Protecting Confidential Virtual Machines from Hardware Performance Counter Side Channels

Xiaoxuan Lou*[1], Kangjie Chen*[1], Guowen Xu†[1], Han Qiu[2], Shangwei Guo[3], Tianwei Zhang[1]
[1]Nanyang Technological University, [2]Tsinghua University, [3]Chongqing University
{xiaoxuan001, kangjie001, tianwei.zhang}@ntu.edu.sg, †guowen.xu@foxmail.com,
qiuhan@tsinghua.edu.cn, swguo@cqu.edu.cn

***Abstract*—In modern cloud platforms, it is becoming more important to preserve the privacy of guest virtual machines (VMs) from the untrusted host. To this end, Secure Encrypted Virtualization (SEV) is developed as a hardware extension to protect VMs by encrypting their memory pages and register states. Unfortunately, such confidential VMs are still vulnerable to micro-architectural side channels, and Hardware Performance Counters (HPCs) are a prominent information leakage source. To make matters worse, currently there is no systematic defense against the HPC side channels.**

We introduce `Aegis`, a unified framework for demystifying the inherent relations between the instruction execution and HPC event statistics, and defending VMs against HPC side channels with provable privacy guarantee and minimal performance overhead. `Aegis` consists of three modules. Application Profiler profiles the application offline and adopts *information theory* to quantitatively estimate the vulnerability of HPC events. Event Fuzzer leverages the *fuzzing* technique to automatically generate interesting inputs, i.e., instruction sequences, that can effectively alter the HPC observations. Event Obfuscator injects noisy instructions into the protected VM based on the *differential privacy* mechanisms for high efficiency and privacy. We present three case studies to demonstrate that `Aegis` can defeat different types of HPC side-channel attacks (i.e., website fingerprinting, DNN model extraction, keystroke sniffing). Evaluations show that `Aegis` can effectively decrease the attack accuracy from 90% to 2%, with only 3% overhead on the application execution time and 7% overhead on the CPU usage.

I. Introduction

The maturity of cloud computing ecosystems prompts an increased emphasis on the privacy guarantee, making it a top concern along with the performance efficiency for cloud service providers and customers. To protect the guest virtual machine (VM) from the privileged but potentially malicious hypervisor, a promising mechanism called Trusted Execution Environment [45], [60] is utilized to realize confidential computing in virtualization scenarios. This mechanism adopts a hardware memory encryption engine to encrypt the VM's memory space transparently with a VM-specific key stored in the hardware layer. Even the hypervisor cannot access the encryption key and extract the memory content of the VM. AMD Secure Encrypted Virtualization (SEV) [47] is the first commercial realization of such a mechanism, and has been widely applied in modern cloud services [3], [9]. Other processor vendors have also released similar extensions, e.g., Intel TDX [13] and ARM CCA [6].

Encrypted virtualization can defeat most attacks that directly break the confidentiality or integrity of VM data. However, it is generally vulnerable to side-channel attacks, which use side-channel information (e.g., execution time, memory footprints) to infer the secrets in the encrypted environment. Although the latest SEV-SNP version has provided multiple protections [5] (e.g., Branch Target Buffer Isolation, disabling Instruction Based Sampling) against certain transient-execution and side-channel attacks, AMD admits that "*it is not able to protect against all possible side-channel attacks*" [61]. Over the years, a variety of attacks have been proposed to breach the confidentiality of SEV systems, which establish side channels via performance counters [72], cache occupancy [54], unprotected I/O operations [51], [58], page faults [40], [50], [57], etc.

Among these attacks, Hardware Performance Counter (HPC) side channels are particularly easy to exploit. HPCs are implemented as a tool for software profiling, debugging and system modeling. Security researchers also repurposed them for malware and intrusion detection [28], [70], [78]. Unfortunately, their capability of inspecting program's behaviors can be abused by the adversary to infer the secret from the victim program [20], [38], [52], [69], [77], especially in the cloud scenario where the malicious host can arbitrarily read the HPC register values mapping to a victim VM. While SEV claims HPC leakage can only reveal trivial information about the application inside the VM and hence does not prevent it from the hardware [61], its competitor Intel TDX seriously considers HPC leakage as a target that must be prevented and proposes specific hardware modifications to isolate virtualized guest HPC registers from the malicious host [12]. We present three case studies in Section III to show that HPC side channels can indeed lead to more severe confidentiality attacks, e.g., leaking the website access, keystroke, and machine learning models. Hence, the confidential VM should well mitigate HPC leakage to protect guest secrets, just as Intel TDX has done.

However, currently all public cloud platforms mainly adopt SEV to achieve their confidential VM services, including Azure confidential VM [7], Google cloud [10] and AWS EC2 [2], while TDX is only a preview version or even has not been considered [8], [11]. Given AMD has announced that SEV-SNP is the latest stable version, it seems that AMD has no motivation to update the hardware design to prevent HPC

*Co-first authors with equal contributions.
†Corresponding author

side channels, which exposes almost existing confidential VM systems to the security threat. As a result, the lack of hardware defence against HPC side channels in SEV is an urgent and practical security issue, which motivates us to design a practical software-based solution to mitigate this vulnerability for off-the-shelf platforms immediately.

Past efforts have been devoted to reinforce the encrypted VMs and enclaves against different micro-architectural side channels, e.g., CPU cache [33], [37], page faults [25], [62], branch prediction [42], [49]. However, to the best of our knowledge, there are no previous works to systematically and comprehensively investigate the HPC side-channel defenses against malicious hypervisor. This motivates us to design an effective and efficient defense solution for the customers to protect their sealed application from HPC side channels. However, achieving this goal is not a trivial work and several challenges must be carefully addressed. First, modern processors usually support a large number of HPC events (usually in the thousands). This gives the adversary more flexibility to establish the side channel. Meanwhile, it also gives the defender more difficulties to analyze the possible threats. Second, HPCs cannot count performance events precisely because of the external interference, e.g., hardware interrupts, which may mislead the identification of HPC value changes and result in false analysis results. Finally, one common side-channel defense strategy is to obfuscate the adversary's observations. A straightforward way is to inject random noise directly. However, this method introduces extra noise into VMs, which incurs additional performance overhead. Besides, randomly injecting noise cannot provide a rigorous privacy guarantee.

Motivated by these challenges, we propose `Aegis`, a framework that can automatically analyze the potential HPC side channels of a victim application from customers, and effectively prevent HPC side-channel attacks with a provable security guarantee. `Aegis` consists of three modules:
(1) **Application Profiler**. It is used to profile the protected application and extract all vulnerable HPC events that can act as attack surfaces. It gives us the first look at the application and a comprehensive understanding on usable attack surfaces. The vulnerability of different events are estimated and ranked with *information theory*.
(2) **Event Fuzzer**. It is used to find out all possible instruction gadgets that can alter the profiled vulnerable HPC events. We model this as a bug identification task, and design an automatic tool based on the *fuzzing-like* technique to generate interesting "inputs", i.e., instruction gadgets, to evaluate if the system reports a "bug", i.e., value change in the target HPC events.
(3) **Event Obfuscator**. It is located inside the victim VM and injects specific numbers of instruction gadgets into the execution flow of the VM as the noise to obfuscate HPC values monitored from the outside attacker. The key insight is to model the execution behaviors of the victim VM as statistical data, and then inject random noise following a specific *differential privacy* mechanism. This reduces potential information leakage and makes the victim's activities indistinguishable from the attacker's observations. Meanwhile, the differential privacy mechanism can theoretically guide us to identify the optimal amount of noise, achieving desired privacy guarantee with the minimal impact on the system.

Our framework `Aegis` is the first work aiming to systematically and comprehensively mitigate HPC side channels on encrypted VMs. It is a general and provable solution that can be readily deployed inside the VM for a strong privacy guarantee. Two differential privacy mechanisms (Laplace and d^*) are adopted to defeat attacks. Experimental results show that both mechanisms can effectively reduce the attack accuracy from $> 90\%$ to 2%, closing to random guess, while only introducing about 3% overhead on the application execution time and 7% overhead on the CPU usage.

II. Background and Related Works

A. Hardware Performance Counters

Modern processors implement a large spectrum of Hardware Performance Counters (HPCs) in each CPU core to record the occurrences of hardware-related events for processes and the entire computer system. These counters provide developers a useful tool for dynamic software profiling, debugging and system modeling. However, HPCs also expose an exploitable surface, where the attacker can obtain the execution behaviors of a protected program in the system and further recover sensitive information. Prior works showed the feasibility of revealing secret keys from the cryptographic applications based on HPCs [20], [69]. Furthermore, HPCs can also provide high-resolution timing information to facilitate cache attacks [52], [77] or even reveal the website accesses [38].

B. Secure Encrypted Virtualization

Secure Encrypted Virtualization (SEV) is a new feature in AMD processors [47], which combines AMD-Virtualization (AMD-V) and Secure Memory Encryption (SME) technologies to encrypt individual VMs with their own keys, aiming to protect VMs from the untrusted hypervisor. A dedicated co-processor, Platform Security Processor (PSP), is introduced to generate distinct ephemeral keys for each VM, and encrypts VM's data outside the processor. This extension can effectively protect the secrets in VMs from physical attacks (e.g., cold boot and DMA attacks) and privileged software attacks, making the encrypted VM a total black-box to the malicious hypervisor. A later SEV-ES [46] version further encrypts VM's CPU register state during world switches, which prevents the malicious hypervisor directly accessing or modifying VM states. The latest SEV-SNP [61] version finally achieves integrity protection of VM memory with Reverse Map Table and fixes vulnerabilities in two earlier versions. Other processor vendors, e.g., Intel [13] and ARM [6], are also working on these security features, which is a promising direction for trustworthy cloud computing.

