformations applied at the source or binary level, include insertion, deletion, rewriting, and other strategies, using a fuzzing-based approach. Fuzzing tools are typically used to permute a single type of microarchitectural attacks, for example, Meltdown or Rowhammer-type attacks. Attacks generated by these tools do not comprehensively explore the evasion space, missing a slew of potentialy dangerous attacks or evasion strategies. This is because code transformations [30], [34], [66], [89] do not automatically translate to the complex feature space of the classifier.

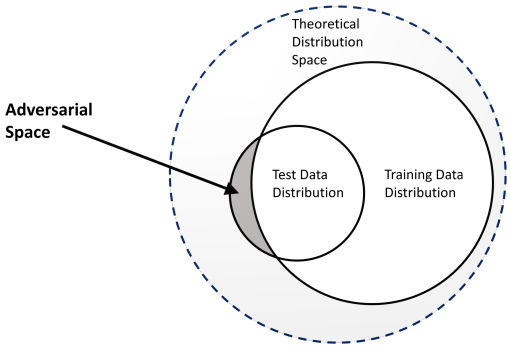


Figure 1. Root Cause of ML Vulnerability

Our solution— Disabling Adversarial ML Attacks by Increased Classification Margin beyond the Leakage Window. Evasive attacks exploit the minimum margins from samples to classification boundaries (see figure 2). Recognizing that microarchitectural attacks are timing sensitive– *they rely on bringing the microarchitecture to specific state during a short transient window and transferring the secret before activity of other processes and kernel destroys its footprints*, our solution is to push the classification boundaries in the worst *adversarial* directions until further attempts to evade disables the attack. We do that by retraining the classifier on a very large set of adversarial microarchitectural samples automatically generated through a comprehensive search using Generative Adversarial Networks (GANs) [33].

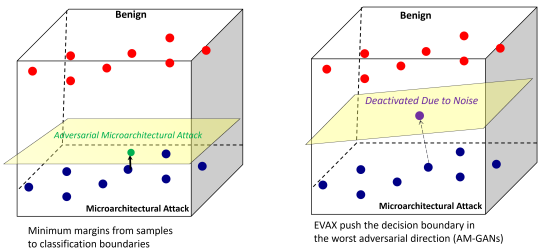


Figure 2. Large classification margin toward adversarial directions, to disable timing-sensitive attacks.

Microarchitectural attackers can only leak data for a limited time interval before the CPU eventually catches up, detects the fault, or aborts transient execution. This implies that there

is only a limited transient window in which the victim can leak data e.g., in LVI victim inadvertently computes on the poisoned load values, and all required gadget instructions need to complete within this window to transmit secrets. The transient window is ultimately bounded by the size of the processor's reorder buffer (ROB). For example, our experiments show adversarial ML efforts in systems with small ROB fail to evade our detector. ROB size reflects the boundary of transient window and is a property of system that can be learned through self-supervised learning (e.g, Generative Modeling).

We introduce the Evasive Vaccination framework (EVAX) which generates adversarial examples that push decision boundaries in adversarial directions automatically. To identify the adversarial directions we leverage Generative Adversarial Networks (GANs). GAN contains two different networks; a Generator and a Discriminator which play an adversarial game against each other. In GAN, two models have opposite agendas: discrimination and generation. During each step of the training, the output of each model improves the other. Because the training of the generator in GAN is toward gradient descent (adversarial direction), retraining with the generated examples that fool the discriminator, enlarges the margins of decision boundaries to the point that code transformations needed for evasion disable the attack. EVAX can fine-tune the generation of new microarchitectural attack samples to virtually *infinite* granularity.

In line with the ethical guidelines of Google's SafeSide project [2], EVAX aims to create attack samples that are useful for training and analysis but cannot be weaponized by attackers. EVAX Generated samples are vectors of microarchitectural counter values. With no direct mapping between code transformations and feature values an attacker cannot reverse engineer attack code from our attack samples. In contrast, the traditional automatic attack generation frameworks [30], [34], [66], [89] can provide attackers with powerful tools to compromise even a hardened system.

We also showed that when the counters representing the inner-nodes of a Generator deep NN in GANs are combined and moved to the input-layer, the accuracy of the linear NN (fast and HW friendly) becomes equivalent to a deeper NN. Interestingly, we concurrently observed and showed empirically that the same property proved by the recent GoogleMind study [57] exists in the systems ML field as well; it proves that for wide NN the learning dynamics simplify considerably and that in high width they are governed by a linear model. Our finding gives an empirical equivalent of their mathematically proven theorem while providing a practical method to widen the NN automatically (see Sec. VI of the paper). GoogleMind study, while proving it theoretically, has not proposed a method/tool to automatically widen the NN and make it linear. Our paper proposes an effective and automated strategy to select highly correlated internal signals of DNN and combine them into one as an input node, to design fast and HW-friendly ML for solving complicated problems. This simplification of NN can also make DNN more interpretable. This equivalence can actually be represented mathematically but it is beyond the scope of this paper.

This paper makes the following important contributions:

- Selectively triggering defenses, which reduces the overhead of state-of-the-art defenses such as InvisiSpec (from 27% to 1.26%.) and Fencing (74% to 3.46%), making them practical for deployment.
- It demonstrates a robust end-to-end defense system for 19 categories of microarchitectural attacks. Many have no mitigation and have not been previously detected.
- It presents an automatic method for engineering new security-centric HPCs.
- It provides a generative framework for automatically generating microarchitectural samples of attacks of a given *type* covering a broad range of microarchitectural attacks.
- It presents a novel training method that disables powerful Adversarial ML Attack and defends against automatically generated attacks from fuzzing tools (23.5% improvement).
- Shows effectiveness for Zero-Day attack detection with a 10.8% accuracy improvement over the state-of-the-art.
- Proposes a new metric to verify, interpret, and visualize the state of microarcitectural attacks as well as correctness and quality of the attacks generated.

II. BACKGROUND

In this work we focus on detection of Microarchitectural *side channel* attacks and memory attacks such as RowHammer.

Side Channel Attacks. Non-Transient side-channel attacks include flush-based attacks such as Flush+Fush [39], Flush+Reload [91]. Conflict-based attacks such as Prime+Probe [61] and Evict+Time [70], as well as attacks on other microarchitectural structures such as the branch predictor [7], [25]. Transient attacks consist of a leakage gadget and a transfer gadget. The leakage gadget can be either fault-based such as Meltdown [59], Zombieload [79], LVI [87] and Fallout [14], or Speculation based such as, SpectrePHT [54], SpectreBTB(or SpectreV2) [15], [54], SpectreRSB [55] and NETSpectre [80]. Transient attacks use techniques from the non-transient attacks as their transfer gadget. Transient attacks have the capacity to leak *arbitrary memory* from a victim process and can even dump the whole kernel or victim memory. Transient attacks such as Meltdown and Spectre leverage Out of Order Execution (OOO) and Speculative Execution to leak data during the transient speculation window, breaking memory isolation.

Example of exploitable Performance Optimization properties: The Branch Predictor predicts which direction and

target of a branch should be taken. The Front-end fetches the instrucions and decodes them to micro-operatons (uOP). Instructions are speculativly executed on the predicted path, enabling run ahead to fill up the idle execution units and hide memory access latency. If the branch prediction was incorrect, the ROB allows rollback to the correct state. The Scheduler allows the CPU to use data values as they become available and renames registers to solve RAW, WAR and WAW hazards. The ROB keeps track of all uOps with respect to the instruction stream. We can load caches during speculative execution. The Address Generation Unit (AGU) load and store units are directly connected to the memory subsystem to process its requests. *Inaccessible data is provided before interrupt flags and during wrong paths.*

Transient Attack Examples. In a Meltdown attack the attacker: 1) flushes the memory space in the cache; 2) performs a Syscall/prefetch to have the kernel address loaded in L1 cache; 3) Loads the secret to a register R; 4) fills the ROB with dependent instructions which use a different execution unit; 5) Issues a transient load to an address, indexed by the content of Register; and finally 6) Measures the time of Reload. Main phases in Spectre-PHT are: 1) Attacker Flushes the accessible cache lines; 2) Mistrains the branch predictor to make a wrong prediction; 3) Performs a transient access to inaccessible data; and 4) Times the access to the monitored line. Even if the processor prevents speculative execution of instructions in the user process access the kernel memory, the Spectre attack still works.

Memory Attacks - Rowhammer compromises data integrity rather than confidentiality. DRAM cells leak charge over time and must be periodically refreshed to prevent data corruption. Attackers can exploit the shared DRAM buffer to not only construct a side channel (DRAMA [74]) but also amplify the data corruption and cause bit flips in targeted memory pages (Rowhammer [50]), compromising OS and browser security [29], [37], [38]. Recent mitigations are shown to be not fully effective against Rowhammer [21], [31]

LVI - In LVI [87], first the adversary injects its data. The victim loads this data and stores it in a microarchitectural buffer. The load whose translation takes long time to settle and thus the stored adversary data gets forwarded speculatively. When the recovery happens, victim's secret is inferrable by accessing microarchitectural buffers. Flushing shared buffers, software patches, microcode mitigations, Spectre hardware mitigations and constant time programming [17] all fail to mitigate LVI. The only current mitigation fences on every load which is reported to have a 900% overhead [87].

Why Transient Attacks Can be Detected in Microarchitectural Layer. We observe that each attack phase leads to misuse or under utilization of processor resources which is a direct side effect of bringing the microarchitecure into a state of leakage and recovery, making these invariant features for leakage. The processor has HPC's built in for debugging, verification and optimization purposes which monitor events occurring in processor resources. Affected resources include the execution engine, branch predictor, TLBs, caches, MMU caches, buses, buffers, cache coherency, power and more. Flags of activities in the system propagate up and down the pipeline (e.g., stalls or transient instructions travel down the pipeline) and can be detected by monitoring different events affecting these resources.

