Noname manuscript No.
(will be inserted by the editor)

The Kingsguard
OS-level Mitigation against Cache Side-Channel Attacks using Runtime Detection

Maria Mushtaq · Muhammad Muneeb Yousaf · Muhammad Khurram Bhatti · Vianney Lapotre · Guy Gogniat

Received: date / Accepted: date

Abstract Most of the mitigation techniques against access-driven Cache Side-Channel Attacks (CSCAs) are not very effective. This is mainly because most mitigation techniques usually protect against any given specific vulnerability of the system and do not take a system-wide approach. Moreover, they either completely remove or greatly reduce the performance benefits. Therefore, to find a security vs performance trade-off, we argue in favor of *need-based protection* in this paper, which will allow the operating system to apply mitigation only after successful *detection* of CSCAs. Thus, detection can serve as a *first line of defense* against such attacks. In this work, we propose a novel OS-level runtime detection-based mitigation mechanism, called the Kingsguard, against CSCAs in general-purpose operating systems. The proposed mechanism enhances the security and privacy capabilities of Linux as a proof of concept, and it can be widely used in commodity systems without any hardware modifications. We provide experimental validation by mitigating three state-of-the-art CSCAs on two different cryptosystems running under Linux. We have also provided results by analyzing the effect of the combination of multiple attacks running concurrently under variable system noise. Our results show that the Kingsguard can detect and mitigate known CSCAs with an accuracy of more than 99% & 95%, respectively.

Keywords Hardware Security, Linux, Intel x86, Side-Channel Attacks, Cryptanalysis, Detection, Mitigation, Machine Learning, RSA, AES, Flush+Reload, Flush+Flush, Prime+Probe.

1 Introduction

In recent years, high resolution and stealthy Side-Channel Attacks (SCAs) and their variants such as: Flush+Reload [16], Flush+Flush [15], Prime+Probe[22],

M. Mushtaq
LTCI, Télécom Paris, Institute Polytechnique de Paris, France
E-mail: maria.mushtaq@telecom-paris.fr

Spectre [19] and Meltdown [21] have completely exposed the vulnerabilities in modern computing architectures. At the software level, modern cryptographic algorithms are theoretically sound and require enormous computing power to break. However, much research work has shown that cryptosystems, such as AES, can be compromised due to the vulnerabilities of the underlying hardware on which they run [36], [15]. The SCAs do not target the cryptosystem algorithm itself. Rather, they target the underlying implementation of systems on which these cryptosystems execute [37]. The SCAs can use a variety of physical parameters, e.g., power consumption, electromagnetic radiation, memory accesses and timing patterns to extract secret keys/information [36], [15], [29], [18]. The baseline idea here is that the SCAs can analyze the variations in these parameters during the execution of cryptosystems on a particular piece of hardware and can determine the secret information used by cryptosystems based on the observed parameters. The Cache Side-Channel Attacks (CSCAs) are a special type of SCAs, in which a malicious process deduces the secret information of a victim process by observing its use of caching hardware. The inherent features that any known CSCA exploits are the cache timing and access patterns.

Despite valiant efforts, mitigation techniques against SCAs are not very effective. This is mainly because mitigation techniques usually protect against any given specific vulnerability of the system and do not take a system-wide approach. Moreover, they either completely remove or greatly reduce the performance benefits of resource sharing. In addition to this, the attacks are becoming sophisticated and stealthier [15], [13]. Thus, they overcome statically applied mitigation techniques. Therefore, on the one hand, protection against these CSCAs needs to be applied across the entire computing stack and, on the other hand, mitigation strategies must not remove the hard-earned performance benefits of computing systems. In this work, we advocate for the use of *need-based protection* mechanisms, which are imperative to effectively mitigate CSCAs without sacrificing the benefits of resource sharing. Our arguments are in favor of enhancing the capability of the Operating System (OS) by using a detection-based mitigation approach that would help the OS to apply mitigation *only* after successful detection of a CSCA. Thus, detection can serve as the *first line of defense* against such attacks. Such a solution would incur as little overhead as possible without significant performance or monetary cost. Such a solution, however, becomes very challenging in the absence of an effective detection mechanism, which needs to be highly accurate, and should incur minimum system overhead at runtime, cover a large set of attacks and be capable of early-stage detection, i.e., before the attack is completed, at the very least. Rather than applying static mitigation against CSCAs, which is constantly active and thus costly in terms of performance, a detection-based mitigation mechanism would be dynamic. It would neutralize side-channel threat as and when it happens. The following are the major contributions of this paper.

1. We propose a novel OS-level runtime detection-based mitigation mechanism, called the Kingsguard, against CSCAs, that enhances the security and privacy capabilities in a general-purpose OS.
2. We demonstrate that the Kingsguard is capable of detecting and mitigating state-of-the-art CSCAs, such as: Prime+Probe, Flush+Reload and Flush+Flush attacks on AES and RSA cryptosystems while running under Linux. We support our claims with an extensive experimental evaluation.
3. The Kingsguard is resilient to noise generated by the system under various loads. We provide results under *realistic* system load conditions. Results demonstrate the robustness & portability of our proposed mechanism.
4. We demonstrate the effectiveness of the Kingsuard on Linux OS as a proof of concept. However, the Kingsguard is scalable across other operating systems and attack vectors as well.