C. Fuzzing

Fuzzing is a popular software testing technique to automatically find bugs in software applications [56]. A fuzzer typically generates a significant number of test inputs and monitors software execution over these inputs to detect abnormal behaviors. There are two common fuzzing strategies: (1) mutation-based fuzzers [19], [41] select an initial set of inputs as the seed, and then generate inputs by applying mutations, e.g., splicing, bit flipping. They usually require the analyst to have prior domain knowledge. (2) Grammar-based fuzzers [21], [39] exploit existing input specifications to generate a grammar model to conduct inputs. They sometimes fail to reach certain *corners* of the input space. Recently, hardware fuzzing has become increasingly popular. Researchers adopt this technique to find undocumented x86 instructions [32], discover side channels [71], improve Meltdown [75] and Spectre attacks [67]. Different from those works that use fuzzing to facilitate hardware attacks, we apply it to defeat side channels.

D. Differential Privacy

This technology was originally used to protect statistical databases by withholding information about individuals [34]. One popular Differential Privacy (DP) solution is ϵ-DP. A randomized algorithm $\mathcal{A} : \mathcal{X} \rightarrow \mathcal{Z}$ satisfies ϵ-DP if for any adjacent datasets x, x' and all $Z \subseteq \mathcal{Z}$, it has: $\mathbb{P}(\mathcal{A}(x) \in Z) \leq exp(\epsilon) \times \mathbb{P}(\mathcal{A}(x') \in Z)$, where $\epsilon \geq 0$ indicates the privacy budget to control the level of privacy protection. Chatzikokolakis et al. [24] proposed a generalization of differential privacy called d-privacy. Compared to ϵ-DP, d-privacy is more suitable for datasets with time series correlations, giving better privacy guarantees under the same privacy budget. Specifically, a metric d on a set $\mathcal{X}$ is defined as a function $d : \mathcal{X}^2 \rightarrow [0, \infty)$. A randomized algorithm $\mathcal{A} : \mathcal{X} \rightarrow \mathcal{Z}$ satisfies (d, ϵ)-privacy if for all $Z \subseteq \mathcal{Z}$, it has: $\mathbb{P}(\mathcal{A}(x) \in Z) \leq exp(\epsilon \times d(x, x')) \times \mathbb{P}(\mathcal{A}(x') \in Z)$. DP has also been used to mitigate storage side channels [74] and network side channels [80]. In this paper, we apply this technique to defeat the HPC side channels. This is more challenging as there are more possible leakage sources (e.g., a large number of vulnerable HPC events) and it is hard to control the micro-architectural noise.

III. HPC Side Channels

A. Threat Model

We consider an IaaS cloud scenario protected with encrypted virtualization feature, e.g., AMD SEV, which hence establishes mutual distrust between the customer and the cloud provider. The customer first asks to launch an encrypted VM in the cloud and then perform remote authentication and attestation to confirm if the hardware details and security settings are correct. After the setting confirmation, the sensitive data is sent to the encrypted VM through an encrypted communication channel. As for the cloud provider, we assume it is honest-but-curious, namely it will abide by the defined service agreement but will also seek to gain more sensitive information that it is not explicitly authorized to have. For example, the hypervisor would provide correct register values to guest VMs, including the HPC values. This assumption is realistic for most commercial cloud platforms and has been considered in prior works [50], [72].

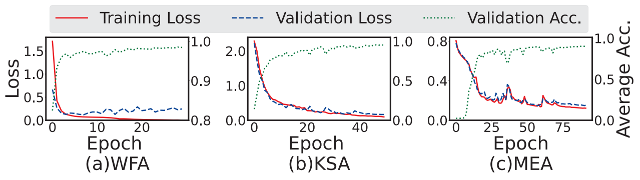


Fig. 1: Training curves of three HPC side-channel attacks.

While the SEV protection prevents the malicious hypervisor directly extracting the VM's information (e.g., memory, registers, instructions), some micro-architectural side channels can still be constructed to infer the customer's data. In this paper, we mainly focus on the HPC side channels, where the adversary only needs to monitor HPC events passively, making this attack more stealthy than other active side-channel attacks. Prior works have demonstrated that the adversary can leverage HPCs to accurately extract the victim program's secrets [20], [38], [69], making them a severe threat to the confidentiality of trusted computing systems.

B. Abstraction of HPC Side-channel Attacks

Following previous attacks on various side channels (e.g., power [27], CPU cache [79]), we also model the HPC side-channel attacks as a machine learning task. In the offline stage, the attacker can deploy a template application executing with different secrets $\mathcal{Y}$. Meanwhile, he profiles the application execution behaviors and collects the HPC event leakage traces $\mathcal{X}$. Each trace $x \in \mathcal{X}$ is a time-series of length T, where every time slice $x[t]$ is a vector of monitored events, $1 \leq t \leq T$. Then the attacker trains a parameterized machine learning model f_θ to establish a mapping between $\mathcal{X}$ and $\mathcal{Y}$, i.e., $f_\theta : \mathcal{X} \mapsto \mathcal{Y}$. In the online stage, when the actual victim runs with a secret, the attacker monitors its HPC leakage x, and is able to predict the secret as $y = f_\theta(x)$.

Below we present three practical HPC side-channel attacks against the encrypted VM. The attacks are implemented on an AMD EPYC 7252 processor that supports the SEV protection. The victim VM has the configurations of 4 CPUs, 8G memory and 80G disk, and is launched by the qemu script from AMD [4]. Both the host platform and VM use the Ubuntu 20.04 OS with kernel version 5.11.0. We assume the attacker monitors four HPC events for each attack, as modern processors are usually equipped with four HPC registers. In this demonstration, the events we used are: `RETIRED_UOPS`, `LS_DISPATCH`, `MAB_ALLOCATION_BY_PIPE` and `DATA_CACHE_REFILLS_FROM_SYSTEM`, which cover instruction retirements, operation dispatch and cache accesses. The selection of these HPC events is determined by the ranking results shown in Section VIII-A. These four events would leak most information about the secrets sealed in the confidential VM.

C. Website Fingerprinting Attack

First, we demonstrate an effective website fingerprinting attack (WFA), where the attacker can precisely reveal the website accesses in the encrypted VM. Previous works realized this goal in various scenarios [31], [63], [64].

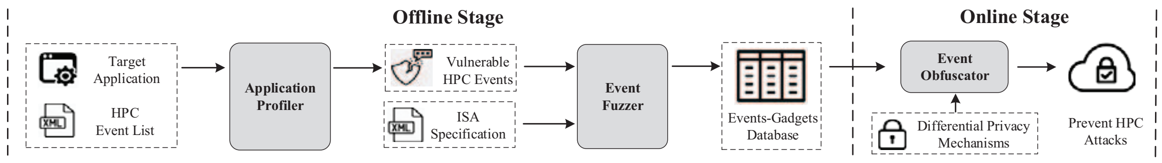


Fig. 2: Overview of our `Aegis` framework.

Attack implementation. We design a compact CNN model to fingerprint the website. The model consists of four convolution layers and three fully-connected layers, and also employs common optimizations like batch normalization [43] and dropout [66] layers. We select 45 websites from Alexa top-50 websites [1] (excluding 5 blocked websites) as the attack targets, i.e., labels ($\mathcal{Y}$) of the model. The attacker accesses each website using the Chrome browser for 1000 times in the template VM. These websites are accessed in a rolling sequence, which can capture the variants of the sites over time and also avoid IP address blocking by the web servers. At the same time, he uses the host HPCs to sample the counts of four profiled events ($\mathcal{X}$). The sampling process lasts for 3 seconds with an interval of 1ms, giving him a tensor with the size of 4×3000 for each website access. We randomly select 70% and 30% of the dataset for training and validation, respectively.

Attack results. Fig. 1a depicts the trends of model accuracy and loss during the training. We can see that the attack accuracy improves very quickly until reaching a stable value (e.g., 98.72%). The well-trained model is then used to predict 4500 actual website accesses (100 accesses for each website) in the victim VM, which achieves an accuracy of 98.57%. It confirms that HPCs can be used to conduct website fingerprinting attacks with high fidelity and efficiency.

D. Keystroke Sniffing Attack

Another classic side-channel attack is the keystroke sniffing attack (KSA), where the attacker aims to infer the keystrokes entered by the victim user. Past works realized the attack with different side channels (e.g., timing [65], memory [44]).

Attack implementation. As the timing characteristics of keystroke actions can leak information about what those keystrokes are [65], the attack target ($\mathcal{Y}$) is set as the number of keystrokes occurred during the period T, whose timing patterns can be used to infer the keys pressed. The collected leakage trace ($\mathcal{X}$) is the same as WFA. The attacker can also adopt the same CNN model in WFA to predict keystroke actions. Following the settings in previous works [44], [74], we use the tool `xdotool` [17] to simulate the keystroke actions. It generates K keystrokes in 3 seconds, where K is a random number between $[0, 9]$. This process is repeated 10,000 times in the template VM, where the corresponding HPC event leakage is captured simultaneously. We randomly select 70% for training while the remaining is for validation.

Attack results. Fig. 1b shows the training curves of the attacker's model, where the prediction accuracy can finally reach 95.21%. It also shows a very high sniffing accuracy (95.48%) on the test set.

E. Model Extraction Attack

Our last case is the model extraction attack (MEA), which steals the complete neural network architecture of a DNN model. This attack has been realized with the remote-query fashion [68], power [73] and cache [55], [76] side channels.