III. Related Works

Current Mitigations & their Overhead. InvisiSpec [90] adds a separate speculative buffer for placing the speculative load data until it is safe to be visible. InvisiSpec suffers from high overhead (27%). Fencing speculative loads to defend against Spectre attacks, have over 70% overhead. Utilizing microcode-level optimizations to surgically inject specialized fences [84] mitigates some variants of Spectre attack but does not mitigate the dangerous MDS and LVI attacks. Fencing all loads, suffers from high performance overhead of up to (900%) [87]. In this work we reduce these overheads drastically, by 95% or more, while providing full defenses for even larger spectrum of attacks than individual patches. With reduced overhead, these mitigations become feasible, providing security in an elastic way to the microarchitecture.

SW Detectors. Malware detectors from SW layer are unable to detect microarchitectural attacks because microarchitectural attack signature are fundamentally different than Malware [62]. Prior works used features related to committed state such as instruction mixes or memory access distribution. Detection of transient attacks requires access transient features. SW can only monitor 4 HPCs at a time without multiplexing counters. This provides opportunity for attacker to evade detection. SW detectors have low sampling frequency (100ms vs ns in HW) making them vulnerable to Bandwidth evasion. [58] Sampling and computation happen on the critical path leading to high overhead. And thus, prior work in SW detection fails to detect stealthy transient attacks. [58], [62]. Therefore, in this work we focus on HW-based detection of transient attacks.

Static/Non-ML detections. Prior work in HW based detection has followed two general approaches: (1) Static-signature-based detection similar to Cyclone [40] and (2) anomaly detection using ML similar to [62]. Static-signature-based detection, detects static properties for example Cyclone [40] detects contention-based cache attacks by tracking the cyclic interference property in hardware. Cyclone can only detect two types of microarchitectural cache attacks (Flush+Reload and Prime+Probe).

Cyclone detects after leakage while ML-based approaches detect attacks prior to leakage. [62] ML-based approach can generalize to new attacks because it is focused on *system properties* looking for activities that are anomalous with respect to normal system behaviour unlike static-signature-based detection.

PerSpectron. Recent work has demonstrated success at detecting microarchitectural attacks in hardware with low performance overhead [62]. While software detectors are vulnerable to bandwidth evasion techniques, detection in HW is resilient to such evasions [62]. Due to their efficacy, manufacturers are beginning to support Hardware Malware Detectors (HMDs) to mitigate threats such as Ransomware [6] and malicious mobile apps [5]. PerSpectron utilizes a single layer perceptron in hardware to monitor 106 features spanned over the processor i.e., replicated detectors in the processor to detect seen Spectre-PHT, Spectre-RSB, Meltdown, Prime+Probe, Flush+Reload, Flush+Flush attacks and CacheOut, Spectre-BTB out of sample.

Perceptron is fast and HW friendly. But can only learn linear separable functions. This problem exacerbate when trying to learn many novel categories/classes of attacks e.g., new LVI type attacks, with one structure. This work extends the number of novel attacks from 7 to 19 while increasing the overall detection accuracy. PerSpectron is also susceptible to AML attacks. In this work we propose an adversarial hardening methods that makes PerSpectron resilient to adversarial attacks. ML based detectors [62] can also be tuned to increase sensitivity to detect several microarchitectural attacks. One major issue with PerSpectron is high false positives and vulnerability to new Evasive technologies.

ML Related Studies. Augmenting the training dataset has been used to increase the accuracy of a model. [9] Shows that SNN can achieve same accuracy as DNN when trained with data generated by DNN using model compression. [12] uses GAN data augmentation for robotic grasping. [24] Conditional RGAN used to generate synthetic medical records for training physicians. However [32] suggests that linear models lack the capacity to resist adversarial perturbation. We show that not to be the case. [64] shows that virtual adversarial training efficiently moves decision boundaries in the adversarial direction. [67] demonstrated that iterative linearization of the classifier can generate minimal perturbations that are sufficient to change classification labels. [69] suggest SSL when there are no high-quality labeled datasets from similar domains to use for fine-tuning. [72] discussed a number of methods for injecting adversarial examples during training to improve the generalization of a machine learning model. [83] Google improved image recognition with deep CNN. [72] used GAN to improve student/teacher network learning. Apple, Google and other researchers have used virtual adversarial training to improve ML models [49], [69].

IV. Threat Model

We assume the following: (1) A machine with a Hardware Malware Detector (HMD) trained to detect microarchitectural attacks similar to PerSpectron [62]. (2) An attacker with access to a similar detector, capable of using it to craft and test evasive attacks. (3) We assume no specific evasion strategy; *i.e.*, we assume no limits on how the attacker chooses to evade provided the program is legal and the attack is preserved. The attacker could be using arbitrary methods to evade signature-based detectors, including those developed in the malware community [4], [54] or fuzzing based approaches [30], [66], [89].

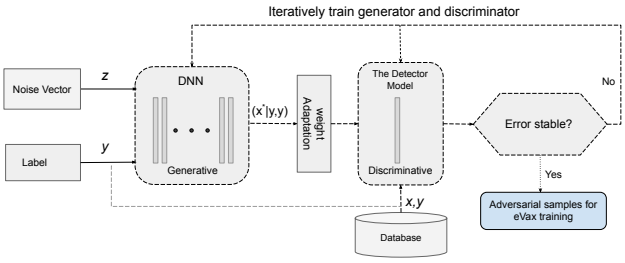


Figure 3. EVAX AM-GAN Training Diagram

V. EVAX

A. Training Generative Modeling Framework

The Goal. The goal of EVAX is to develop a detector that generalizes across a large number of seen and unseen microarchitectural attacks and is also resilient to evasion.

Recall that adversarial algorithms aim to generate a single example that fools the detector for a given input and potential target class; typically, the algorithm determines an effective perturbation pattern that guides the input across the classification boundary to cause it to be misclassified.

We would like to automatically generate attack samples that represent the feature map for these undisclosed or undiscovered variations of attacks. These samples will harden our detector and sharpen its classification boundaries making it resilient to evasion.

Our Solution. To solve this problem, we use a training process with a (GAN)-based game. The idea is that since GANs can generate realistic instances of data that no one has seen, vector representations of these instances can also be used to describe evasive or undisclosed variations of attacks. GANs improve the training process by turning the attack generation problem into a *self supervised* learning problem between two sub-models, a Generator and a Discriminator. The two sub-models are trained together in an *adversarial game*. The Generator learns the domain specifics automatically through playing the game with the Discriminator instead of training on limited data. In a traditional attack detector, the classifier learns from input/output data; therefore, the classification ability is limited to seen data points. But GAN does not learn from an input/output data mapping to tell the model how it should generate an attack. The Discriminator, in contrast to the classifier in conventional detectors, learns how inputs and outputs can be paired in a microarchitectural system,

```
1: procedure VACCINATEHARDWAREDETECTOR(S′)
2:     procedure AM-GAN TRAINING( )
3:         while Generator is improving do
4:             for each training iteration do
5:                 [x, c , t] ← get_sample [S]   ▷ x is a sample   ▷ t ∈ target malicious/safe   ▷ c is attack's type   ▷ S is the training set
6:                 procedure DISCRIMINATOR TRAINING( )   ▷ Train the discriminator to distinguish between real and generated data.
7:                     y← Discriminator (x,c)   ▷ y is 1 for real and matching , 0 for fake
8:                     d_x ← t-y   ▷ d_x is discriminator's error
9:                     z← RandomNoise(145)   ▷ z is a noise vector for 145 features
10:                    x* ← Generator (z,c)   ▷ x* is adversarial/fake example
11:                    y* ← Discriminator (x*)   ▷ y* is 1/0 for real and matching/fake
12:                    d_y* ← y*-y   ▷ d_y* is discriminator's error
13:                    Update(Discriminator , d_y* + d_x , c)   ▷ Update the discriminator's weights seeking to minimize d_x + d_y*
14:                procedure GENERATOR TRAINING( )   ▷ Train the generator to output data that "fools" the discriminator
15:                    z ← RandomNoise(145)   ▷ z is a noise vector for 145 features
16:                    x* ← Generator (z , t , c)   ▷ x* is adversarial/fake example
17:                    y ← Discriminator (x*)   ▷ y is 0/1 for fake, unmatching/real and matching pairs
18:                    d_x* ← t-y   ▷ d_x* is Discriminator's error
19:                    Update(Generator , d_x*)   ▷ Update the generator's weights seeking to maximize d_x*
20:    procedure AUTOMATIC ATTACK GENERATION(c′,t′))
21:        l ← [c′,t′]   ▷ t′ ∈ desire target malicious/safe   ▷ c′ is desired attack's type, 0 if safe
22:        z ← RandomNoise(145)   ▷ z is a noise vector for 145 features
23:        x′ ← Generator(l,z)   ▷ Generate an attack for label l, x′ is the generated attack sample
24:        S′ ← add(x′,c′,t′)   ▷ S′ is the augmented training set
25:        for each training iteration do   ▷ Generate a batch of new attacks
26:            [c′,t′] ← get_label [S]   ▷ Iterate through label list
27:            AutomaticAttackGeneration(c′,t′)   ▷ Generate new attack for type c′ and target t′
```

Figure 4. EVAX Training algorithm.

and it tells the model whether it was correct. Because of this, the Generator ultimately learns a broad range of the possible correct answers

This is an important principle of our design and the key reason for EVAX's high effectiveness.