The rest of this paper is organized as follows. Section 2 provides related work on the detection and mitigation techniques for CSCAs. Section 3 presents the Kingsguard mitigation mechanism. Section 4 provides experiments & results for selected case studies. Section 6 concludes this paper.

2 Related Work

We provide the state of the art on SCA detection and mitigation mechanisms that is the most relevant to the proposed work in this paper.

2.1 State of the Art in Detection Mechanisms

In recent years, many researchers have demonstrated that known SCAs can be detected using different approaches [3]. Some of the OS-based detection solutions are user-level, whereas others are kernel-level. Broadly, detection mechanisms can be divided into three categories i.e., Signature-Based Mechanisms ([5], [31], [30], Anomaly-Based Mechanisms ([7], [20], [9]) and Signature- + Anomaly-Based Mechanisms ([39], [4], [10]).

The authors in [5] have proposed a mechanism to detect F+R targeting AES cryptosystems while showing good accuracy under no load conditions. Accuracy deteriorates under load conditions and the authors did not discuss the impact of a system overhead caused by the detection technique. The authors in [31] proposed a tool named *SCADET* to detect P+P attacks. Results show that *SCADET* achieves good accuracy but it lacks a discussion section on detection speed and system overhead caused by the proposed mechanism, which leaves the issue of runtime adaptability.

Another piece of work in [30] detects F+R attacks successfully but the authors did not report the performance overhead and detection speed of the proposed mechanism. The authors in [7] used Gaussian Anomaly Detection to detect P+P attacks. However, their detection technique shows good accuracy only in isolated conditions and suffers from high false positives in realistic load conditions. The *CacheShield* [9] is an unsupervised anomaly detection mechanism to inspect F+R, P+P and F+F attacks under load conditions. The

authors report that *CacheShield* offers good accuracy but it is not fast enough to detect attacks at an early stage (detects after 37% and 50% of key computation). Moreover, the proposed mechanism uses only 2 HPCs for variable attacks (running RSA, AES, ELgamal). Attacks working with slightly different behavior may not be detected by *CacheShield* due to their lack of a variable set of distinguishable and non-correlated features. The detection mechanism also relies on threshold determination for classification decisions. Once a different attack vector is introduced, the threshold may no longer be a sophisticated method to classify abnormal activity. Variation in attack behavior can lead to indistinguishable behaviors. Thus, a pre-determined threshold can vary from a given range with the introduction of new behaviors (attacks) and does not act as an intelligent mechanism to report multiple threats simultaneously. Furthermore, the detection technique does not report on performance degradation caused by this method. A technique called *SpyDetector* [20], has been proposed for the detection of F+R, F+F and P+P attacks, but the authors did not report the detection speed and overhead of *SpyDetector*. Therefore, it cannot be determined whether or not *SpyDetector* is able to perform early-stage detection at runtime (especially for stealthy attacks such as F+F) or the cost of *SpyDetector* in terms of performance. Another technique, called *CloudRadar* [39] detects P+P and F+R with good accuracy and negligible overhead in virtual environments at a predetermined threshold and sampling frequency. Results show that *CloudRadar* detects attacks in isolated conditions and no realistic scenario for the mechanism has been tested. Another approach, proposed in [4], detects cache and branch predictor based SCAs. Their experimental results show good detection accuracy, but the technique does not provide details on detection speed and the impact on performance overhead. Keeping in view the shortcomings of earlier work on runtime detection, we proposed a detection technique, called *NIGHTs-WATCH*, in [26] that couples machine learning models and HPCs to detect F+R and F+F at runtime. This research work reports good detection accuracy, low performance overhead, negligible misclassification rate and a high speed of detection. Later on, this work was extended to successfully demonstrate the detection of P+P attacks also in [25], [27]. We extended the work in [26] by training our machine learning models with three attacks collectively, namely; Flush+Flush, Prime+Probe and Flush+Reload that are running both on RSA and AES cryptosystems. We can detect and subsequently mitigate these attacks while they are running in any temporal order or simultaneously under realistic execution scenarios.

2.2 State of the Art in Scheduling-Based Mitigation Techniques

For most side-channel attacks, timing variation is important to learn the victim's interests i.e., repeated cache line accesses during a cryptographic operation. Many mitigation approaches work to counter the timing channel e.g., [6] provided a mitigation technique to limit the information revealed through the timing channel, [32] proposed an instruction-based scheduler to mitigate the timing channel, [34] proposed to eliminate fine-grained timers in Xen. But obfuscating timing information can also negatively affect other useful applica-