Attack implementation. The prediction label $\mathcal{Y}$ in MEA is a sequence of layers forming the target DNN model architecture. Hence, this attack should be modeled as a sequence-to-sequence learning task instead of a classification task. We design a RNN model with the CTC decoder [36] to solve this problem. A bidirectional GRU [26] is adopted as the RNN module, as it enables better long-term memory and fully leverages temporal contexts. The best predicted layer sequence is identified with the beam search, which reveals the target model architecture. We select the 30 most commonly used DNN models from the Pytorch official library [16]. We run the inference execution of each model for 1000 times, collect the HPC event sequences, and construct the training and validation sets in a similar way as WFA. Then we train the RNN model to predict the target layer sequence.

Attack results. Fig. 1c shows the model training results. The prediction accuracy keeps increasing and finally stays at a stable value (e.g., 91.8%). Note that the accuracy reflects the statistics of the matched layers between prediction and label sequences. The test on the victim VM also shows high attack accuracy (i.e., 90.5%), indicating that the attacker can almost extract the complete architecture of the target DNN model running in the encrypted VM.

IV. Framework Overview

We aim to design a defense framework for guest users to protect their applications from HPC side channels. A user can only deploy the defense inside his VM, but cannot control the privileged software or hardware on the host server. He does not know which performance counter(s) the attacker will use for information extraction.

We introduce `Aegis` to achieve our defense goal. The basic idea is to inject noisy instructions into the protected VM's execution flow, which can mask its secret from the HPC events. `Aegis` has the following benefits. (1) *Unified*: given a protected application, `Aegis` can mitigate different HPC side channels, regardless of the extraction methodology or performance counters used. (2) *Provable*: we can theoretically guarantee the security of the defense under the given privacy budget. (3) *Automatic*: the entire defense deployment can be achieved automatically without any prior knowledge or human efforts. Fig. 2 shows the workflow of `Aegis`, which consists of three modules in the offline and online stages. The two modules in the offline stage are only performed for one time, and the analyzed results would be applied in the online stage.

Application Profiler (Section V). This module is used to profile the target application with user-specified secrets in the VM, and collect the corresponding leakage of all available HPC events. It adopts information theory to quantify the correlation between the secret data and each event, and identifies the HPC events that are vulnerable as side channels. These events serve as the target for us to defend against.

Event Fuzzer (Section VI). This module aims to automatically find out the possible instruction gadgets that can alter the side-channel observations of the HPC events identified from Application Profiler. It takes a machine-readable ISA (Instruction Set Architecture) specification and the list of vulnerable events as the input. With a well-designed grammar model, the fuzzing performs on a significantly reduced search space and finally outputs the instruction gadgets that can disturb the values of these vulnerable HPC events.

Event Obfuscator (Section VII). This module injects the instruction gadgets, selected by Event Fuzzer into the protected VM at runtime, which can effectively mask the HPC values observed by the adversary outside the VM, and prevent side-channel information leakage. To provide a provable security guarantee, we introduce the differential privacy mechanisms to regulate the number of injected instruction gadgets.

V. Application Profiler

A. Challenges

The first step of `Aegis` is to identify possible attack surfaces (e.g., HPC events) that can leak secrets from the target application. There are several challenges to achieve this goal.

C1. Numerous HPC events. A modern processor usually supports a large number of HPC events and any event can be vulnerable to the victim application. For example, in our experiment platforms, the Intel Xeon E5-1650 processor has 6166 usable events while the AMD EPYC 7252 processor has 1903 supported events. While each event should be fuzzed with all possible combinations of instruction sequences (Section VI), such numerous events place a heavy burden on the comprehensive analysis of potential leakage sources.

C2. Heterogeneity and Non-determinism. The number and type of available HPC events highly depend on the processor family. Table I shows the statistics of HPC events from two Intel CPUs and two AMD CPUs. We can see that CPUs from the same family (e.g., Intel E5 family) share similar hardware features (i.e., HPC events), while processors from different families can vary greatly. Besides, it is well known that HPCs cannot provide precise counts of system performance events, because of the external interference, e.g., kernel interactions, hardware interrupts [29]. This introduces extra noise to the profiling process, which may mislead the observation of changes in HPC values.

C3. Vulnerability quantification. Different events lead to various levels of vulnerability to HPC side-channel attacks, i.e., some events can leak more information about the target. Hence, the quantification of event vulnerability is necessary to perform more efficient defense, as we can pay more attentions to those more vulnerable events. However, this issue has never been discussed before and it poses a practical challenge.

HPC Statistics	Intel Xeon E5-1650	Intel Xeon E5-4617	AMD EPYC 7252	AMD EPYC 7313P
# of HPC Events	6166	6172	1903	1903
# of Different Events	/	14	/	0

TABLE I: Statistics of HPC events in various processors.

B. Profiling Design

We design an offline Application Profiler module to tackle the above challenges and identify the vulnerable HPC events for a given application executing specified secrets. Basically, we launch a template VM on a template server where we have the host privileges. This server should have a similar processor model (i.e., in the same processor family) as the target cloud server[1] to guarantee the generality of the identified events. Note that the template server can be either a local server or provided by a third-party entity, which has no conflict of interest (e.g., a government agency or a neutral authority). For instance, the guest can rent a bare-metal server from public cloud providers (e.g., AWS or Azure) to serve as the template server, which can have much less resources than the target cloud server and only the processor model needs to be similar. Then we run the target application with a set of customer-specified secrets in the template VM for multiple times and monitor each available HPC event from the host. Finally, the monitored results are automatically analyzed by the program to rank the HPC events based on their vulnerability, i.e., information gain that can be used to extract the application. After the profiling, we can be sure that, at least for the specified secrets, all possible HPC events this application could trigger have been exhaustively activated.

Monitoring setup. We need to first extract the full list of available HPC events for the given processor model. This can be achieved with a third-party tool `libpfm4` [15]. Then we configure the performance monitoring tool to measure each count of events. In our implementation, we adopt the Linux kernel interface `perf_event_open`, which can effectively reduce the measurement noise as it interacts with the kernel directly. We also set the `pid` and `exclude_kernel` attributes to achieve VM-specific monitoring and prevent influence from the host kernel or other processes. For each time of profiling, we simultaneously monitor four HPC events, which is determined by the upper limit of available HPC registers on the processor. It achieves a suitable trade-off between the monitoring performance and accuracy, as the `perf` subsystem uses time multiplexing [14] for monitoring when there are more monitored events than available registers, which would affect the value accuracy.

Warm-up profiling. We perform a warm-up profiling to compact the event list and reduce the complexity of vulnerability

[1]The processor model of the cloud server is obtained from the AMD PSP during the remote attestation.

CPU Processor	Percentage of various event types (%)					
	H	S	HC	T	R	O
Intel Xeon E5-1650	0.39 (100)	0.31 (0)	1.00 (100)	36.15 (7.98)	7.75 (99.37)	54.40 (0)
AMD EPYC 7252	1.26 (100)	1.00 (0)	3.26 (100)	87.17 (1.57)	5.20 (91.83)	2.11 (0)

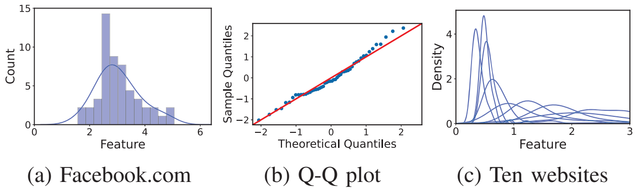
TABLE II: HPC event distribution, including events of Hardware (H), Software (S), Hardware Cache (HC), Tracepoint (T), Raw CPU (R) and Others (O). Data in brackets shows the percentage remaining after the warm-up profiling.

analysis. The key idea is that a majority of HPC events cannot reflect the activities inside a guest VM. To exclude those events, we measure and compare the event counts when the VM runs the application and when it is idle. The events without any value changes in the counts will be removed from the list, as they cannot reflect the application behaviors. After this warm-up profiling, we only get less than 10% of the events. Take the website fingerprinting analysis as an example, in our two experiment platforms, only 738 (Intel) and 137 (AMD) events remain for further analysis after 5 repeated warm-up profilings, where the results of each profiling are almost the same.

To give more insights on the profiled HPC events, we perform a comprehensive analysis and summarize different types of HPC events in two CPU processors, as shown in Table II. Note that Table II actually contains all available events that can be monitored through `perf` subsystem, which include events that are not collected through hardware features, e.g., software (S)/tracepoint (T) events. But for the sake of description simplicity, they are covered together and also named as HPC events. We observe that the tracepoint events (T) and other events (O) account for nearly 90% of the total number. The *tracepoint events* measure the access states on the tracepoints provided by the host kernel infrastructure, such as most system calls, most of which cannot precisely capture the application behaviors isolated inside the VM. The *other events* mainly denote the cases at a low level, like hardware breakpoints provided by the CPU, which are generally not be invoked by normal VM applications. We also give the remaining percentage of each event type after the warm-up profiling in Table II, where the remaining events mainly consist of hardware events (H/HC) and CPU raw events (R), while software (S)/other (O) events and most tracepoint events (T) are removed. It indicates that HPC leakages from sealed VM applications are mainly reflected at the hardware level.

Event ranking. Our last stage is to profile and rank HPC events based on their vulnerabilities. Given a specific event, assume $\mathcal{Y}$ denote the set of customer-specified secrets executed in the victim application, which contains N_y objects. $\mathcal{X}$ denotes the profiled value traces for the given HPC event, containing $N_x = N_y \times m$ objects, which repeatedly performs m measurements for each secret object. It can average out the non-determined event values.