The Game Setup. Our Discriminator in this game has the architecture of our hardware detector. We start with a single layer perceptron-based classifier similar in architecture to PerSpectron [62]; however, this technique would generalize to any ML-based detector. Our Generator is a deep neural network, which creates an unusual asymmetry between the Generator and Discriminator. We will label this an Asymmetric Model GAN (AM-GAN). As the adversarial game unfolds, the Generator learns each time a sample fools the Discriminator. Instead of recognizing a pattern, the Generator learns to create attack samples essentially from scratch; indeed, the input into the Generator is mainly a (noise) vector of random numbers, the feedback from the Discriminator and the label. Conversely, each time the Discriminator rejects an attack sample, the Generator uses the feedback to improve its strategy. Even though the Generator always generates samples without the knowledge of seen attacks (ground truth), the overall training of our AM-GAN is supervised because real attack samples are used to train the Discriminator. As the Generator gets better at producing realistic attack samples, the Discriminator gets better at telling generated samples from original data. Moreover, as the Discriminator gets better at detecting a certain evasion strategy, the Generator is forced to explore additional strategies, exposing the Discriminator to them, further improving its robustness. Both models continue to improve simultaneously.

Integrating Labels. Our AM-GAN (Figure 5) can also determine the kind of attack sample it generates. In a simple GAN, there is no means to control the attack type; utilizing a conditional GAN (CGAN) allows us to customize properties of samples for each attack class. Our Discriminator learns to accept matching pairs from the seen database while rejecting pairs that are mismatched and pairs that are produced by the Generator. Our final Generator is capable of producing realistic microarchitectural attack samples for each attack type we give a label to. We enter the descriptive features of an attack into our Generator and have it output a range of matching samples, expediting the process of adversarial attack sample generation. In Section V-D we introduce a metric to measure the semantic loss for generated attacks and visually verify this capability of our AM-GAN.

B. Attack Sample Generation

Figure 5 depicts a block diagram of our AM-GAN architecture and it's training process (see Algorithm in Figure 4). In this section we explain the detailed steps of our training algorithm.

Training the Generator: For each training iteration:

- The Generator takes a new random noise vector z and an attack type label *e.g., spectre-RSB-type attack*. The Generator generates an example $x^{\star}$ that strives to be both an adversarial attack (*e.g.*, looks like a safe program) and

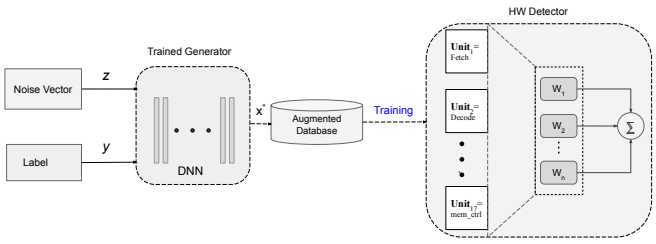


Figure 5. EVAX HW Detector Training Diagram

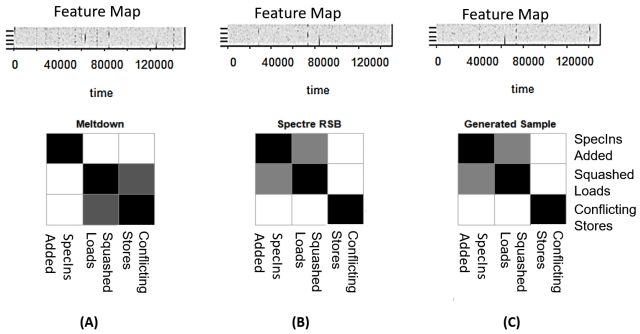


Figure 6. Part of the Gram matrix during leakage phase for two attacks and an attack sample generated by our framework with the label SPECTRE-RSB – the darker color represents larger values. Attacks (B) and (C), similar in type, have similar Gram matrices. Even though the values of the features may be very different, the Gram matrix–a measure of the correlation between pairs of features–is similar.

a convincing match for its label.

- The Discriminator network classifies $x^\star$ into 0/1. One indicates a match with seen sample-labels from the dataset and zero indicates unmatched sample-labels and generated sample-labels.
- The classification error of the Discriminator will be computed and backpropogated to update the Generator weights, seeking to maximize the Discriminator's error.

Training the Discriminator: For each iteration:

- We choose a seen program sample x from the training dataset with its associated label.
- Generator gets a new random noise vector z and label describing the attack type and synthesizes an adversarial example $x^\star$.
- Discriminator receives the seen example x and label, the adversarially generated sample-label $x^\star$ and the label that was used to generate them.
- For both examples the Discriminator outputs a probability indicating whether the input example was seen data from the dataset, matching its label pair or unmatched pairs.
- The classification error of Discriminator is computed and backpropogated to update its weights, seeking to minimize classification error.
- The training ends with our AM-GAN reaching Nash equilibrium, meaning that at this point the Generator produces fake examples indistinguishable from seen attacks in the training data, and the Discriminator can at best randomly guess whether a particular example is a seen attack or a generated one.

C. Training the Hardware Detector

It is important not to confuse our Discriminator with our final Detector. We emphasize that we can not use this Discriminator as the attack detector directly. While they are both classifiers, the Discriminator adapts to answer a different, although related, question: whether a generated attack sample is *fake* or *real* (seen attack or generated). It does not perform well as a Detector whose goal is to classify *suspicious* vs. *safe* activity. After reaching a stable error (it is hard to achieve Nash equilibrium) we perform the process of hardening our actual detector using AM-GAN:

- We feed the trained Generator with each attack type from the training set (conditioning inputs) and collect the generated examples, using them to augment the training database. The examples produced by our Generator are adversarial attack samples that carry the footprints of microarchitectural attacks but are very hard to distinguish from benign programs.
- We retrain our hardware detector with the augmented database. The generated examples which consistently fool the Discriminator are used to train our EVAX, thereby hardening it against such evasive attacks.

D. Interpretability & Quality measure for the Generated Samples

In AM-GAN, just by looking at the two loss functions it is unclear when we've actually finished training, since when one gets better the other one gets a larger loss. In principle, Nash equilibrium gives us a reasonable stopping criteria but in practice reaching equilibrium is difficult. To cover the adversarial space, we would want to start collecting training data from the generator when the generated examples are different enough from the seen data in values but belong to the same class. As explained in Section II, several basic blocks represented as misused or under-utilization are required to be present simultaneously for attacks to build a side channel. So we want to score samples based on the presence and approximate values of low-level microarchitectural states required for successful construction of a channel, leakage, and recovery of each type.

Our solution is based on the observation that while one attack and its evasive version have different distribution of instructions they share commonalities in micro-arhitectural state during different phases of the attack. We want every new attack generated by AM-GANs Generator, conditioned by an attack type (label), to have some common correlation patterns during leakage between features (mutual correlation pattern) with the seen example of the attack category. So we

define a *Gram matrix* that contains the dot product between all possible feature pairs.

Gram Matrix. To numerically measure how often two feature maps are present together, we multiply the values of two vectors in each position and sum the results. If the resulting value is high, the features are highly correlated. Figure 6 shows the Gram matrices for three features for each of the attacks. The ribbons on the top of Figure 6 are the program map. Each pixel shows the value of the microarchitectural features over time. The bottom part of the Figure shows *correlations* between three chosen features.

Microarchitectural Leakage Snapshot (Visualization). For example, to generate the attack (C) in Figure 6 we have fed the model with a *spectre-RSB* condition. We were able to visually check that EVAX is working as expected since the Gram matrices for the three features in attack (B) and (C) match but (A) and (C) mismatch since attack (A) is meltdown and (B) is spectre. While *leakage style* is similar, the feature maps on the other hand for attack (B) and (C) are different – verifying that AM-GAN found another variation of the same attack type matching a different binary.

Interpretability of Features. Visualization of AM-GAN generated samples also helps the designer to identify and interpret the highly correlated features. For example, we can see that the features *Conflicts in Instruction Queue* and *SquashedLoads* fire strongly together in Meltdown but not in Spectre-RSB and the generated attack (C). Conversely *Speculative Instructions Added* often fire strongly together in Spectre-RSB and the generated attack (C). We can visually see that Meltdown-type samples rely on Conflicts in Instruction Queue (OoO execution). We see that squashed loads in Speculative attack samples generated are caused by speculation not exceptions. These examples verify that our AM-GAN framework is trained long enough to represent the continuous target distributions of attacks from which we can sample for training EVAX. This is a powerful tool that can be used for further analysis and tuning of the features of the final product and for developing mitigations of future attacks.

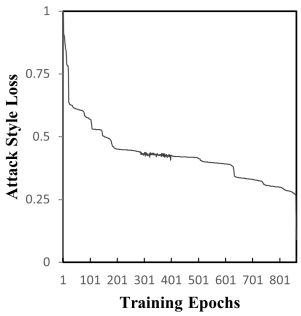


Figure 7. Attacks Style Loss during AM-GAN training.

Attack Quality Measure. To calculate the attack style loss, we calculate the Gram matrix (GM) for a set of features for the base and generated attacks and then compare their similarity using sum of squared errors. Attacks (B) and (C) which are similar in type, will have similar Gram matrices resulting in leakage style loss near zero. Algebraically, the attack leakage style loss (L_{GM}) between the base attack (B) and generated attack (G), can be written as follows:

$$L_{GM}(B,G) = \frac{1}{4\alpha N^2}\sum_{ij}(GM(B)_{ij} - GM(G)_{ij})^2$$

Where N is the number of features and α is a constant.

In Figure 7 we can see that attack footprint style loss (L_{GM}) reduces with each training iteration–the output becomes stylistically closer to the type of the attacks that it is labeled to produce — meaning as the training epoch increases, the *quality* of the generated attacks gradually becomes higher.

We monitor this metric as AM-GAN is being trained and start collecting training data when the absolute value of the (L_{GM}) for our generated examples becomes quite small (0.1 ± 0.006) indicating the sample represents the microarchitectural state of leakage related to the attack type. We visually verified the quality and semantic consistency of one example generated of each attack type in our database before starting to collect samples to augment our dataset.