tions and attackers can obtain the information on victims through other useful methods such as observing the access patterns of the victim [29].
The authors in [38] proposed bystander VMs with configurable workloads to introduce noise in the victim process and to distract the attacker process from the victim's activity. But this approach works for covert channels only. There is no evidence of this solution in the case of CSCAs. Noise-based solutions work differently for each attack (based on information on the attack principle) and noise-based solutions are not always successful in obfuscating sensitive information. The authors in [23] introduced *Shuffler*, a light-weight scheduling defense mechanism which quantifies the negative impact a VM can have on other VMs and effectively limits the vulnerability probability in VMs. The *Shuffler* scheduler works effectively for Prime+Probe attacks with negligible overhead. It claims that the same principle should work for Flush+Reload attacks but no empirical evidence of its effectiveness is provided under other stealth and high resolution CSCAs. Such defenses are effective in solving a particular threat at a specific cache level, but none of the solutions mitigates the threat at all cache levels and for a set of sophisticated, stealth and high-resolution attacks under one umbrella. Moreover, such solutions introduce a significant performance penalty. This defense mechanism is particular to one approach (P+P) that can likely be exploited by more sophisticated attacks such as F+F, F+R etc. *Düppel* [41] is a mitigation technique which includes mitigation for time-shared caches such as L1 and L2, TLB and BTB. *Düppel* modifies the guest OS Kernel and does not need to change hypervisor or cloud providers. Unlike the noise producing techniques, *Düppel* repeatedly cleans the L1 cache along with the execution of the tenant workload, which incurs performance overhead.
In this work, we advocate in favor of *need-based protection* mechanisms, which are imperative to effectively mitigate CSCAs without sacrificing the benefits of resource sharing. Such a solution would also incur as little overhead as possible without significant performance or monetary cost. These objectives, however, become even more challenging to achieve in the absence of an accurate *need-assessment* mechanism. Therefore, the challenge is to provide real-time early-stage detection of CSCAs on the one hand, while providing mitigation as an OS-based service on the other hand. Mitigation in this case would be responsible for isolating cryptosystems from unwanted sharing and removing threats at runtime, which will be fast and incur minimum overhead. Moreover, a detailed description on the state of the art of detection and scheduling-based mitigation can also be found in [24].

3 The Kingsguard: Detection-Based Mitigation

The Kingsguard is an OS-level runtime detection-based mitigation mechanism, which is designed to detect, and subsequently mitigate, a large set of CSCAs. It enhances the capability of the OS, particularly Linux general-purpose distribution, with security features against side-channel information leakage. Figure 1 illustrates that the Kingsguard mechanism works in two distinct stages carried out by two distinct modules: the detection module and the mitigation module.

In the first stage, it uses multiple machine learning models that take, as input features, the real-time behavioral data of concurrent processes running on Intel's x86 shared memory architecture through hardware performance counters. These data are provided to the detection module, which is embedded inside the encryption library by the Kingsguard as shown in Figure 1. The detection module operates in the user space. We elaborate on this module and its functionality in Section 3.2 in more detail. Based on these data collected through HPCs, the Kingsguard detects if any malicious process is trying to manipulate the encryption process in order to extract information. If no malicious activity is reported, all processes run normally at their pre-assigned privilege levels. However, if malicious activity is detected, then the Kingsguard mechanism invokes the mitigation module in the kernel space and enters into the second phase. In this phase, through a netlink socket between user and kernel spaces, the Process IDs (PIDs) of all processes that were using the encryption library at the time of detection are provided to the mitigation module in the kernel space. The Kingsguard immediately suspends any ongoing encryption activity while identifying the IDs of processes that were using the encryption library. The mitigation module then evaluates these PIDs to separate trusted processes (usually system processes) from untrusted processes (usually user processes), if any. The module then initiates the procedure of removing untrusted process from the system and resumes the execution of all trusted processes (both system as well as user processes). The notion of *trust* is used in the sense that trusted processes are the system processes and untrusted processes are only those user processes that are trying to use an encryption service, and their execution of the cryptographic library is classified as an attack process. On the other hand, the processes running at root and other innocent user processes are considered as safe processes and non-vulnerable to attack.

In the following, we first elaborate on the threat model with which the Kingsguard mechanism deals and then provide the design details of the detection and mitigation modules.

3.1 Threat Model

We assume an advantageous scenario for the attacker to demonstrate that the Kingsguard remains effective even under weaker assumptions. Tromer *et al.* [33] have classified SCAs into synchronous and asynchronous attacks depending on whether or not the attacker can trigger the processing of known inputs (usually plain or ciphertexts). Synchronous attacks, where the attacker can trigger and observe encryption, are generally easier to perform from the attacker's perspective, and thus harder to defend against, since the attack does not need to determine the start and end of each encryption. We have assumed strong position for the attacker because attacker has the capability to exactly know when to start the encryption and when it will be stopped (attack process is synchronized with the victim process and it arbitrarily chooses plain texts), and therefore will consider the scenario of synchronous attacks where the attacker can request and observe encryption of arbitrarily chosen plain texts. Moreover, to minimize the effect of external noise for the attacker,

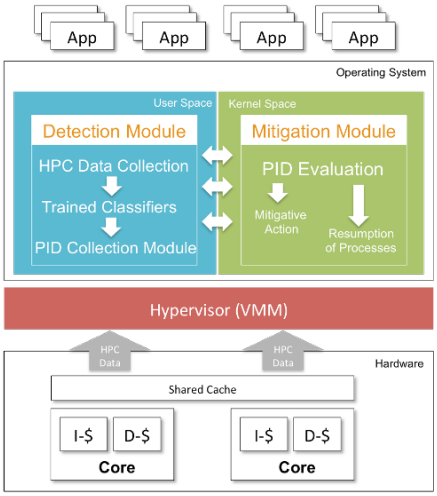

Fig. 1: The Kingsguard – the big picture.