While the leakage trace $x \in \mathcal{X}$ is a time-series, we first



(a) Facebook.com (b) Q-Q plot (c) Ten websites

Fig. 3: The distribution of HPC event values.

extract the feature value of the sequence with Principle Component Analysis (PCA) [59], which is a widely used feature extraction method for processing high-dimensional data while preserving most of the information. For the sake of simple computation, we follow previous work [65] to fit the monitored event values as a Gaussian-like unimodal distribution. Our observation also shows that most event values indeed distributed normally. Fig. 3a gives an example distribution over the event `DATA_CACHE_REFILLS_FROM_SYSTEM` on the website *facebook.com*. In Fig. 3b, we quantitatively compare the real distribution of the event values to Gaussian distribution $\mathcal{N}(0,1)$ with the Q-Q plot [35]. The result confirms that the HPC event values of a secret (e.g., a website access) indeed follow the Gaussian distribution. Hence, we can naturally assume that the probability of the event value x between the target secret $y \in \mathcal{Y}$, $P(x|y)$, forms a univariate Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$, i.e., $P(x|y) = \frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$. Fig. 3c shows the estimated distributions of the event values on 10 websites. Although the distributions of some websites overlap slightly, they can still be classified easily, which explains the high attack accuracy of WFA in Section III.

With Gaussian modelling of HPC event values, we can then quantitatively estimate the information gain, i.e., *mutual information*, induced by the given HPC event. First the entropy of the probability distribution of secret y is $H(\mathcal{Y}) = -\sum_{y\in\mathcal{Y}} P(y)logP(y)$. Then, given a event feature value x_0, the entropy of the probability distribution of secret y is $H(\mathcal{Y}|\mathcal{X} = x_0) = -\sum_{y\in\mathcal{Y}} P(y|x_0)logP(y|x_0)$, where $P(y|x_0) = \frac{P(x_0|y)P(y)}{\sum_{y\in\mathcal{Y}} P(x_0|y)P(y)}$. Hence, the *mutual information* can be computed as:

$$I(\mathcal{Y};\mathcal{X}) = H(\mathcal{Y}) - \int P(x_0)H(\mathcal{Y}|\mathcal{X} = x_0)dx_0 \qquad (1)$$

where $P(x_0) = \sum_{y\in\mathcal{Y}} P(x_0|y)P(y)$. The computed $I(\mathcal{Y};\mathcal{X})$ is the metric to quantify vulnerability of the HPC events.

VI. Event Fuzzer

For each identified vulnerable HPC event from Section V, `Aegis` calls the offline module Event Fuzzer to search for the instruction sequence gadgets, which can alter the HPC event value, and obfuscate the adversary's side-channel observation. To make this process automatic and efficient, we propose to use the fuzzing technique to find the qualified instructions.

A. Challenges

There exist a couple of challenges to achieve our goal.

C4. Lack of prior knowledge. Since previous works have never systematically discussed the relationship between instruction execution and changes in HPC counts, we are totally blind to the validity of instructions on obfuscating HPC values. This poses a significant challenge, as we have to fuzz all possible combinations of instructions without knowing their effects ahead, which leads to an infinite search space.

C5. Undesired side effects. The instruction sequence may exhibit unexpected side effects, which could fool the fuzzer and mislead the testing path. For instance, the `store` instruction not only loads the target data into the memory, but also changes the cache state. It creates multiple branches of the code testing, which hence exponentially increases the complexity of the fuzzing process.

C6. Inherited dirty state. To accelerate the search process with high efficiency, all generated instruction gadgets are fuzzed in an uninterrupted way. Unfortunately, such scheme can leave the dirty state of the current gadget to the subsequent ones. For example, following gadgets would inherit the cache state of previous gadgets. Such dirty state entangles successive gadgets, which greatly interferes the fuzzing process and often results in false positive results.

B. Design Overview

The search of instruction gadgets is modeled as a fuzzing problem. We consider the value change of identified HPC events as a *runtime bug*, and the target is to generate more efficient *inputs* (i.e., instruction gadgets) that can lead to such *bugs*. Given the lack of prior knowledge, using mutation-based fuzzing is not a reasonable choice, as we have no idea about well-performing seeds. To combat that, we adopt grammar-based fuzzing, which however requires a well-designed format model to reduce the search space of inputs.

While the instruction gadget aims to change the HPC event to a specific state, we divide it into two actions, as shown in Fig. 4. (1) We first bring the event to a known *reset state* (S_0) with a *reset instruction sequence*. For instance, to monitor the event of cache references, we issue a `clflush` instruction to empty the cache line. (2) Then a *trigger instruction sequence* is used to transition the event from (S_0) to the desired *trigger state* (S_1), in which the value of monitored HPC event is changed due to the effect of trigger instructions. The combination of the reset and trigger sequences forms the instruction sequence gadget for obfuscating the HPC events, which hence can be considered as the format model for the fuzzing input generation.

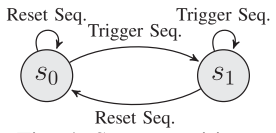


Fig. 4: State transition.

Fig. 5 shows the workflow of our Event Fuzzer, which consists of the following steps. ① We first clean up the machine-readable ISA specification and remove all illegal instructions for the platform microarchitecture. ② We search for qualified instruction gadgets for the profiled HPC events.The module automatically generates the reset and trigger instruction sequences from the cleaned instruction list, and constructs the gadgets following the fuzzing grammar, i.e., the input format model discussed above. Then it executes the generated gadgets and monitors the value change of the target HPC events. All gadgets that can alter the event value are recorded as the obfuscating factors. ③ We further validate the effectiveness of the identified gadgets with multiple mechanisms, aiming to remove the fuzzing paths invoked by undesired side effects of instructions and mitigate the corner cases caused by the inherited dirty state. ④ Finally we cluster the similar gadgets and filter the best ones for the profiled HPC events. We elaborate the details of each step as below.

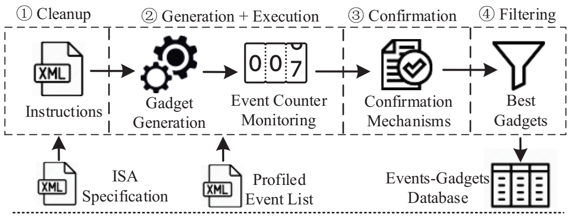


Fig. 5: Workflow of Event Fuzzer.

C. Instruction Cleanup

We first sort out all possible instructions for the target ISA. Note that this is a one-time step, and the cleaned list can be used to fuzze all events. This paper mainly focuses on the x86 architectures, but the methodology is applicable to other ISA (e.g., ARM) as well. To this end, we obtain a machine-readable x86 instruction list (namely the ISA specification) from uops.info [18]. The list extends each instruction with additional attributes (e.g., effective operand), resulting a large number of instruction variants. It also gives comprehensive information about each instruction variant, e.g., extension or category, which will be used for filtering in Section VI-F.

This ISA specification list contains many illegal instructions which cannot be executed on the given microarchitecture. To remove those instructions, we transfer the ISA specification to an assembly file, and test each instruction. The instructions that cannot complete normally will be excluded from the list. This process significantly reduces the number of instructions in the assembly file. For both Intel and AMD processors, only a small portion (24.16% and 24.31%) of instruction variants are legal. The distribution of faults in the test is similar between two processors, where the majority (98.84% and 98.69%) of the faults are caused by illegal instructions.

D. Code Generation and Execution

This step aims to generate the instruction gadget that can change the given HPC event to the state S_1. Recall that the instruction gadget consists of a reset sequence and trigger sequence. We randomly sample instructions from the cleaned list to form the sequences, and test their impact on the HPC event. To reduce the fuzzing complexity, we select one instruction for each sequence, and the fuzzing results confirm that this is enough. Our methodology can be easily extended to multi-instruction sequences with larger search spaces, which will be considered as future work.

We adopt the `RDPMC` instruction to read HPC values before and after the gadget to measure the corresponding changes. An increased count value indicates that the gadget may affect the monitoring HPC event. We take several techniques to make the measurement accurate and stable. (1) To reduce the system noise caused by external factors (e.g., interrupts), we properly configure the operating system environment for code execution. By pinning the process to a specific CPU core, we can prevent core transitions from affecting the measurement of HPC values. Besides, we also isolate this entire physical core (e.g., using the Linux kernel parameter `isolcpus`) to ensure that the process is not interrupted by the scheduler. (2) To avoid data corruptions caused by running the instruction gadgets, the code is placed in a dedicated page with the address space between a special prolog and epilog. The prolog saves all callee-saved registers, and creates one page of scratch space on the stack in case some instructions may trash stack values. Furthermore, it initializes all registers that will be used as memory operands to the address of a pre-allocated writable data page. This prevents the corruption of process memory and ensures that executed instructions access the same memory page. The epilog restores the registers and stack state, so that the architectural change can be reverted. (3) To ensure the correct measurement of HPC register values, we inject serializing instructions (e.g., `CPUID`) around the code to regulate the execution flow.

E. Result Confirmation

This step further validates if a gadget reported by the above step is indeed an obfuscating factor to the given HPC event. We analyze the identified gadgets to eliminate other side effects that can also affect the HPC event values, e.g., unreliable reset sequence whose side effects act as the trigger sequence (Challenge C5) or dirty state inherited from previous executions (Challenge C6). To remove those incorrect gadgets, we propose the following mechanisms.