VI. HW DESIGN

A. Feature Engineering

We want to achieve accuracy comparable to a deep neural network with a Security Friendly (fast classification) & HW Friendly (light) model (shallow/linear neural network). Simple models can provide classification fast enough to detect transient attacks before leakage. However, we also find that such simple models are capable of generating highly accurate predictions.

Solving a multi dimensional problem with a simple architecture. We want to use the same simple (e.g., perceptron) classifier PerSpectron uses to classify many more categories of novel attacks that the previous model fails to learn, such as LVI-type. Our solution is to add Dimension to the input space. Introducing new performance counters and monitoring a large set of microarchitectural features transforms a Non-Linearly Separable Space to Linear. This allows a simpler architecture — no hidden layer is necessary. This is because the capacity of an ML model depends on both input space and the model itself. As illustrated in figure 8 higher dimension enables a simpler classifier. It seems that there is a constant total value of *classification Volume* which can be captured by the dimension of input space multiplied by order of classification function (complexity); if you increase the input dimensions, you can use a simpler classifier.

In order to expand the detection ability of the prior works to more categories of novel attacks, we include more HW performance counters (145 instead of 106) including 12 newly engineered security-centric performance counters. These new engineered features enabled the same linear model trained

for classifying 6 categories of attack to also learn 19 other classes of attacks. Additionally, the new security performance counters are created by an automated method of selection and combining multiple signals originated from the performance counters into one new, highly correlated counter for *security*. Conducting this search by brute-force requires simulation time that is intractable. For example to simply choose three counters to combine out of 1160 one must test 259,476,920 possible combinations as most combination does not correlate to the security anomalies. We automated this selection using AM-GAN which can select any number of useful counters (1,2,3,4,...) without the need of running all combinations. We will explain this method (collapsing independent counters into a smaller set and linearizing the network while maintaining the accuracy) next.

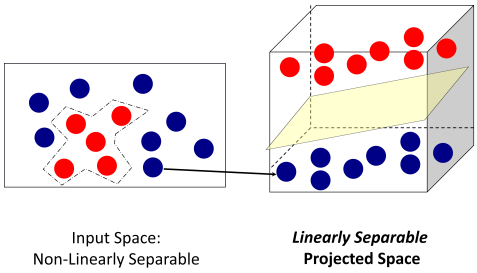


Figure 8. Effect of added dimension (new performance counters for security) on classification.

Automating Performance Counter Engineering with GANs for Security. Performance Counters are designed for tuning performance. Some of these performance counters are correlated with security. Prior work [62] identified (manually) complex performance counters such as Number of Clean Evicts, that was shown to be useful in detection of stealthy cache attacks (see Figure 9).

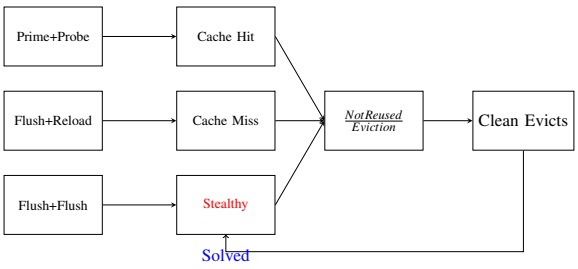


Figure 9. Complex HPCs for detection of stealthy cache attacks.

In this work we use the hidden nodes from our trained AM-GAN Generator to *automatically* engineer new counters for security. The hidden nodes receive the information from the upper layer and processes it. Then, the obtained value is sent to the next layer. The output layer will also process the information from the hidden layer and give the output. The top weights with largest absolute value dominate the instantiation of each layer. We look at hidden nodes that are connected to the first layer. We sort the weights of the hidden layer of the network and select the top 12 nodes connected to the input HPCs. We then define the Boolean AND Logic of connected HPCs to that node as a new HPC specifically engineered for Security. These are different than 133 highly correlated features that we already use (27 more than prior work). See figure 10.

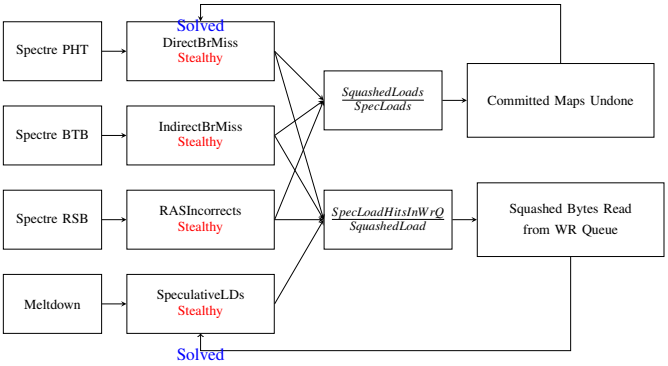


Figure 10. Complex HPCs for detection of stealthy Speculative and Meltdown type attacks.

An example of these engineered features is *SquashedBytesReadFromWRQu*. This feature is equivalent to a node in the third layer of our AM-GAN generator combining the signals from two nodes (1) *Spec Loads Hit in Write Queue* and (2) *number of Squashed Loads* with heavy weights. *Spec Loads Hit in write Queue* (itself a combination of two HPCs) and *Number of Squashed Loads* with heavy weights are already availble HPCs. We then merge these two signals into one to engineer a new signal we call *SquashedBytesReadFromWRQu*, which can easily be implemented in the CPU with minimal logic. We found this new HPC specifically exposes security vulnerabilities in the transient domain. For instance, as shown in Figure 11, we show that this new HPC detects both MDS and LVI attacks.

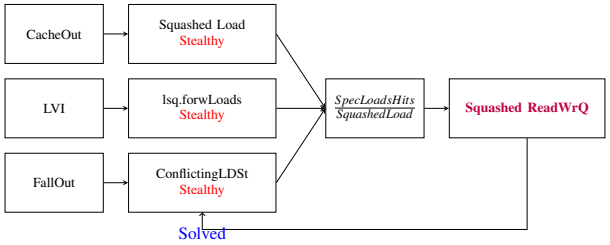


Figure 11. Complex HPCs automatically engineered capable of detecting unseen MDS-type and LVI attacks.

Replicated Feature Detector. We also use the entire space by replicating mutually correlated features. Prior work suggested use of *Replicated detectors* to simplify the problem of detection for perceptron-based detectors. The idea is to not disregard the mutually correlated (redundant) features and instead replicate them in the processor i.e., different stages

of pipeline, buses, etc. They showed that if a feature vector was useful in detecting one target (seen variant), it is likely that a similar feature detector in different positions in the pipeline can detect the evaded information (unseen variant). Replicated feature vectors also allows each patch of program to be represented in several microarchitectural ways–making the trained model resilient to several evasions. Feature replication greatly reduces the number of free parameters to be learned by the model– enabling a simple HW design for fast classification. Replicated features also allows simpler circuit for centralizing the signals as one signal can be captured by another signal down or up the pipeline through replicated feature detector design.

Perform the feature engineering offline Selecting replicated features and engineering new features is done automatically through sub-sampling the hidden layers in the DNN Generator Network. To enable competitive performance with a shallow NN we apply the above steps of feature selection, replication, and feature engineering offline. This eliminates the need for hidden nodes and makes our detector more resilient to evasion. Figure 12 shows the overview of our framework.

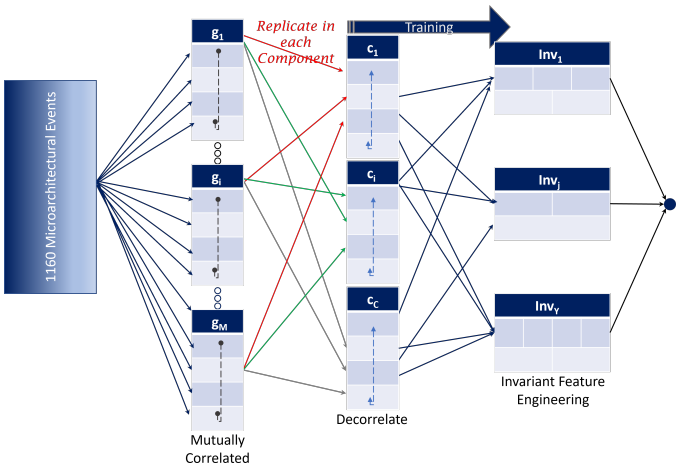


Figure 12. EVAX Feature Selection (offline). Invariant Feature Engineering using GANs.

#	Security HPCs
1	Squashed Bytes AND Bytes Read from WR Queue
2	Committed Maps AND rename.Undone
3	iew.Mem Order Violation AND dtlb.rdMisses
4	lsq.squashedStores AND lsq.forwLoads
5	membus.trans_dist::ReadSharedReq AND lsq.ignoredResponses
6	iq.SquashedNonSpecLD AND dcache.ReadReq_mshr_miss_latency
7	rename.serializingInsts AND iew.ExecSquashedInsts

Table I
SUBSET OF NEW PERFORMANCE COUNTERS FOR SECURITY ENGINEERED BY EVAX.

B. Hardware Overhead

We choose the perceptron neural network [47] because of its speed and small footprint. We also keep our hardware as similar to PerSpectron as possible to facilitate comparison. Our training method can potentially improve the robustness of any ML based detection method. See section VIII-D for the result of training on AM-GAN samples for a DNN. A diagram of a Perceptron-based detector in the system is shown in Figure 13.