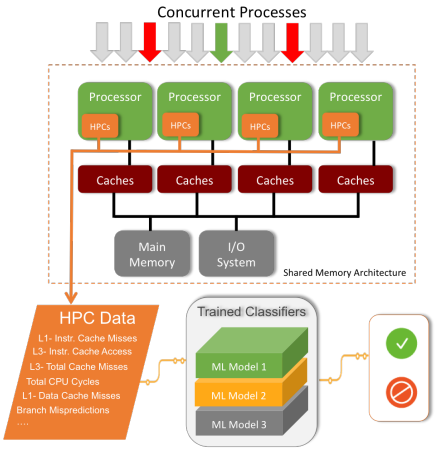

Fig. 2: Detection Module of the Kingsguard Mechanism.

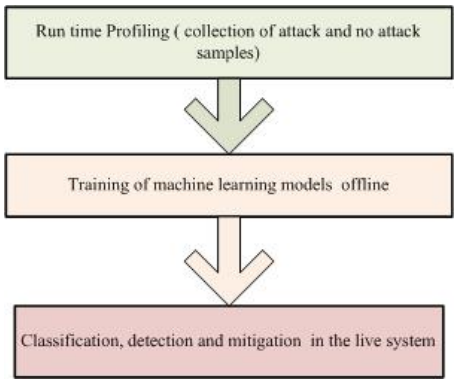

Fig. 3: The Kingsguard – overall design and development flow.

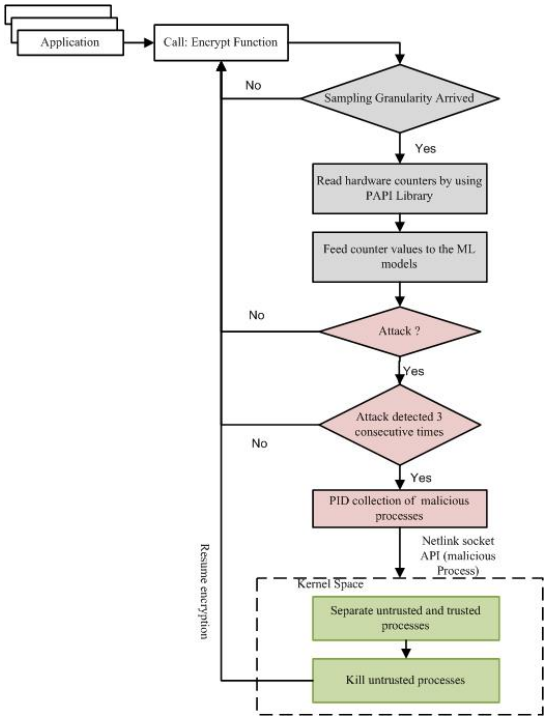

Fig. 4: Run time classification, detection and mitigation mechanism.

we assume that the attacker can be a co-resident on the same machine as the target encryption process. We also assume that the attacker can execute user-mode code on a processor core that is shared with the target encryption process but does not have access to the address space of the target process. We demonstrate that the Kingsguard works for same-core attacks as well as for cross-core attacks.

Table 1: List of selected Cache SCAs as use cases on Intel's core i7 machine.

#	**Cache SCAs**	**OpenSSL Version**	**Cryptosystem**
1	Flush+Reload	0.9.7l	RSA
2	Flush+Flush	0.9.7l/1.0.1f	AES
3	Prime+Probe	0.9.7l/1.0.1f	AES

We assume that attacks are persistent in nature, *i.e.*, the attacker process can repeat the same attack a reasonably large number of times. We also assume that any legitimate benign process can potentially be an attacker, thus the OS does not have prior knowledge or any specific privilege level associated with the attacker. Lastly, we assume that our threat model comprises multiple attacks, which can execute in any temporal order, thus the mitigation must protect the target encryption process under all possible execution scenarios. We have considered 3 state-of-the-art CSCAs targeting 2 different cryptosystems as use cases, *i.e.*, Flush+Reload on RSA and Flush+Flush and Prime+Probe on AES. Table 1 provides details on these use cases along with the OpenSSL versions being used and the time to recover the key by each of these attacks on an Intel's core i7-4770 CPU machine. These are well-established attacks and more details on them can be found in [15], [29], [40].

3.2 Runtime Detection Module

One of the distinguishing features of the Kingsguard mechanism is that it is designed to work at runtime, *i.e.*, when the attack is actually happening. The challenge of designing a runtime detection module for CSCAs is three-pronged. Firstly, a detection module would cause a slowdown in the encryption time and thus it could lead to significant performance overhead while trying to achieve higher detection accuracy. Secondly, accurate but late detection is useless for runtime detection. Theoretically, 50% completion of an attack is considered as sufficient for success [29]. Thus, detection speed is equally important for runtime adaptation. And thirdly, a detection mechanism must be highly accurate and should not lead to a higher number of False Positives (FPs) and False Negatives (FNs) at runtime. We considered all these aspects while designing the detection module for the Kingsguard.