Multiple executions. As the external factors (e.g., hardware interrupts) can disturb the counts of HPC events, we opt to run the same gadget for multiple times and take the median of measured values. The number of repeated executions sets a trade-off between the fuzzing efficiency and confirmation accuracy: more repetitions increase the confidence of the confirmed results, but also cause longer fuzzing time. In our implementation, we set this parameter to 10, which is proved to be an appropriate value for balancing the trade-off.

Repeated triggers. To ensure the HPC value change is indeed caused by the trigger sequence, rather than other undesired side effects of the reset sequence, we also execute the code with only the reset sequence (cold path), in addition to the one with both reset and trigger sequences (hot path), as shown in Fig. 6. In each path, we repeat the instruction sequence(s) for R times. The median of the measured count changes is denoted as v_1 and v_2, and the cumulative count changes are denoted as V_1 and V_2, respectively for the cold and hot paths. When these values meet the constraints: $V_2 - V_1 = (1 - \lambda_1)R(v_2 - v_1)$ and $V_2 > \lambda_2 V_1$, we confirm that the event value change is mainly caused by the trigger sequence, i.e., transitioning the state to S_1, and the reset sequence can actually change the

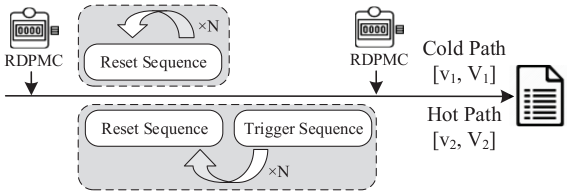


Fig. 6: Execution of repeated triggers.

state back to S_0 in each execution. Our implementation sets λ_1 as a range of [-0.2, 0.2] and $\lambda_2 = 10$.

Gadgets reordering. In order to perform fuzzing as fast as possible, we execute a generated gadget shortly after the previous one, which can cause the dirty state and affect the measurement. To address the issue, we reorder all the gadgets randomly and repeat the executions. We ignore those gadgets that have different behaviors in the reordered test. This can mitigate the influence of repetitive dirty states and remove incorrect candidates with cross-validation.

F. Gadgets Filtering

For a specific HPC event, there can be different instruction sequence gadgets that can disturb its count value. In practice, certain events like the `Retired Instruction` can be affected by nearly all instruction executions. Hence, we need to filter the confirmed gadgets from the previous step and cluster them into groups with the same features. This is achieved by analyzing the properties of reset sequences and trigger sequences, including the extension and ISA (e.g., `BASE` or `X87-FPU`) to which the instruction belongs, and the general category (e.g., arithmetic or logical) of the instruction. The intuition of such scheme is that these properties can strongly indicate the root cause of the executed instructions in the underlying microarchitectural level. This step can considerably reduce the number of reported gadgets, and alleviate the burden of the following analysis. Besides, we also extract the gadget that causes the highest value change for each HPC event, as it can lead to larger disturbance to the HPC monitoring with fewer instructions executed.

VII. Event Obfuscator

A. Challenges and Insight

To achieve the defense against HPC side channels, we need to tackle two challenges: (1) How to provide provable privacy guarantee for the HPC event obfuscation? (2) How to defeat the attack with minimal introduced noise and performance overhead? The key insight of our methodology is to model the HPC side-channel defense as a differential privacy problem. Previous works have proposed similar methods to mitigate storage side channels [74] and streaming traffic leakages [80], showing the security of systems injected with differential privacy noise can be guaranteed with theoretical proof.

Let x denote the captured HPC leakage trace and $x[t]$ denote a slice of leakage values at time t. Specifically, to prevent information leakage, we change the HPC measurement from $x[t]$ to $\widetilde{x}[t] = x[t] + r_t$, where r_t denotes the random noise following specific distributions. With the injected noise (i.e., instruction sequence gadget) by the victim VM, the malicious hypervisor cannot distinguish the actual behaviors from $\widetilde{x}[t]$. The scale of the random noise r_t can dominate the defense effectiveness, and is also restricted by the VM performance: larger amount of random noise can improve the privacy at the cost of extra performance overhead. Therefore, we leverage the differential privacy principle to theoretically identify the minimal noise under a given privacy budget, which can reduce the impact on the VM performance.

B. Differential Privacy Mechanisms

We describe two mechanisms to generate the random noise guided by differential privacy.

Laplace Mechanism. This is the most fundamental mechanism for differential privacy, which is achieved by adding controlled noise from the Laplace distribution. As a symmetric version of the exponential distribution, the Laplace distribution can be represented by its Probability Density Function (PDF): $\texttt{Lap}(b) = \frac{1}{2b}exp(-\frac{|x-\mu|}{b})$, where μ is the location and b is the scale. Consider a Laplace distribution with $\mu = 0$, $b = \frac{\triangle_{x[t]}}{\epsilon}$, and $\triangle_{x[t]} = \max\limits_{(x[t],x[t]' \in \mathcal{X})} |x[t] - x[t]'|$, where $x[t]$ and $x[t]'$ are two adjacent series at time slice t. For each sequence x in the monitored HPC sequence set $\mathcal{X}$, the Laplace mechanism computes a noisy sequence $\widetilde{x}$ as follows: $\widetilde{x}[t] = x[t] + r_t, r_t \sim \texttt{Lap}(\frac{\triangle_{x[t]}}{\epsilon})$. For simplicity, we set $\triangle_{x[t]}$ to 1, as the sequence data have been normalized. We have the following theorem:

Theorem 1. *The Laplace mechanism* $\mathcal{A}(x[t]) = x[t] + \texttt{Lap}(\frac{\triangle_{x[t]}}{\epsilon})$ *satisfies* ϵ*-DP.*

Proof. Let r_t be the noise injected to $x[t]$, i.e., $r_t \sim \texttt{Lap}(\frac{\triangle_{x[t]}}{\epsilon})$. We have

$$\mathbb{P}(\mathcal{A}(x[t]) = Z) = \frac{\epsilon}{2\triangle_{x[t]}} \exp\big(\frac{-\epsilon|r_t - x[t]|}{\triangle_{x[t]}}\big). \tag{2}$$

Similarly, $\mathbb{P}(\mathcal{A}(x[t]') = Z) = \frac{\epsilon}{2\triangle_{x[t]}} \exp\big(\frac{-\epsilon|r_t - x[t]'|}{\triangle_{x[t]}}\big)$. Thus,

$$\begin{aligned}\frac{\mathbb{P}(\mathcal{A}(x[t]) = Z)}{\mathbb{P}(\mathcal{A}(x[t]') = Z)} &= \exp\big(\frac{\epsilon(|r_t - x[t]'| - |r_t - x[t]|)}{\triangle_{x[t]}}\big) \\ &\leq \exp\big(\frac{\epsilon(|x[t] - x[t]'|)}{\triangle_{x[t]}}\big) = \exp(\epsilon).\end{aligned} \tag{3}$$

□

***d** Mechanism.** This mechanism is extended from Chan et al. [23] and a particular distance metric d^* is used to achieve d-privacy. Assume x and x' are two HPC event sequences, the d^* metric is defined as: $d^*(x, x') = \sum_{t \geq 1} |(x[t] - x[t-1]) - (x'[t] - x'[t-1])|$. Let $\mathbb{N}$ denote the natural numbers and $D(t) \in \mathbb{N}$ denote the largest power of two that divides t, i.e., $D(t) = 2^j$ if and only if $2^j \mid t$ and $2^{j+1} \nmid t$. The d* mechanism computes the noisy $\widetilde{x}[t]$ as follows: $\widetilde{x}[t] = \widetilde{x}[G(t)] + (x[t] - x[G(t)]) + r_t$, where

$$G(t) = \begin{cases} 0 & \text{if } t = 1 \\ t/2 & \text{if } t = D(t) \geq 2 \\ t - D(t) & \text{if } t > D(t) \end{cases} \tag{4}$$

$$r_t \sim \begin{cases} \texttt{Lap}(\frac{1}{\epsilon}) & \text{if } t = D(t) \\ \texttt{Lap}(\frac{\lfloor log_2 t \rfloor}{\epsilon}) & \text{otherwise} \end{cases} \tag{5}$$

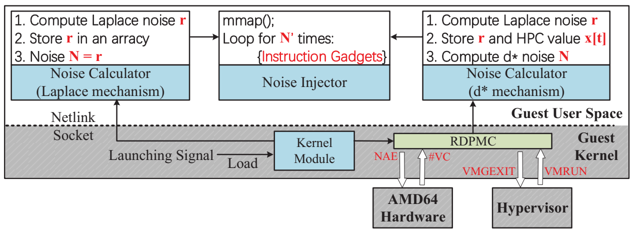


Fig. 7: Workflow of Event Obfuscator.

Theorem 2. *The* d^* *mechanism satisfies* $(d^*, 2\epsilon)$*-privacy.*

Proof. It was proven by Xiao et al. [74]. We omit the proof details here due to the page limit. □

Comparisons. The Laplace mechanism is relatively simple and exhibits acceptable privacy guarantee. Furthermore, it suits for a much stricter threat model, where the malicious host even controls and manipulates the calls reading the HPCs. In comparison, for the d^* mechanism, the noise added to each $x[t]$ is closely related to its previous sequences. This intuitively increases the randomness between adjacent sequences, thereby providing better privacy guarantee. However, it is not well suitable for systems requiring high real-time processing speed. Besides, given the limited number of available HPC registers on the processor, the number of concurrently protected events is also restricted. Therefore, d^* mechanism is better suited for reinforcing protection for multiple critical HPC events. We implement both mechanisms in `Aegis` and experimentally discuss their merits and demerits in Sec.VIII. In practical scenarios, customers can choose an appropriate strategy according to the actual system conditions and demands.