We construct a quantized desired response $d(n)$ to a perceptron: $d(n)$ is 1 if $x(n)$ belongs to malicious class and 0 if $x(n)$ belongs to the benign class. The perceptron computes the weighted sum of the input patterns $x(n)$ comparing it to a threshold value. If the sum exceeds the threshold, the output is +1; otherwise, it is 0. The central computation of perceptron-based inference, the dot-product computation, is performed using a modest circuit that adds partial products sequentially. Since `0` and `1` are the only possible input values, multiplication is unnecessary to compute the dot product. We only need to add a weight when the input bit is `1`. The sign bit of the result gives the prediction. The weights are static since

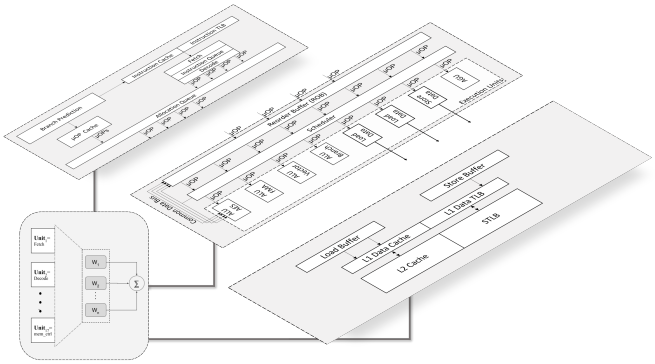


Figure 13. Hardware Detector

EVAX uses offline training; they would only change based on a new security patch. The dot-product is not latency-sensitive, in contrast to the perceptron-based branch predictors [36], [47] that must employ expensive adder trees to compute the dot-product quickly. In our case, we need a single adder to add or subtract serially each feature value. This will give us a result in a few hundred cycles in the worst case.Since the dot product can be computed sequentially, the hardware overhead of perceptron consists of three main components: 1) logic to find the dot product of the 145 weights and inputs, consisting mostly of a 9-bit adder and a register to accumulate the partial product, 2) storage for the weights and inputs, and 3) communication channels from the sources of the inputs to the storage for the inputs. Weights are in the range of [-2,1] and there are 145 of them. So the range of the values is from -290 to +145, or 435 distinct values that can be stored in 9 bits. The first two components have almost negligible overhead. We estimate that the dot product

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/disp/issue/commit 8 wide numPhysIntRegs=256,numPhysFloatRegs=256
L1 I-Cache	32KB, 64B line, 4-way
L1 D-Cache	64KB, 64B line, 8-way
L2 Shared Cache	2MB bank, 64B line, 8-way, responsLatency=20 mshrs=20, tgtsPerMshr=12, writeBuffers=8 tagLatency=20, dataLatency=20

Table II
PARAMETERS OF SIMULATED ARCHITECTURE

computation logic would have a transistor count no more than 4,000, a tiny fraction of the logic used for a perceptron-based branch predictor. The communication of the 145 signals from the disparate parts of the processor core to the dot product computation also has a very low cost in terms of area, but the design complexity would be non-trivial. Still, the effort should be no worse than that required to support performance monitoring units ubiquitous in today's processors that take signals from all over the core to a central location. Note that perceptron learning in hardware is practical for various applications, including branch prediction, prefetching, and replacement policies [85]. Recent microarchitectures from Oracle [81], AMD (, *e.g.* Bobcat, Jaguar, Piledriver, Zen, etc.), and Samsung [13], [36], [76] are documented to feature perceptron-based branch predictors.

Weight & Feature Updates. EVAX is capable of being updated via a vendor distributed patch. We anticipate newly emerging attacks in the future will require updates to neural weights and additions to the set of features being monitored. This is a process similar to microcode updates which is a secure feature available in current processors. With *future proofing* in mind, we envision that all available hardware performance counters will be connected to our detector allowing us to update the neural weights and monitored feature set as new attacks emerge. Evax inherits all of these hardware advantages from PerSpectron.

VII. METHODOLOGY

General Infrastructure. We use the Gem5 [11] cycle-level simulator to implement and evaluate the security and performance of our end-to-end defense. We simulate O3CPU which is a detailed and highly configurable out of order CPU model (it does timing simulation for both CPU and memory which is essential for simulating microarchitectural side channel attacks). We simulate a full-System model which include Operating System (Ubuntu 18.04 v. 4.8.13) and underlying architecture (X86 ISA). Our methodology should extend easily to other ISAs as well. Table II gives the parameters of the simulated architecture.

Attack Generation in gem5. We use methods described in [28] to successfully implement our attacks in gem5 for security analysis. We also extended the supported instructions [1] in gem5 to support new variants of microarchitectural attacks such as Medusa. Our modified gem5 uses an extensible DRAM simulator (Ramulator [51]) which runs as part of a full-system gem5. By design, gem5 and Ramulator do not include the management of Rowhammer effects, so we developed a dedicated memory corruption module and updated some existing components similar to [28] that stores an associative map to maintain the link between virtual and physical addresses. It determines the neighbors of each row and establishes the affected ones, counts the number of activations in each row since the last refresh, and affects one bit-flip threshold to each row. It establishes if one bit-flip occurs and modifies the affected cells in consequence. These modifications make it possible to successfully run existing Rowhammer attacks on an x86 platform associated with classical DRAM. Additionally, we use the same disk image to compare with real hardware, while keeping the same binaries and libraries between the real world and the simulated world. This makes testing attacks and ensuring simulation of security features and side-channel sources (e.g., caches, DVFS) more reproducible and easier than on a real platform. We create 30 Simpoints per benchmark in full system mode. We warm up by 1M instructions and then start execution from the precise checkpoints running 1M instructions from each (30M total). We simulate attacks to completion. Contrary to typical architectural studies, we generate many more, smaller simpoints of benign codes, since we need to train to detect short patterns quickly. From gem5 we collect values of 1160 microarchitectural counters, similar to prior work [62]. For these events, we measure total number, cycles, rate, average, and distribution. For counters, we maintain a maximum seen value for each sampling simulation point. Statistics are normalized over the maximum value of the counter.

Infrastructure for ML & HW Security Analysis. We have extended our framework to collect statistics once every 100,000, 10,000, 1000 and 100 instructions and sample all event counters for each program. We collect performance counters from the Gem5 simulator to train our detector, and measure prediction accuracy and leakage information. We do not collect performance counters from SW because the available number of performance counters in SW is limited – Intel processors can monitor four at a time. Many features correlated with malicious behavior are not available in software. GEM5 is not limited to the commit state and includes a rich feature set to monitor the speculative execution state and we can compare with the state-of-art detector, which relies on sampling a large set of microarchitectural features at once available in Gem5. Using this tool, we collect multi-dimensional time-series traces of applications. We use the Keras library [20] to implement and train our AM-GAN. We chose class Conditioned CGAN [63] as our reference GAN network. We used the FANN C library [68] to implement the

final EVAX and PerSpectron models to work with Gem5.

Infrastructure for Performance & Security Analysis. For performance analysis we have simulated two defenses implemented in GEM5 for the Spectre threat model. For Fencing we simulate a fence after every branch instruction. for InvisiSpec [90] we add support to track when instructions reach their visibility points, and issue load requests accordingly. We add a SpecBuffer at each L1, and a SpecBuffer for LLC. We modified GEM5 to bypass these defenses during performance mode and switch back to them after each true flag. We evaluate execution in secure mode for 10,000 instructions, 100,000 instructions and 1M instructions.

Workload We run individual SPEC CPU 2006 applications [41]. 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. For transient attacks, we simulate Spectre-PHT [54], Spectre-BTB [15], [53], Spectre-RSB [55], Spectre-STL [43], Meltdown [59], three variants of Medusa [66], Rowhammer [30], [50], SMotherSpectre [10], LVI (non-SGX environment) [87], FallOut [14], Branchscope [27], Microscope(non-SGX environment) [82], Leaky Buddies attack (CPU side) [23], RDRND [89], FlushConflict [89] and DRAMA [30]. For cache attacks, we run Flush+Flush [39], Flush+Reload [91], Prime+Probe [70].

Evasive Attacks. We generate 1.2M samples using our automated attack generation tools. We use Transynther [66], TRRespass [30] and Osiris [89] for automatic attack generation. For our manual evasive attacks experiments, we applied techniques in malware [4], [54] to produce 29 attacks and 400,000 attack samples that can evade the state of the art hardware detector [62]. We produce a total of 257,066 attack samples (and 70,000 benign program samples to balance the data) using AM-GAN at each Cross Validation Fold.

Cross Validation Setting. We use K-fold cross validation for measuring mean accuracy on unseen attacks. At every fold, we remove all the samples belonging to one attack in the test set so that they are not used for model selection or AM-GAN training. For transient execution attacks, to ensure EVAX detects the leakage and injection channel of the attacks, we also exclude the samples from recovery/transmission phase (check-pointed in our database) from the testing set of each fold. We use a set of fixed features. We don't select new sets of features whenever we exclude a sample, but we retrain the weights at each fold.

VIII. Security and Performance Analysis

In this section we discuss the security provided by EVAX. We report the robustness of detector against new evasive technologies. We show the performance of detector for defense against AML attacks, after retraining the detector on samples generated by these tools. We present detection accuracy for out-of-sample-attacks (zero-day setting). We then present the end-to-end solution with false positive and false negative results and with overall performance results compared to current always on mitigations.

A. End-to-End Performance Results for Zero Leakage

In this section we present results for ML detectors trained on all the attacks in our dataset. To guarantee security, EVAX is tuned to have very high sensitivity. Therefore it detected all of the attacks in our training set before leakage. Figure 14 shows the performance of the adaptive architecture enabled by EVAX compared, to PerSpectron and InvisiSpec. We can see that the IPC for InvisiSpec is lower than all the adaptive policies most of the time. Perspectron performs slightly better than InvisiSpec but still at very low IPC. EVAX keeps the IPC above 0.85 in most regions for mitigating Spectre (EVAX-SpectreSafe). IPC drops slightly for EVAX-FuturisticSafeFence which Fences all the loads. This architecture mitigates even the most dangerous class of microarchitectural attacks (e.g., LVI) with an overhead still significantly lower than prior mitigation techniques that are less aggressive.