The detection module of the Kingsguard has two major components: 1) Selection of appropriate hardware events that will reveal, at runtime, an insight into the cache behavior while under attack and 2) selection of appropriate machine learning models that could perform binary classification of *Attack* vs *No Attack* scenarios with high accuracy, high speed and minimum performance

overhead. One of the key features of the Kingsguard is that it operates under realistic system load conditions on commodity hardware. Therefore, we emulate the load conditions by running memory-intensive SPEC and CPU-intensive STREAM benchmarks simultaneously on the system, as independent background load. The load conditions are defined such that a *No Load (NL)* condition involves only a Victim and an Attacker process running, an *Average Load (AL)* involves Victim, Attacker and any two benchmarks running and a *Full Load (FL)* condition involves Victim, Attacker and any four benchmark processes running in the background. In order to maintain consistency in our results, we have used SPEC benchmarks in the training phase for the ML models, while using both SPEC and STREAM benchmarks during the testing phase. Doing so eliminates any bias of the ML models towards background system load conditions. It is important to mention that the state-of-the-art attacks [15], [29], [35], [2] have been demonstrated as running in isolated conditions, *i.e.*, attacker and victim being the only load on the system. Therefore, assuming realistic load conditions helps in validating the actual threat level these attacks pose on the one hand, while making it possible to assess the effectiveness of mitigation techniques on the other hand. Figure 2 illustrates an abstract view of the detection module and Figure 4 illustrates the complete flow chart of the Kingsguard.

3.2.1 Methodology

The methodology that the Kingsguard uses in its detection module is inspired by that proposed in [28]. It consists of three distinct phases, namely; 1) Runtime profiling, 2) Training of machine learning models and 3) Classification & detection as illustrated in Figure 3. In the following, we describe these phases in detail. During the runtime profiling phase, samples from selected hardware events are collected using HPCs under variable load conditions (NL, AL & FL). Once collected, these runtime profiles of the victim process are used for training and cross validation of the machine learning models in the next phase. We have collected labelled training data for roughly 1 million samples from all possible execution scenarios for victim process. Our training data is unbiased, i.e., it contains an equal number of samples for both attack and no-attack scenarios. Our models have not been overtrained becuase data is balanced as we collected equal number of attacks and no attacks samples. Further k-fold validation of the data samples is performed. Moreover, even if model is overtrained then at the test time we get data from hardware in real time, and offline data does not have any effect on the online data. Training is a one-time process in our proposed technique. In the last phase, trained classifiers use runtime data coming from hardware events for binary classification, i.e., *Attack* or *No Attack*.

3.2.2 Selection of Hardware Events

There are many hardware events that provide valuable information regarding normal vs abnormal behavior of running processes. We have performed extensive experimentation with all 12 hardware events with different scopes (L1-L3

cache level and system-wide events) as presented in Table 2. The selection of events is strictly based on the following factors: 1) the relevance of events with the attack behavior, 2) the high precision and distinctiveness of information on normal/abnormal behavior, 3) the selection of diversity and minimum correlation, 4) the minimum number of events to avoid performance overhead and multiplexing as counters can be non-deterministic if multiplexed.

Since we target access driven CSCAs, we only consider hardware events that are most affected by these attacks. We performed experimentation on a larger set of hardware events and selected the 12 most significant events as shown in Table 2.

Thus, we selected L1-Data Cache Misses (L1-DCM), L3-Total Cache Accesses (L3-TCA), L3-Total Cache Misses (L3-TCM) and Total CPU Cycles (TOT-CYC) as the most suitable minimum number of hardware events that are used by the Kingsguard detection module for the selected use case attacks.

Table 2: Selected events related to CSCAs

Scope of Event	**Hardware Event as Feature**	**Feature ID**
L1 Caches	Data Cache Misses	L1-DCM
	Instruction Cache Misses	L1-ICM
	Total Cache Misses	L1-TCM
L2 Caches	Instruction Cache Accesses	L2-ICA
	Instruction Cache Misses	L2-ICM
	Total Cache Accesses	L2-TCA
	Total Cache Misses	L2-TCM
L3-Caches	Instruction Cache Accesses	L3-ICA
	Total Cache Accesses	L3-TCA
	Total Cache Misses	L3-TCM
System-wide	Total CPU Cycles	TOT_CYC
	Branch Mispredictions	BR_MSP

Table 3: Detection results using LDA, LR and SVM models for Flush+Reload attack on RSA Cryptosystem

Model	**Load**	**Accuracy(%)**	**Speed(%)**	**FP(%)**	**FN(%)**	**Overhead(%)**
LDA	NL	99.51	0.98	0.488	0.001	0.94
	AL	99.50	0.98	0.492	0.008	
	FL	99.44	0.98	0.491	0.068	
LR	NL	99.51	0.98	0.489	0	1.63
	AL	99.50	0.98	0.494	0.006	
	FL	99.47	0.98	0.489	0.040	
SVM	NL	98.82	0.98	0.397	0.782	1.29
	AL	90.01	0.98	0.169	9.82	
	FL	95.79	0.98	3.211	0.998	

3.2.3 Selection of Machine Learning Models

In this section, we discuss the rationale for selecting machine learning models for the Kingsguard. The difference between attack and no attack scenarios is quite significant under NL conditions, which could be easily separated using a threshold. However, under a more realistic load condition (average and full), this situation worsens. Due to increased interference with caches, it becomes

hard to separate an attack scenario from a no attack scenario with simple threshold-based approaches. Adding to the problem, in practice, a system can be exposed to multiple CSCAs simultaneously or in any temporal order, which would further increase the difficulty in distinguishing an attack scenario using data from hardware events.