C. Design Details

We implement the online Event Obfuscator as a portable software suite in the victim VM. It is triggered by the cloud customer whenever the critical applications to be protected are launched. Fig. 7 shows the workflow of our implementation. It consists of two components: (1) a *kernel module* is used to launch the protection service and monitor the HPC values for the computation of d^*-noise in the d^* mechanism; (2) a *userspace daemon* is used to calculate the amount of random noise and inject it into the execution flow of the VM.

Specifically, the kernel module mainly plays the role of a controller. After receiving the launching signal from the user, it wakes up the userspace daemon. If `Aegis` adopts the d^* mechanism, the module also needs to monitor the real-time HPC event values x with the `RDPMC` instruction. The recorded HPC values are sent to the userspace daemon with the `netlink` sockets for the subsequent noise generation, which is computation-inefficient in the kernel mode. The userspace daemon consists of two components: noise calculator and noise injector. To support high injection rates, we need to accelerate the calculation of noise value. Hence, the noise calculator maintains a buffer to store the precomputed random noise sequence r following $\texttt{Lap}(\frac{1}{\epsilon})$ for Laplace mechanism or Eq. 5 for d^* mechanism. Note that the random number r is directly transferred from the uniform distribution in $[0, 1]$, while using library APIs introduces much longer latency. For the Laplace mechanism, the calculated r is the noise. For the d^* mechanism, HPC values $x[t]$ sent from the kernel module are stored to calculate noise.

After the calculation, the noise injector is called to add the identified amount of noise, i.e., instruction gadgets, into the VM's execution flow. In theory, we can obfuscate each vulnerable HPC events by injecting its corresponding noise gadgets. However, given it usually has hundreds of vulnerable events, such method also introduces hundreds of injected gadgets, which may lead to large performance overhead. In the practical implementation, the identified gadget sets for various HPC events usually have intersections, which allow a gadget to interfere more than one event. For example, gadgets corresponding to `HW_CACHE_L1D:WRITE` can induce changes in almost cache-related events. Therefore, the optimal solution is to extract the smallest gadget set that can cover the most vulnerable events. In our settings, to cover all 137 vulnerable HPC events, we only require 43 instruction gadgets, which hence significantly reduces the overhead invoked by instruction injection. By stacking these gadgets together, we conduct a code segment that executes repeatedly to inject noise to vulnerable HPC events. The number of repetitions of the code execution is determined by the noise value computed from the noise calculator. This will not affect the original VM execution, and the incurred performance cost is acceptable under a satisfactory privacy budget.

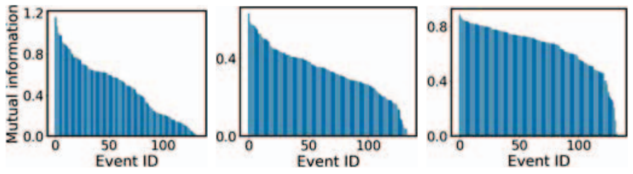
In our implementation, we explicitly pin Event Obfuscator and protected applications to the same virtual CPU core, so that the malicious hypervisor cannot arrange them to different physical cores, and cannot distinguish to bypass our defense. Note that with the protection of SEV, processes on the same virtual CPU core are indistinguishable for the hypervisor even it owns the highest privilege.

VIII. Evaluation

A. Profiling Evaluation

With the warm-up profiling, we can compact the number of vulnerable HPC events from M to N, where M denotes the number of all available HPC events and N is the number of filtered vulnerable events. In our settings, the value of M is 6166 for Intel CPU and 1903 for AMD CPU. Hence, the time spent on this step is $T_W = (M \times t_w \times 2)/C$, where C denotes the number of available HPC registers that support concurrent monitoring (e.g., $C = 4$ in our testbed), and t_w denotes the monitoring time for each HPC event (e.g., 1 second in our implementations). Note that the profiling of M events should be performed twice to compare their counts. As a result, the warm-up profiling takes 0.85 hours on Intel CPU and 0.26 hours on AMD CPU.

To further estimate the vulnerability of filtered HPC events, we compute the *information gain* induced by the event values to infer the application secret. For each specified secret (i.e., 45 websites, standard keystrokes or 30 DNN models used in Section III) of the target application, given a specific HPC event, the application is repeatedly executed for 100 times



(a) Website access. (b) Keystroke. (c) Model execution.

Fig. 8: The mutual information of each HPC event.

to launch the secret, e.g., visiting a selected website. Then we repeat the operation for each HPC event profiled from the warm-up profiling, which finally generates $N \times S \times 100$ leakage traces, where S is the size of the specified secret set. Therefore, the time spent on this profiling step can be calculated as $T_P = (N \times S \times 100 \times t_p)/C$, where t_p is the profiling time for each event and still 1 second in our settings. Specially, the three target applications in our experiments take 42.81 hours, 9.51hours and 28.54 hours, respectively. Note that this time cost has large space for optimization, e.g., the number of repeated executions can be reduced to 10 times, which is enough for a rough analysis.

The *mutual information* is computed following Eq. 1. Fig. 8 shows the *mutual information* of each HPC event for website accesses, keystrokes and DNN model executions occurred in an SEV VM. A higher mutual information reflects higher relevance between the event and the victim application, i.e., the event is a more vulnerable attack surface. We can see that Fig. 8a and 8b drop much faster than Fig. 8c, meaning that for the DNN model execution attack there are more vulnerable HPC events. It is because DNN models invoke more interactions with the underlying hardware, e.g., memory accesses and logical calculation, which lead to more HPC leakages.

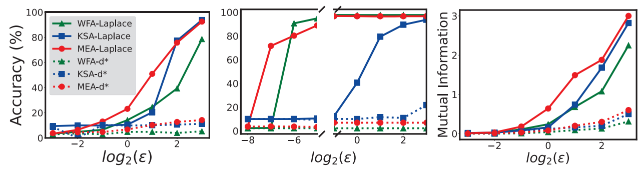
B. Fuzzing Evaluation

We run Event Fuzzer on two processors (i.e., Intel Xeon E5-1650 and AMD EPYC 7252) and evaluate the performance of the fuzzing process. For the Intel CPU, 3386 instructions remain after the cleanup step, leading to a total of $3386^2 = 11,464,996$ possible instruction gadgets. These gadgets are repeatedly fuzzed for each profiled HPC event obtained from Application Profiler, thus performing 738 repetitions. A full fuzzing run terminated in 9.3 hours, which results in a throughput of 253,314 gadgets per second. For the AMD processor, while it similarly has $3407^2 = 11,607,649$ usable gadgets, the fuzzing process can be completed in just 2.2 hours, as the processor has much less (i.e., 137) usable HPC events for executing repetitions. The throughput of fuzzing is similar, which reaches 235,449 gadgets per second. Table III shows the detailed time consumption for each step in the fuzzing process. It can be seen that the generation and execution of gadgets take the most amount of fuzzing time, while other three steps can be achieved in a short time.

After the filtering step, most HPC events only correspond to hundreds or even dozens of usable gadgets, but there are still multiple events corresponding to numerous (e.g., thousands of) gadgets. Availability of more gadgets means we have more

CPU Processor	Time Consumption (seconds)			
	Cleanup	Generation + Execution	Confirmation	Filtering
Intel Xeon E5-1650	< 1	33210	132	60
AMD EPYC 7252	< 1	7791	29	18

TABLE III: Time consumption for each fuzzing step.



(a) Train w/o noise (b) Train with noise (c) Real mutual info

Fig. 9: Impact of ϵ on various attacks.

choices to obfuscate the HPC event, but it also introduces higher analysis complexity. In the Intel CPU, the mean and median value of the gadgets for all events are 892 and 505, respectively. The event with the most fuzzed gadgets (i.e., 9934) is `MEM_LOAD_UOPS_RETIRED:L1_HIT`. In the AMD CPU, the mean and median are 617 and 440, where the event with the most usable gadgets (i.e., 6219) is `RETIRED_MMX_FP_ INSTRUCTIONS:SSE_INSTR`. It can be seen that events related to the instruction numbers usually tend to be more vulnerable, as they can be modified by most executing instructions. This observation is matched with the profiled results from the above Application Profiler.

C. Defense Effectiveness

To evaluate the effectiveness of `Aegis` against HPC side-channel attacks, we vary the privacy budget ϵ, and measure its impact on the attack accuracy. The experiment settings are the same as shown in Section III. As the number of injected instruction gadgets cannot be negative, each noise element is truncated by a clip bound of $[0, B_u]$, where the upper bound B_u is determined empirically based on the profiling of HPC events. For example, we set $B_u = 2e4$ for the event `RETIRED_UOPS`. According to the attacker's capabilities, we consider the following two scenarios:

First, the attacker trains the attack model on clean data collected from the template VM, as the victim VM is a black box for him. Most realistic side-channel attacks follow this case, including our attack cases in Sec.III. Figure 9a shows the impact of privacy budget ϵ on the accuracy of three attacks for the Laplace and d^* mechanisms. The value of ϵ is set as $[2^{-3}, 2^{-2}, ..., 2^3]$. From this figure, we can summarize four remarks: (1) both mechanisms can effectively mitigate HPC side-channel attacks, which decrease the attack accuracy from $> 90\%$ to 2%; (2) a larger ϵ leads to a higher attack accuracy, since it adds less noise; (3) with the same ϵ, d^* mechanism can provide stronger privacy guarantee, especially for large ϵ (e.g., $\epsilon \geq 2^0$); (4) WFA and KSA are more sensitive to the noise than MEA, as their accuracy decrease much faster with the increase of noise. It may be because website accesses and keystrokes have more similar leakage patterns that are easier to be affected by the injected noise.