False Positives and Negatives (for Zero Leakage). Figure 15 shows EVAX provides 85% improvement in False Positives (0.27 FN to 0.034) and 72% improvement in False Negatives (0.11 FN to 0.03). EVAX has 0.034 FP in every 10K instruction, which is 4 FPs in every 1M instruction. This is a practical False Positive for deployment. This number is 0.0005 FP and 0.0001 FN for sampling every 100 instructions. This result is for a model that is also trained on a dataset of samples taken every 100 instructions. Our model's accuracy in detection of microarchitectural attacks increases by higher sampling frequency which is expected since these attacks perform their anomalous behaviour during the transient window.

Performance Overhead (for Full Security Coverage). For end to end results we turn on mitigation at every true flag by our detector and we execute 1M instructions in secure mode to deactivate possible attacks. We only measure performance of benign program since performance of malicious programs is not a concern. We evaluate performance against hypothetical Futuristic Attacks Model that can exploit any speculative load and Spectre Model that can exploit speculative loads follow an unresolved control-flow instruction [90]. Figure 16 shows an end to end defense performance comparison. In the figure, Fences-FuturisticSafe simulates a fence before every load instruction, FuturisticSafeSpec is InvisiSpec working under Futuristic threat model. EVAX-FuturisticSafe enables Fences-FuturisticSafe and EVAX-SafeFence enables InvisiSpec. EVAX-SpectreSafe simulate a fence after every branch instruction. Figure 16 shows that EVAX reduces Fencing overhead for Spectre mitigation from 74% to 3.46% (95% reduction). EVAX reduces InvisiSpec performance overhead for spectre mitigation from 27% to 1.26% (95% reduction).

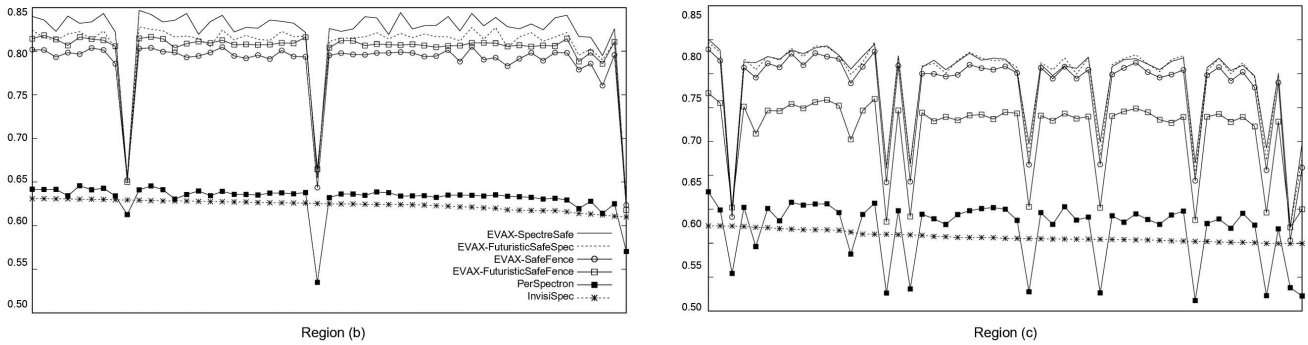


Figure 14. Performance (IPC) of an adaptive architecture with EVAX (vertical axis), compared to Perspectron and InvisiSpec. With EVAX, we also examine increasingly conservative fencing schemes to anticipate future attacks.

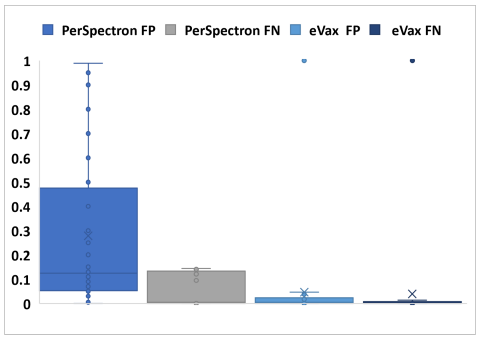


Figure 15. False Positive (FP), False Negative (FN) Distribution

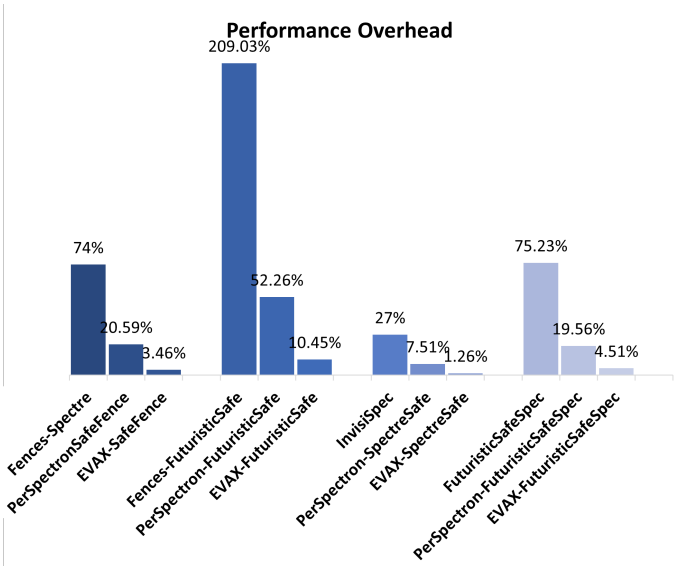


Figure 16. End to end Defense Performance Comparison

We show that EVAX reduces PerSpectron's performance overhead by factor of 6 for Spectre Fencing and by a factor of 4 for Futuristic (including LVI) mitigation. We show that EVAX reduces Fences overhead for Futuristic attack mitigation from 209% to 10% (95% reduction). EVAX reduces InvisiSpec overhead for Futuristic attack mitigation from 75% to 4% (94% reduction).

B. Resiliency Against New Evasive Tech

In this experiment, we have set aside one third of the samples from our dataset for test and both detectors are trained on the remaining 2/3 of the attack samples in our dataset. None of the detectors are trained on the attacks generated by the automated tools [30], [66], [89]. Figure 17 shows PerSpectron under the attack by fuzzing tools that break even the latest Intel processor. These tools include Automated Rowhammer attacks (TRRespass), Automated fault-based attacks (Medusa), and Automated timing based attacks (Osiris). We evaluate EVAX and PerSpectron on 1.2 Million attack samples generated by these powerful tools. We tune the EVAX output threshold to achieve detection resiliency as measured by the ROC curve. The average Area Under the Curve (AUC) of 0.797 for PerSpectron improves to 0.985 for EVAX (23.5% improvement). If we retrain the ML model on the same evasive tools, the accuracy on the adversarial attacks that evades the state-of-the-art plateaus at 78% using Automated Attack Discovery Tools. The accuracy improves to 93% with EVAX (where leakage = Zero, FNs before leakage.) AM-GAN generated samples effectively cover the gap between PerSpectron classification boundary and the Microarchitectural attack leakage window (see Figure 18).

C. K-fold Cross Validation Setting

One of the main differences between attack samples generated by our AM-GAN and fuzzing approaches is that it tells the model that there are *multiple correct answers*, *i.e.* multiple purturbation mechanisms or variations of a same attack, including those which haven't been seen (fuzzing is constrained by only a limited set of perturbations). This

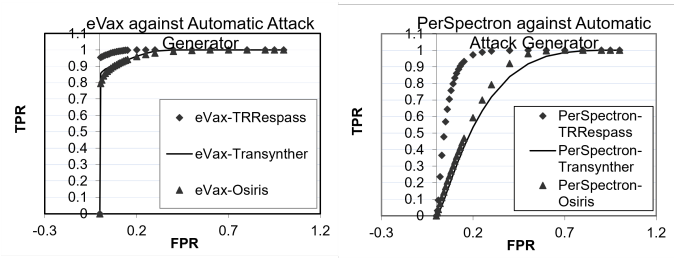


Figure 17. Resiliency (ROC curve) Against 1.2M Evasive Attacks

aligns with the objective of *seeking lower generalization error*, also known as the out-of-sample error or the risk. Generalization error is the conditional probability of correct prediction under an unknown state of parameters of the data generating mechanism, Figure 19 shows the classification error for EVAX, PerSpectron and PerSpectron hardened with sampling from the fuzzing tool (P.Fuzzer) in a zero day setting (K-fold cross validation). Evax drops the mean generalization error for PerSPectron, even when hardened by Fuzzing tools, by an order of magnitude.