For such a system, machine learning models can be helpful in appropriately learning the behavior of each CSCA using HPC data, which has also been reported in past research as we explained in Section 2.1. Any selected ML model is supposed to deal with such a data mix in order to separate an attack from a no attack scenario. Our experiments show that not all ML models yield acceptable results and it is important to understand the diversity of data affecting the caches in attack/no attack scenarios for the selection of ML models. Since the Kingsguard uses runtime CSCA detection, we have applied stringent criteria for the selection of ML Models that best suit our design constraints, i.e., classification accuracy, implementation feasibility for runtime detection, minimum performance overhead, distribution of errors (false positives and false negatives) and detection speed. Detection accuracy is the single most important parameter for the module. Therefore, we have tested the detection accuracy for at least 12 ML models with a training data set collected for all 3 attacks running on the system. We then compare these models with the rest of the design constraints. For instance, Figures 5 and 6 illustrate the detection accuracy of all 12 models that are tested against F+R and F+F attacks, respectively, with variable load conditions, i.e., NL, AL and FL conditions. Many models perform well on detection accuracy but we argument in favor of two design challenges; 1) models should be easy to embed inside our detection module 2) models should have less implementation complexity (thus lesser computational complexity) so that the performance overhead is not excessive.

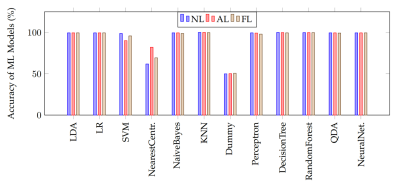

Fig. 5: Accuracy of ML Models for F+R (RSA)

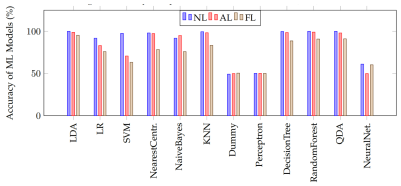

Fig. 6: Accuracy of ML Models for F+F (AES)

As shown in Figure 5, most of the ML models show high accuracy for F+R attacks, except Nearest-Centroid and Dummy Classifiers. For the F+F attack in Figure 6, most ML models exhibit low accuracy. Moreover, under load conditions, the classification accuracy of all models degrades against an F+F attack. This degradation is attributed to the stealthy nature of F+F attacks. Unlike F+R, for F+F attacks the Nearest Centroid classifier manifests good classification accuracy. Based on detection accuracy, the subset of ML

models that can be used at runtime consists of: LDA, LR, SVM, Naive-Bayes, KNN, Decision Tree, Random Forest and QDA. Although the most important is detection accuracy, it is not the only parameter to consider while deploying a high-speed runtime CSCA detection module. Another most important parameter to examine while comparing ML models is their implementation feasibility. Also, ML models should be able to quickly provide their decision while keeping their performance overhead minimum. Under these criteria, our experiments reveal that LDA, LR and SVM are the best performing ML models for the Kingsguard's detection module due to their light-weight implementation, high detection accuracy and minimum performance overhead. Section 4 provides detailed results.

3.3 Runtime Mitigation Module

As illustrated in Figures 1 and 4, once the trained classifiers report an attack, the mitigation module does not suspend encryption immediately. It suspends encryption only if same process exhibits malicious behavior on three consecutive detection samples. This number is chosen arbitrarily to remove the false positives. Once it is established that the current process is truly a malicious process, then the mitigation module suspends the encryption and obtains the PID of the malicious process. After acquiring the malicious PID, it is passed to the kernel part of the mitigation module. The kernel part of the mitigation module evaluates whether this PID is trusted or not. If it is trusted then encryption is resumed again, otherwise the mitigation module kills that process immediately. The Kingsguard considers synchronous attacks, in which an attacker process triggers the encryption by using the encryption library. Thus, the attacker itself is considered as a direct user of the encryption service. Therefore, we assume that an attacker, like any legitimate benign process in the system, would access the encryption library before attacking it in a synchronized fashion. Implementations of CSCAs such as Flush+Flush [15], [14], Flush+Reload [36], [8] and Prime+Probe [29], [14] are synchronous attack implementations. Therefore, in our experiments, we have also used synchronous attack implementations. Moreover, we do not consider the case where an attacker process, being the parent process, spawns a child process that executes the actual attack. We are providing a detection-based mitigation solution for general-purpose Linux distributions, which are inherently non-deterministic and not real-time with the assumption that Linux offers fair scheduling. As long as the Linux general purpose distribution is fair, our solution will target a larger set of vulnerable commodity hardware and protect it. We have proposed a solution to improve the common-case security.