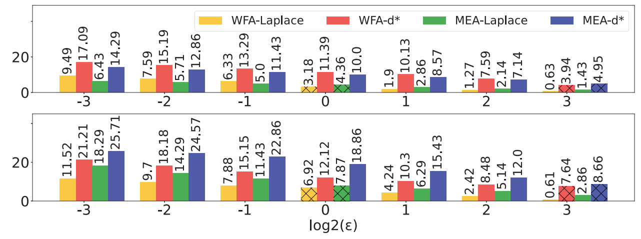


Fig. 10: Impact of ϵ on the latency overhead (upper) and CPU usage (lower).

Second, we consider a more powerful attacker, who knows the defense details adopted in the VM (e.g., the privacy mechanism, the value of ϵ). In this case, the attacker can train his attack model with noisy data to increase the model's robustness in the exploitation phase. Figure 9b shows the attack accuracy of such models under the defense with our `Aegis`, where $\epsilon \in [2^{-8}, 2^{-7}, ..., 2^3]$. We observe that d^* mechanism can still defeat these advanced attacks well, while Laplace mechanism requires a smaller ϵ to suppress the attack accuracy. We conclude that by slightly decreasing the privacy budget, `Aegis` can still mitigate HPC attacks even when the attack model is trained in a more robust approach. Given it needs to inject more noise, the defense would induce much larger overhead. However, as such an attacker is too strong and is nearly impossible in the practical scenario, we would not consider it in our following efficiency analysis.

Note that our defense is effective for all machine learning based attack models, as the correlation between the HPC side channels and the running secrets are significantly reduced. Fig. 9c shows the value of real mutual information $I(\mathcal{X}; \mathcal{X}')$ between the clean HPC leakage traces $\mathcal{X}$ and the noised HPC leakage traces $\mathcal{X}'$ under different size of noise. It can be seen that with the increased noise (i.e., smaller ϵ), the value of $I(\mathcal{X}; \mathcal{X}')$ keeps decreasing to a small value. Hence, the mutual information $I(\mathcal{X}'; \mathcal{Y})$ between the noised trace $\mathcal{X}'$ and the secret $\mathcal{Y}$ also decreases equivalently [30]. Therefore, although we only show three attack cases in this paper, the effectiveness of our method can be well guaranteed on other attacks.

D. Defense Efficiency

As keystrokes are transient actions that only take negligible resources, we mainly focus on the defense against WFA and MEA. We evaluate the efficiency of `Aegis` from latency overhead and CPU usage. Figure 10 shows the overhead invoked by various mechanisms.

First, we assess the impact of `Aegis` on the performance of the protected applications in the VM. For each test, we continually access 45 websites or run 30 DNN models to measure the execution time. Figure 10 shows the average time of loading a website and running a model inference under two DP mechanisms with different ϵ values in `Aegis`. The website loading time is recorded by the built-in development tool in the Chrome browser, while the DNN model inference time is measured by a timer written in python. From the figure, we can find that (1) smaller ϵ leads to longer execution

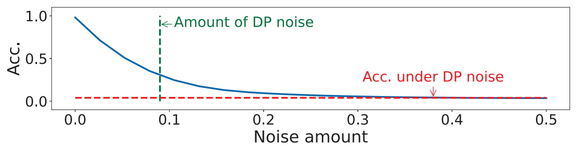


Fig. 11: Attack accuracy with the random noise

time, as more noisy instructions are injected; (2) under the same ϵ, d^* mechanism induces more overhead than Laplace mechanism. To guarantee privacy against attacks, we select $\epsilon = 2^0$ for Laplace mechanism (marked with shadow), which causes 3.18% and 4.36% overhead on the execution time of website accesses and model inferences. For d^* mechanism, $\epsilon = 2^3$ is enough to mitigate attacks, where the execution time increases by 3.94% and 4.95% for the two applications. Hence, the time overhead introduced by `Aegis` is slight.

Second, we measure the impact of `Aegis` on the VM resource consumption by monitoring its CPU utilization. As `Aegis` injects instruction gadgets into the VM's executions, it may consume extra CPU resources with certain overhead. According to the basic feature of differential privacy, i.e., the statistical characteristics of noisy result $\widetilde{x}$ are similar to the original data x, we speculate such cost penalty caused by `Aegis` should be small. Figure 10 shows the CPU usage of the victim VM under two DP mechanisms. The VM's CPU usage is measured from the host with the `top` tool every 0.2 seconds and each usage value is the average of 5 experiments. Specifically, `Aegis` has smaller influence on the website accesses, which may be because they involve fewer CPU interactions. With Laplace mechanism, the CPU usage penalty introduced by `Aegis` is 6.92% and 7.87% for website accesses and model inferences. For d^* mechanism, it is 7.64% and 8.66%. Hence, `Aegis` only leads to a small CPU overhead.

IX. Discussion

A. Alternative Defense Strategies

Constant HPC output. Setting the HPC output to a constant can also mask HPC side channels. However, such method is actually impractical in the real implementation. To make the HPC output as a constant, we must add more counts (i.e., noise) to the original HPC values, until reaching the peak HPC value p in the leakage trace. It introduces much more noise than our solution. For example, to obfuscate the leakage of accessing *www.youtube.com*, setting the HPC event (e.g., `DATA_CACHE_REFILLS_FROM_SYSTEM`) values to a constant p totally introduces 595,371,616 event counts, while our Laplace mechanism only introduces 33,090,214 events. Hence, constant HPC output invokes nearly 18× more noise, which is an overkill defense.

Random noise. Instead of the DP noise, simply adding random noise can also obfuscate the HPC leakage. However, this strategy has two limitations. First, such random noise cannot provide a provable privacy guarantee, which still exposes the encrypted VM to the leakage threat. Second, this strategy usually introduces more noise than DP mechanisms to achieve the same privacy protection. We use different scales of random noise to obfuscate the HPC leakage and the attack accuracy is shown in Fig. 11. The x-axis denotes the upper bound of random noise in the range of $[0, 0.5] \times p$, where p denotes the peak HPC value in the leakage trace. In the figure, we also label the amount of Laplace noise ($\epsilon = 2^0$) required to effectively defeat the attack, i.e., decreasing the attack accuracy to $< 5\%$. With the same amount of injected noise, random noise mechanism can only decrease the attack accuracy to 32%, which is much higher than DP mechanisms. Besides, we also show the attack accuracy under the effective defense with the DP noise. To achieve the same accuracy, the upper bound of random noise needs to be at least $0.4p$, which introduces $4.37\times$ more noise than the Laplace mechanism.

Isolating guest HPCs. The root cause of HPC side channels is the sharing of HPC registers between the guest and host. Hence, isolating guest HPCs from the malicious host can fundamentally eliminate such side channels, as demonstrated by Intel TDX. However, since this defense necessitates specific hardware modifications, it is not feasible for existing SEV-based systems, which motivates our software-based solution. While our work can effectively and efficiently mitigate HPC side channels on SEV VMs, it still introduces extra overhead and is not the optimal solution for defending such hardware side channels. Hence, we advocate for AMD to enhance their hardware design, which is a promising alternative of our software design to address this issue from the root.

B. Analysis with Multiple Tries

The injected noise to the HPC leakage traces actually can be averaged out by the attacker by obtaining a group of leakage traces corresponding to the same secret [30]. However, in the practical scenario, the adversarial hypervisor cannot force the user VM to repeatedly run the same secret for many times. Hence, the adversary cannot collect multiple leakage traces with the same secret to remove the impact of injected noise. Besides, even if the adversary can collect such multiple traces, the attack can be easily defeated by attaching a constant secret-dependent noise to the execution so that the adversary cannot average out the injected noise through analyzing multiple leakage traces. Such method can well protect the user secrets and also reduce the overhead of noise generation.

X. Conclusion

In this paper, we propose `Aegis`, a unified framework that can mitigate HPC side-channel attacks with a provable privacy guarantee and minimal performance overhead. It also comprehensively profiles the vulnerability of HPC events, and automatically demystifies the correlations between the specific instruction gadgets and HPC event statistics. Our future works aim to study the defense effect of noise gadgets with more instructions, and investigate the effectiveness of `Aegis` on more fine-grained attacks, e.g., stealing cryptographic keys. Besides, we also tend to generalize our framework to more micro-architectural attacks, e.g., cache and memory side channels, Meltdown [53] and Spectre [48] attacks, or the latest voltage glitching attack [22].

XI. Acknowlodgement

This research/project is supported by the National Research Foundation, Singapore under its AI Singapore Programme (AISG Award No: AISG2-PhD-2021-08-023[T]), the Cyber Security Agency of Singapore under its National Cybersecurity Research & Development Programme (Development of Secured Components & Systems in Emerging Technologies through Hardware & Software Evaluation <NRF-NCR25-DeSNTU-0001>). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the view of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

References

[1] Alexa top 1000 most visited websites. [Online]. https://www.htmlstrip.com/alexa-top-1000-most-visited-websites.

[2] Amazon ec2 user guide. [Online]. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/sev-snp.html.

[3] Amd expands confidential computing presence on google cloud. [Online]. https://www.amd.com/en/press-releases/2022-05-25-amd-expands-confidential-computing-presence-google-cloud.

[4] Amd secure encrypted virtualization (sev) github repository. [Online]. https://github.com/AMDESE/AMDSEV.

[5] Amd64 architecture programmer's manual, volume 2: System programming. [Online]. https://www.amd.com/system/files/TechDocs/24593.pdf.