EVAX is exposed to not only many attack samples but also many generated *benign program* samples through AM-GAN training of which represent comprehensive combination of normal utilization of micro architectural component that exist in the safe programs. While AM-GAN differentiate different attack types to generate samples of each for training, our final classifier is binary. Everything that is not microarchitecturally benign is classified as a microarchitectural attack. So when the system learns a very large corpus of safe behaviour it improves its ability to identify many anomalous activities in regards to microarchitectural utilization which are mal-utilized by attacks. Because AM-GAN generates a large number (theoretically indefinite) samples of *benign* and *malicious* programs through the adversarial game setup, it eventually learns many of the constraints and functionality rules of the pipeline, and augments this knowledge to the detector. For example, consider a feature *PendingQuiesceStallCycles* in fetch stage (Q) which is invariant to all kind of stalls down

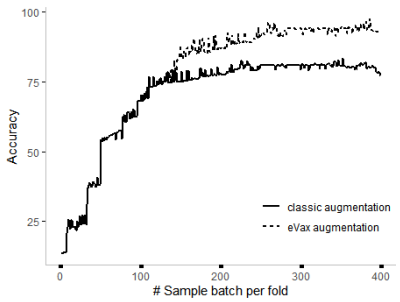
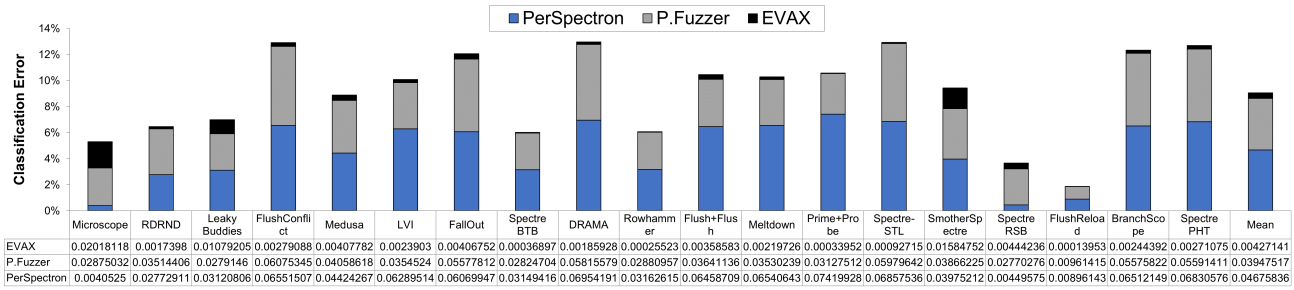


Figure 18. Filling the adversarial space. Increasing the accuracy on AML attacks. At 93%, leakage is Zero. Thus all attempts to evade the detector disabled the attack.

in the pipeline, e.g., flush, misses, and traps. $V = [f,m,t]$ shows the one hot representation of those counters. All of these stall signals propagate backwards to the fetch stage and are accumulated into the counter Q. EVAX's benefit comes from the AM-GAN Generator which learns to generate all possible combinations of input vectors V that trigger this *invariant feature* Q. While the classical training (PerSpectron) learns the first two of three features that trigger Q. EVAX does draw the full connection between the third feature t (traps) and Q automatically through training on benign program samples generated by AM-GANs (new combination of seen behaviours into one that is not seen), allowing it to generalize to other fault-based attacks such as LVI (reverse of meltdown), Medusa and all evasive attacks generated by Transynther that PerSpectron fails to detect. . We are able to see that the weight with value approximately 0 increases to value 2 as AM-GAN trains on benign samples enabling the model to learn the structure of the pipeline. We see this because Hyperplanes/perceptron are easy to understand and the first major step to cracking open a black box of deep learning.

Additionally, EVAX detects the RDRAND covert channel [89] with 95% TPR detection when removed from the training set. Weber et al. [89] showed that this attack is not easily detected nor prevented by any of the current software approaches [73] [45] [18] [42]. Evax also detects FlushConflict in a Cross Validation setting, a microarchitectural kernel-level ASLR (KASLR) bypass that is not mitigated by any of the current hardware fixes [89], with 97% true positive rate (TPR) for EVAX and 63% TPR for Perspectron. This attack works even on the newest Intel Ice Lake and Comet Lake microarchitectures, even with all known mitigations in place. We also consider the Medusa attack [66], a recent Meltdown-style attack that uses cache indexing, unaligned store-to-load forwarding, and shadow REP MOV in its different variations. Note that Medusa evades PerSpectron by completing its meltdown basic block prior to detection. EVAX detects all the meltdown-based steps of this attack with 98% true-positive rate (TPR) detection for EVAX and 38% TPR for Perspectron, demonstrating EVAX's robustness against advanced unseen meltdown-style and MDS attacks.

Interestingly, EVAX's main contribution is the reverse of prior works. PerSpectron mapped high-dimension data to low-dimension $[f,m] \rightarrow V$ (classification), while AM-GAN's Generator maps low dimension data to high dimension $V \rightarrow [f,m,t]$ (generation). This cross-validation experiment, when crafted carefully, can also test EVAX's increased ability to detect zero-day attacks. For example, EVAX can generalize to detect DRAMA [30] with high accuracy (99% TPR) even when it is excluded from the training set, while DRAMA evades PerSpectron. Highly correlated features that are triggered during TRRespass detection include: (1) *selfRefreshEnergy*, (2) *bytesPerActivate*, the number of accessed bytes per row activation in DRAM, (3) *bytesReadWrQ*, the



	Microscope	RDRND	Leaky Buddies	FlushConflict	Medusa	LVI	FallOut	Spectre BTB	DRAMA	Rowhammer	Flush+Flush	Meltdown	Prime+Probe	Spectre-STL	SmotherSpectre	Spectre RSB	FlushReload	BranchScope	Spectre PHT	Mean
EVAX	0.02018118	0.0017398	0.01079205	0.00279088	0.00407782	0.0023903	0.00406752	0.00036897	0.00185928	0.00025523	0.00358583	0.00219726	0.00033952	0.00092715	0.01584752	0.00444236	0.00013953	0.00244392	0.00271075	0.00427141
P.Fuzzer	0.02875032	0.03514406	0.0279146	0.06075345	0.04058618	0.0354524	0.05577812	0.02824704	0.05815579	0.02880957	0.03641136	0.03530239	0.03127512	0.05979642	0.03866225	0.02770276	0.00961415	0.05575822	0.05591411	0.03947517
PerSpectron	0.0040525	0.02772911	0.03120806	0.06551507	0.04424267	0.06289514	0.06069947	0.03149416	0.06954191	0.03162615	0.06458709	0.06540643	0.07419928	0.06857536	0.03975212	0.00449575	0.00896143	0.06512149	0.06830576	0.04675836

Figure 19. K-fold Cross Validation Setting

number of DRAM read requests serviced by the write queue. None of these attacks were included in the EVAX's training set at the fold they were being tested, showing EVAX could generalize to these attacks. EVAX cannot generalize to all zero-day attacks. MicroScope, Leaky Buddies (CPU-side) and SMotherSpectre all evade detection (leak before detection) when not part of the train set. While not perfect, of course, Evax is a clear advance in the state of the art for detection of *zero-day attacks*. However, if we retrain EVAX with the samples held out from MicroScope, Leaky Buddies (CPU-side) and SMotherSpectre, we can detect them with 99.3%, 99.0% and 99.7% accuracy indicating that there is a subset of shared features but not enough to allow *timely* detection in a cross validation setting. This is why it is still important to provide the ability to update the training and feature set of the the detector when such feature-novel attacks appear.

D. Improving Other ML Detectors.

We focus on perceptrons because of their advantages for hardware implementation, fast detection and interpretability. However, the techniques in this paper are applicable to a much wider range of ML solutions for attack detection. Figure 20 shows EVAX's training improves a 16-layer deep neural network's accuracy significantly (range of 0.57-0.90 for traditional training to 0.95-0.99 for EVAX training). Of particular significance is that our AM-GAN training enables a 16-layer neural network to outperform a 32-layer with it. The effect of adding layers in traditional training, not only is not statistically significant but also reduces the accuracy in some cases (Median of 85 for 16 Layer to 77 for 32 Layer). This is due to low quality training data. One of the challenges in ML detector design for adaptive architectures is noise in the training data. For example, while the program phases can be check-pointed in code for reducing noise related to benign and malicious phases of programs, the syscall itself adds noise to the attack sample. The CGAN EVAX framework is able to produce high quality, low-noise data for various types of attacks. This can remove the effect of noisy data and enable the capacity of deeper neural networks. One general take away is that increasing the complexity of neural networks without having a good set of training data can lead to statistically significant reduction in accuracy, as we see in Figure 20. On the other hand, higher quality training data can significantly improve the capacity of shallower networks as the accuracy of one layer NN with EVAX training is significantly higher than even a 32 layer NN with traditional training for all workloads (Range 0.57-0.90 for 32 layer compared to 0.88-0.99 for EVAX).

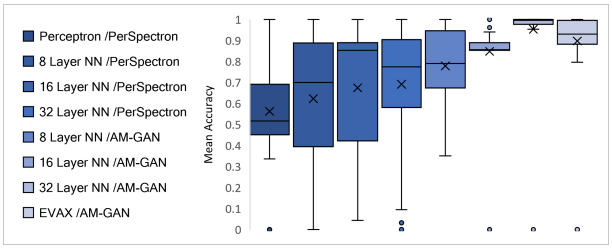


Figure 20. Improving other ML models with EVAX.

IX. Conclusion

EVAX proposes an end-to-end solution for runtime detection-based security against Microarchitectural attacks. We showed that code transformation techniques used in the prior automatic attack generation works are not sufficient for training an ML detector. This paper introduces a methodology for generating attack samples automatically in the microarchitectural layer rather than the code layer using GANs. The samples significantly increase the robustness of both simple and more complex microarchitectural attack detectors. EVAX also provides an automatic method to widen and linearize Deep NN by engineering features using trained GANs. Multiple catagories of attacks are then classified by a single hyperplane, which is fast and easily interpretable. We believe that linearization is the first major step to cracking open a black box of deep learning for systems. The high sensitivity and specificity of EVAX and its fast, implementable HW and interpretable design, makes it practical for deployment, substantially lowering the high performance overhead of current state-of-the-art microarchitectural mitigations.

Acknowledgment

We thank Daniel A. Jiménez for his comments during the drafting of this paper and Intel for a grant supporting this project.

References

[1] "gem5-avx," https://github.com/seanzw/gem5-avx, accessed: 2021-04-30.

[2] "Google. safeside: Understand and mitigate software-observable side-channels," https://github.com/google/safeside, accessed: 2019-01-03.

[3] "Ibm data breach report 2020," https://www.ibm.com/security/digital-assets/cost-data-breach-report/1CostofaDataBreachReport2020.pdf, accessed: 2022-04-08.