The Linux OS provides isolation to kernel space from user space processes. Thus, in order to transfer critical information, such as PIDs, we use a Netlink socket as shown in Figure 1. The Netlink socket is a special Inter-Process Communication (IPC) primitive that is used for transferring information between kernel and user space processes. It provides a full-duplex communication link between the two through standard socket APIs for user-space processes, and a special kernel API for kernel modules [17]. In our mitigation module, we

use the Netlink socket instead of `system calls`, `ioctls` or `the proc file system` for communication between user and kernel spaces. System call and ioctl are complex IPCs in the sense that a session for these IPCs can only be initiated by the user-space applications. There is no way of passing any urgent message by the kernel module for a user-space application by directly using these IPCs. Applications periodically need to poll the kernel to obtain the state changes, although intensive polling is expensive. The Netlink socket solves this problem gracefully by allowing the kernel to also initiate sessions. Moreover, it is a non-trivial task to add system calls, ioctls or proc files for new features; we risk polluting the kernel and damaging the stability of the system. The Netlink socket is simple. Only the protocol type, which is a constant, needs to be specified and then the kernel module and applications can communicate using socket-style APIs immediately. Netlink is asynchronous in nature. It provides a socket queue to smooth the burst of messages. Unlike Netlink, system calls require synchronous message passing, which could affect the kernel's scheduling granularity if the time to process that message is long.

As illustrated in Figure 1, the mitigation module first evaluates whether the received PID is from a trusted process or not. It does so because, at runtime, it is highly likely that the set of active processes that are concurrently using the encryption library also contain some Linux system processes, which are considered as trusted by default. It is therefore imperative for the mitigation module to evaluate the malicious PID to separate trusted processes from untrusted ones (i.e., the user process). Once it is established that the process under consideration is untrusted then it is killed immediately by the mitigation module. The Kingsguard crypto library is used as a shared library (*.so file) by whatever process that wants to encrypt on the system. The Linux Runtime loader maps a copy of the Kingsguard mechanism in the address space of each process. So, when a process exhibits malicious behaviour, the related instance of the Kingsguard reports the detection and acquires the PID of the current process with the getpid() function. In Linux, when a process is using any shared library, the Linux runtime loader maps the functions/part of the shared library code into the virtual address space of the concerned process, which allows monitoring events to point to the exact process that is executing the shared library at the time of detection. Thus, the mitigation module only kills the malicious process and does not incur any performance overhead other than its own execution time overhead, which is reported in the experiments (Section 4).

3.4 Functional Description

Algorithm 1 provides a pseudo-code representation of the working principle of the Kingsguard mechanism. As illustrated, the detection module takes as input the sampling granularity for hardware events (`SamplingGranularity`), which can be either user-defined or automatically adjusted at runtime. By default, the sampling granularity is user-defined (offline) and set to fine-grain sampling. Another input is the total number of iterations for which we tested

Algorithm 1: Pseudo code representation of the working principle for the Kingsguard.

```
 1 SamplingGranularity, MaxIterations
 2 events← ∅, report← False, Victim ← NIL
 3 Victim← Get_Encryption_Lib()
 4 Set_of_Active_Processes ← Get_PIDs(Victim)
 5 Embed_Detection(Victim)
 6 Set_Hardware_Events(Victim)
 7 for i ← 1 to MaxIterations do
 8     if i mod SamplingGranularity == 0 then
 9         Activate_Detection()
10         events ← Read_Hardware_Events()
11         report ← ML_Classifiers(events)
12         Sleep_Detection()
13         if report == True then
               /* Attack is detected                                  */
               /* attack detected for 3 consecutive samples           */
               /* Activate Mitigation                                 */
14             if attack reported 3 consecutive times then
15                 Suspend(Encryption)
16                 Get_PID(Malicious Process)
17                 Untrusted_Processes ← Get_Untrusted_PIDs(Victim)
18                 Trusted_Processes ← Get_Trusted_PIDs(Victim)
19                 Kill(Untrusted_Processes)
20                 Resume_Encryption(Trusted_Process)
                   /* Turnoff Mitigation                              */
21                 return 1
    /* No attack detected!                                           */
22 return 0
```

the module (`MaxIterations`). The number of iterations varies for each attack as discussed in Section 4 (F+R is 1 encryption, F+F is 350-400 and P+P is 4800 encryption attacks). Lines $1-6$ show that a victim process (encryption process) is initialized, the detection module is embedded inside the encryption library and the hardware events are set around the victim process, considering it as the Region of Interest (ROI). For the selected number of iterations, the module activates detection after a number of encryptions equal to `SamplingGranularity` (lines $7-9$). Once activated, the detection module collects the data from hardware events (line 10) and feeds them as *features* to the selected binary classifier (line 11). Based on the classification, the module generates a report on the results. Detection is then deactivated (line 12) and if the report is `True` then an attack is reported (line 13). Otherwise, the victim process continues to execute uninterrupted. In the case of an attack, the Kingsguard immediately suspends all encryption activities in the system (line 14) and starts analyzing all processes currently using the encryption library (lines $15-17$). It separates the untrusted processes from the trusted ones and kills them (line 18) before resuming the encryption services.

Since algorithmic complexity is a measure of how long an algorithm would take to complete given an input of size n, therefore, the algorithmic complexity for Kingsguard's Algorithm-1 is dependent on its input "MaxIterations". The input MaxIterations is finite integer value and the maximum value for MaxIterations can be 4800 in case the Kingsguard is running for Prime+Probe attack. Thus, the called routines would always terminate. Algorthim-1 has a linear time complexity of the order O(n), where n is the MaxIterations as input.