[6] Arm confidential compute architecture. [Online]. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture.

[7] Azure confidential vm options. [Online]. https://learn.microsoft.com/en-us/azure/confidential-computing/virtual-machine-solutions.

[8] Azure linux virtual machines pricing. [Online]. https://azure.microsoft.com/en-gb/pricing/details/virtual-machines/linux/.

[9] Confidential computing: an aws perspective. [Online]. https://aws.amazon.com/blogs/security/confidential-computing-an-aws-perspective/.

[10] Google cloud confidential vm overview. [Online]. https://cloud.google.com/confidential-computing/confidential-vm/docs/confidential-vm-overview.

[11] Google confidential vm supported configurations. [Online]. https://cloud.google.com/confidential-computing/confidential-vm/docs/supported-configurations.

[12] Intel tdx module specification 1.5. [Online]. https://cdrdv2.intel.com/v1/dl/getContent/733575.

[13] Intel® trust domain extensions (intel® tdx). [Online]. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html.

[14] Linux kernel profiling with perf: multiplexing and scaling events. [Online]. https://perf.wiki.kernel.org/index.php/Tutorial#multiplexing_and_scaling_events.

[15] perfmon2 libpfm-4.11.0 released. [Online]. http://perfmon2.sourceforge.net/.

[16] Pytorch models and pretrained weights. [Online]. https://pytorch.org/vision/stable/models.html.

[17] Ubuntu manpage for xdotool. [Online]. https://manpages.ubuntu.com/manpages/trusty/man1/xdotool.1.html.

[18] Andreas Abel and Jan Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 673–686, 2019.

[19] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.

[20] Sarani Bhattacharya and Debdeep Mukhopadhyay. Who watches the watchmen?: Utilizing performance monitors for compromising keys of rsa on intel platforms. In *Proceedings of CHES*, pages 248–266. Springer, 2015.

[21] Tim Blazytko, Matt Bishop, Cornelius Aschermann, Justin Cappos, Moritz Schlögel, Nadia Korshun, Ali Abbasi, Marco Schweighauser, Sebastian Schinzel, Sergej Schumilo, et al. {GRIMOIRE}: Synthesizing structure while fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1985–2002, 2019.

[22] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. One glitch to rule them all: Fault injection attacks against amd's secure encrypted virtualization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2875–2889, 2021.

[23] T-H Hubert Chan, Elaine Shi, and et al. Private and continual release of statistics. *ACM Transactions on Information and System Security*, 2011.

[24] Konstantinos Chatzikokolakis, Miguel E Andrés, and et al. Broadening the scope of differential privacy using metrics. In *Proceedings of PETs*, 2013.

[25] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with déjá vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 7–18, 2017.

[26] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[27] Debayan Das, Anupam Golder, Josef Danial, Santosh Ghosh, Arijit Raychowdhury, and Shreyas Sen. X-deepsca: Cross-device deep learning side channel attack. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[28] Sanjeev Das, Bihuan Chen, and et al. Ropsentry: Runtime defense against rop attacks using hardware performance counters. *Computers & Security*, 73:374–388, 2018.

[29] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. Sok: The challenges, pitfalls, and perils of using hardware performance counters for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 20–38. IEEE, 2019.

[30] Eloi de Chérisey, Sylvain Guilley, Olivier Rioul, and Pablo Piantanida. Best information is most successful. *Cryptology ePrint Archive*, 2019.

[31] Wladimir De la Cadena, Asya Mitseva, and et al. Trafficsliver: Fighting website fingerprinting attacks with traffic splitting. In *Proceedings of ACM CCS*, 2020.

[32] Christopher Domas. Breaking the x86 isa. *Black Hat*, 2017.

[33] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L Cox, and Sandhya Dwarkadas. Shielding software from privileged side-channel attacks. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1441–1458, 2018.

[34] Cynthia Dwork. Differential privacy. In *Proceedings of ICALP*, pages 1–12. Springer, 2006.

[35] Ramanathan Gnanadesikan and Martin B Wilk. Probability plotting methods for the analysis of data. *Biometrika*, 55(1):1–17, 1968.

[36] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*, pages 369–376, 2006.

[37] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Security Symposium*, pages 217–233, 2017.

[38] Berk Gulmezoglu, Andreas Zankl, and et al. Perfweb: How to violate web privacy with hardware performance events. In *Proceedings of ESORICS*, pages 80–97, 2017.

[39] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *NDSS*, 2019.

[40] Felicitas Hetzelt and Robert Buhren. Security analysis of encrypted virtual machines. In *ACM SIGPLAN Notices*, 2017.

[41] Sam Hocevar. Zzuf. [Online]. https://github.com/samhocevar/zzuf/.

[42] Shohreh Hosseinzadeh, Hans Liljestrand, Ville Leppänen, and Andrew Paverd. Mitigating branch-shadowing attacks on intel sgx using control flow randomization. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, pages 42–47, 2018.

[43] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[44] Suman Jana and Vitaly Shmatikov. Memento: Learning secrets from process footprints. In *2012 IEEE Symposium on Security and Privacy*, pages 143–157. IEEE, 2012.

[45] Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Trusted execution environments: properties, applications, and challenges. *IEEE Security & Privacy*, 18(2):56–60, 2020.

[46] David Kaplan. Protecting vm register state with sev-es. *White paper*, 2017.

[47] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. *White paper*, 2016.

[48] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[49] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. In *USENIX Security Symposium*, volume 19, pages 16–18, 2017.

[50] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. Crossline: Breaking" security-by-crash" based memory isolation in amd sev. In *Proceedings of ACM CCS*, 2021.

[51] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. Exploiting unprotected I/O operations in AMD's secure encrypted virtualization. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1257–1272, 2019.

[52] Moritz Lipp, Daniel Gruss, and et al. Armageddon: Cache attacks on mobile devices. In *USENIX Security Symposium*, 2016.

[53] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, 2018.

[54] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*, pages 605–622. IEEE, 2015.

[55] Xiaoxuan Lou, Shangwei Guo, Jiwei Li, Yaoxin Wu, and Tianwei Zhang. Naspy: Automated extraction of automated machine learning models. In *International Conference on Learning Representations*, 2021.

[56] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2019.

[57] Mathias Morbitzer, Manuel Huber, and et al. Severed: Subverting amd's virtual machine encryption. In *Proceedings of EuroSec*, 2018.

[58] Mathias Morbitzer, Sergej Proskurin, Martin Radev, Marko Dorfhuber, and Erick Quintanar Salas. Severity: Code injection attacks against encrypted virtual machines. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 444–455. IEEE, 2021.

[59] Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin philosophical magazine and journal of science*, 2(11):559–572, 1901.

[60] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: what it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/Ispa*, volume 1, pages 57–64. IEEE, 2015.

[61] AMD SEV-SNP. Strengthening vm isolation with integrity protection and more. *White Paper, January*, 2020.

[62] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *NDSS*, 2017.

[63] Anatoly Shusterman, Lachlan Kang, and et al. Robust website fingerprinting through the cache occupancy channel. In *USENIX Security Symposium*, 2019.

[64] Payap Sirinam, Mohsen Imani, and et al. Deep fingerprinting: Undermining website fingerprinting defenses with deep learning. In *Proceedings of ACM CCS*, 2018.

[65] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on SSH. In *10th USENIX Security Symposium (USENIX Security 01)*, 2001.

[66] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[67] M Caner Tol, Berk Gulmezoglu, Koray Yurtseven, and Berk Sunar. Fastspec: Scalable generation and detection of spectre gadgets using neural embeddings. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 616–632. IEEE, 2021.

[68] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *USENIX Security Symposium*, pages 601–618, 2016.

[69] Leif Uhsadel, Andy Georges, and Ingrid Verbauwhede. Exploiting hardware performance counters. In *Proceedings of FDTC-Workshop*, 2008.

[70] Xueyang Wang and Ramesh Karri. Reusing hardware performance counters to detect and identify kernel control-flow modifying rootkits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(3):485–498, 2015.

[71] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. Osiris: Automated discovery of microarchitectural side channels. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1415–1432, 2021.

[72] Jan Werner, Joshua Mason, and et al. The severest of them all: Inference attacks against secure virtual enclaves. In *Proceedings of ACM AsiaCCS*, 2019.

[73] Yun Xiang, Zhuangzhi Chen, Zuohui Chen, Zebin Fang, Haiyang Hao, Jinyin Chen, Yi Liu, Zhefu Wu, Qi Xuan, and Xiaoniu Yang. Open dnn box by power side-channel attack. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 67(11):2717–2721, 2020.

[74] Qiuyu Xiao, Michael K Reiter, and Yinqian Zhang. Mitigating storage side channels using statistical privacy mechanisms. In *Proceedings of ACM CCS*, pages 1582–1594, 2015.

[75] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. Speechminer: A framework for investigating and measuring speculative execution vulnerabilities. *arXiv preprint arXiv:1912.00329*, 2019.

[76] Mengjia Yan, Christopher W Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *USENIX Security Symposium*, 2020.

[77] Ning Zhang, Kun Sun, and et al. Truspy: Cache side-channel information leakage from the secure world on arm devices. *IACR Cryptol. ePrint Arch.*, 2016:980, 2016.

[78] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. Cloudradar: A real-time side-channel attack detection system in clouds. In *Proceedings of RAID*, pages 118–140. Springer, 2016.

[79] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. Analyzing cache side channels using deep neural networks. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 174–186, 2018.

[80] Xiaokuan Zhang, Jihun Hamm, and et al. Statistical privacy for streaming traffic. In *Proceedings of NDSS*, 2019.