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

[5] "Qualcomm Announces Breakthrough Mobile Anti-malware Technology Utilizing Cognitive Computing," https://www.qualcomm.com/news/releases/2015/08/31/qualcomm-announces-breakthrough-mobile-anti-malware-technology-utilizing, 2015.

[6] "Intel Hardware Shield Overview (White Paper)," https://www.intel.com/content/www/us/en/architecture-and-technology/vpro/hardware-shield-overview-brief.html, 2020.

[7] O. Aciiçmez, "Yet another microarchitectural attack: exploiting i-cache," in *Proceedings of the 2007 ACM workshop on Computer security architecture*, 2007.

[8] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[9] J. Ba and R. Caruana, "Do deep nets really need to be deep?" in *NIPS*, 2014.

[10] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: exploiting speculative execution through port contention," *arXiv preprint arXiv:1903.01843*, 2019.

[11] 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*, 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[12] K. Bousmalis, A. Irpan, P. Wohlhart, Y. Bai, M. Kelcey, M. Kalakrishnan, L. Downs, J. Ibarz, P. Pastor, K. Konolige, S. Levine, and V. Vanhoucke, "Using simulation and domain adaptation to improve efficiency of deep robotic grasping," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2018.

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

[14] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar *et al.*, "Fallout: Leaking data on meltdown-resistant cpus," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[15] 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.

[16] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[17] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan, "Fact: A flexible, constant-time programming language," in *2017 IEEE Cybersecurity Development (SecDev)*, 2017.

[18] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Applied Soft Computing*, 2016.

[19] C. Chio, *Machine Learning & Security*. London: O'Reilly, 2018.

[20] F. Chollet, "keras," https://github.com/fchollet/keras, 2015.

[21] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019.

[22] 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)*, 2012.

[23] S. B. Dutta, H. Naghibijouybari, N. Abu-Ghazaleh, A. Marquez, and K. Barker, "Leaky buddies: Cross-component covert channels on integrated cpu-gpu systems," in *Proc. International Symposium on Computer Architecture (ISCA)*, 2021.

[24] C. Esteban, S. L. Hyland, and G. Rätsch, "Real-valued (medical) time series generation with recurrent conditional gans," 2017.

[25] D. Evtyushkin and D. Ponomarev, "Covert channels through random number generator: Mechanisms, capacity estimation and mitigations," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016.

[26] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[27] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," *ACM SIGPLAN Notices*, 2018.

[28] Q. Forcioli, J.-L. Danger, C. Maurice, L. Bossuet, F. Bruguier, M. Mushtaq, D. Novo, L. France, P. Benoit, S. Guilley *et al.*, "Virtual platform to analyze the security of a system on chip at microarchitectural level," in *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2021, pp. 96–102.

[29] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand pwning unit: Accelerating microarchitectural attacks with the gpu," in *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[30] P. Frigo, E. Vannacc, H. Hassan, V. v. der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "Trrespass: Exploiting the many sides of target row refresh," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[31] P. Frigo, E. Vannacc, H. Hassan, V. Van Der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "Trrespass: Exploiting the many sides of target row refresh," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[32] I. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *International Conference on Learning Representations*, 2015.

[33] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," 2014.

[34] B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi, "Absynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures." in *NDSS*, 2020.

[35] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[36] B. Grayson, J. Rupley, G. Z. Zuraski, E. Quinnell, D. A. Jiménez, T. Nakra, P. Kitchin, R. Hensley, E. Brekelbaum, V. Sinha, and A. Ghiya, "Evolution of the samsung exynos cpu microarchitecture," in *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[37] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another flip in the wall of rowhammer defenses," in *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[38] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer. js: A remote software-induced fault attack in javascript," in *International conference on detection of intrusions and malware, and vulnerability assessment*, 2016.

[39] 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*. Springer International Publishing, 2016.

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

[41] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, 2006.

[42] 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

[43] J. Horn, "speculative execution, variant 4: speculative store bypass," https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018, accessed: 2021-06-30.

[44] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$a: A shared cache attack that works across cores and defies vm sandboxing–and its application to aes," in *IEEE Symposium on Security and Privacy*, 2015.

[45] ——, "Mascat: Preventing microarchitectural attacks before distribution," in *ACM Conference on Data and Application Security and Privacy*, 2018.

[46] S. Islam, A. Moghimi, I. Bruhns, M. Krebbel, B. Gulmezoglu, T. Eisenbarth, and B. Sunar, "Spoiler: Speculative load hazards boost rowhammer and cache attacks," in *USENIX Security Symposium*, 2019.

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

[48] K. N. Khasawneh, N. Abu-Ghazaleh, D. Ponomarev, and L. Yu, "Rhmd: Evasion-resilient hardware malware detectors," in *Annual IEEE/ACM International Symposium on Microarchitecture*, 2017.

[49] N. Killoran, L. J. Lee, A. Delong, D. Duvenaud, and B. J. Frey, "Generating and designing dna with deep generative models," *arxiv preprint arxiv:1712.06148*, 2017.

[50] 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.

[51] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer architecture letters*, vol. 15, no. 1, pp. 45–49, 2015.

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

[53] 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.

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

[55] 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.

[56] E. M. Koruyeh, S. H. A. Shirazi, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Speccfi: Mitigating spectre attacks using cfi informed speculation," in *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2020.

[57] J. Lee, L. Xiao, S. Schoenholz, Y. Bahri, R. Novak, J. Sohl-Dickstein, and J. Pennington, "Wide neural networks of any depth evolve as linear models under gradient descent," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/0d1a9651497a38d8b1c3871c84528bd4-Paper.pdf

[58] C. Li and J. Gaudiot, "Challenges in detecting an "evasive spectre"," *IEEE Computer Architecture Letters*, 5555.

[59] 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.

[60] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[61] ——, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015.

[62] S. Mirbagher-Ajorpaz, G. Pokam, E. Mohammadian-Koruyeh, E. Garza, N. Abu-Ghazaleh, and D. A. Jiménez, "Perspectron: Detecting invariant footprints of microarchitectural attacks with perceptron," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[63] M. Mirza and S. Osindero, "Conditional generative adversarial nets," *CoRR*, 2014.

[64] T. Miyato, S. ichi Maeda, M. Koyama, K. Nakae, and S. Ishii, "Distributional smoothing with virtual adversarial training," *Arxiv preprint arxiv:1507.00677*, 2016.

[65] A. Moghimi, J. Wichelmann, T. Eisenbarth, and B. Sunar, "Memjam: A false dependency attack against constant-time crypto implementations," *International Journal of Parallel Programming*, 2019.

[66] D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz, "Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis," in *USENIX Security Symposium*, 2020.

[67] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "Deepfool: a simple and accurate method to fool deep neural networks," 2016.

[68] 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.

[69] A. Oliver, A. Odena, C. Raffel, E. D. Cubuk, and I. J. Goodfellow, "Realistic evaluation of deep semi-supervised learning algorithms," 2019.

[70] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *Cryptographers' track at the RSA conference*, 2006.

[71] R. Paccagnella, L. Luo, and C. W. Fletcher, "Lord of the ring (s): Side channel attacks on the cpu on-chip ring interconnect are practical," in *USENIX Security Symposium*, 2021.

[72] N. Papernot, F. Faghri, N. Carlini, I. Goodfellow, R. Feinman, A. Kurakin, C. Xie, Y. Sharma, T. Brown, A. Roy, A. Matyasko, V. Behzadan, K. Hambardzumyan, Z. Zhang, Y.-L. Juang, Z. Li, R. Sheatsley, A. Garg, J. Uesato, W. Gierke, Y. Dong, D. Berthelot, P. Hendricks, J. Rauber, and R. Long, "Technical report on the cleverhans v2.1.0 adversarial examples library," *arXiv preprint arXiv:1610.00768*, 2018.

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

[74] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "Drama: Exploiting dram addressing for cross-cpu attacks," in *USENIX Security Symposium*, 2016.

[75] X. Ren, L. Moody, M. Taram, M. Jordan, D. M. Tullsen, and A. Venkat, "I see dead μops: Leaking secrets via intel/amd micro-op caches," in *Proc. IEEE International Symposium on Computer Architecture (ISCA)*, 2021.

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

[77] G. Saileshwar and M. K. Qureshi, "Cleanupspec: An" undo" approach to safe speculation," in *Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[78] M. Schwarz, M. Lipp, D. Gruss, S. Weiser, C. Maurice, R. Spreitzer, and S. Mangard, "Keydrown: Eliminating software-based keystroke timing side-channel attacks," in *Network and Distributed System Security Symposium*. Internet Society, 2018.

[79] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," *arXiv preprint arXiv:1905.05726*, 2019.

[80] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, "Netspectre: Read arbitrary memory over network," *arXiv preprint arXiv:1807.10535*, 2018.

[81] 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*, 2012.

[82] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher, "Microscope: Enabling microarchitectural replay attacks," *IEEE Micro*, 2020.

[83] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," 2014.

[84] M. Taram, A. Venkat, and D. Tullsen, "Context-sensitive fencing: Securing speculative execution via microcode customization," 2019.

[85] 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 *International Symposium on Computer Architecture*, 2019.

[86] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient {Out-of-Order} execution," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[87] 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 *IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[88] 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*, 2019.

[89] D. Weber, A. Ibrahim, H. Nemati, M. Schwarz, and C. Rossow, "Osiris: Automated discovery of microarchitectural side channels," in *USENIX Security Symposium*, 2021.

[90] 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.

[91] Y. Yarom and K. Falkner, "Flush+ reload: a high resolution, low noise, l3 cache side-channel attack," in *USENIX Security Symposium*, 2014.