4 Experiments and Results

4.1 Evaluation Setup

We have performed experiments on Linux Ubuntu LTS 16.04 Kernel version: 4.10.0-28-generic running on Intel's core $i7-4770$ CPU at 3.40-GHz with 64KB L1 (32KB L1d + 32KB L1i), 256KB L2, 8192KB L3 and 8GB system memory. We have used the Performance API (PAPI) [1] library to access HPCs on the Intel Core $i7$ machine. For RSA, the axtls Embedded ssl 2.1.4 library is used with bigint options set to the squared algorithm. For AES, we have used the openssl−0.9.7l library. We have used Netlink sockets for communication between the kernel and user space in Linux. Since the Kingsguard has two main modules, Detection and Mitigation, it is pertinent to mention here that the results reported in Tables 3–7 refer to the results of the detection module. The number of FPs generated by each model reports only the FPs of the detection module and not of the entire mitigation framework.

4.2 Case Study 1: Flush+Reload Attack on RSA

In our first case study, we demonstrate the detection and subsequent mitigation of F+R attacks on RSA cryptosystems.

Detection Accuracy We use percentage accuracy to show the validity of trained machine learning models as we have used an unbiased number of *no-attack* and *attack* samples in the training data. All three machine learning models show very high and consistent accuracy under all load conditions in Table 3. Even under FL conditions, the accuracy of LDA and LR remains above 99% while SVM shows above 95% accuracy. Most of the existing state-of-the-art detection mechanisms detect CSCAs in NL conditions. Our results demonstrate that the Kingsguard detection mechanism achieves very high accuracy for Flush+Reload attacks under realistic load conditions. The primary reason behind this high accuracy of machine learning models can be explained with the help of Figure 7, which illustrates the variation in magnitude of hardware events used for detection under attack and no-attack scenarios for FL conditions. Measurements show that all the features show clearly distinctive behavior under FL conditions, leading to the good performance of machine learning models.

Detection Speed Detection speed is a trade-off between how quickly an attack can be detected and how much overhead detection would cost. Flush+Reload is a single encryption attack [36]. For Flush+Reload, we consider detection

speed as a percentage of bits that are encrypted before the attack is successfully detected. Various attacks [29] have demonstrated that theoretically it is sufficient to retrieve 50% of secret key bits for a successful attack and the other 50% of secret key bits can be reverse engineered. Thus, a safe upper bound on the detection speed would be the detection before encryption of 50% of the secret key bits, i.e., before the encryption of 512 bits out of 1024 bits in this case. As shown in Table 3, all machine learning models are able to detect the attack well before this safe limit. In all cases, the detection module is able to detect Flush+Reload attacks in the first 20 bits (0.98%) out of 1024 bits of the RSA execution.

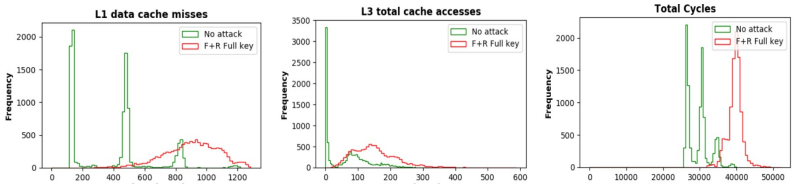

Fig. 7: Experimental results on selected events under FL conditions for RSA: With & Without Flush+Reload Attack.

Table 4: Detection time taken by various ML models under different load conditions for Flush+Flush atack on AES.

Load Type	**LDA (μs)**	**LR (μs)**	**SVM (μs)**
No Load	Min: 26	Min: 52	Min: 52
	Avg: 29	Avg: 55	Avg: 54
	Max: 31	Max: 99	Max: 101
Av. Load	Min: 27	Min: 54	Min: 54
	Avg: 29	Avg: 57	Avg: 58
	Max: 38	Max: 108	Max: 123
Full Load	Min: 30	Min: 57	Min: 58
	Avg: 42	Avg: 94	Avg: 95
	Max: 61	Max: 150	Max: 155

Table 5: Encryption time taken by RSA and AES while under various attacks and variable load conditions.

Load Condition	**RSA under F+R Attack (μs)**	**AES under F+F Attack (μs)**	**AES under P+P Attack (μs)**
No Load	Min: 7264	Min: 209	Min: 728
	Avg: 7604	Avg: 1395	Avg: 763
	Max: 26391	Max: 1680	Max: 924
Av. Load	Min: 7328	Min: 210	Min: 744
	Avg: 9982	Avg: 1477	Avg: 792
	Max: 22600	Max: 2004	Max: 1012
Full Load	Min: 7578	Min: 210	Min: 779
	Avg: 15284	Avg: 2899	Avg: 839
	Max: 28283	Max: 3121	Max: 1061

Confusion Matrix Detection inaccuracies can be further divided into false positives (cases when a no-attack condition is detected as an attack) and false negatives (cases when an attack condition is detected as a no-attack) to analyze detection results in detail. Table 3 shows FPs and FNs by all machine learning models while detecting Flush+Reload on RSA. Table 3 shows that with LDA and LR, the majority of the misclassifications belong to FPs. In the case of SVM, the behavior is different, as SVM exhibits more FN compared to FP under NL and FL conditions.

Performance Overhead Detection granularity and the implementation (code footprint) of machine learning models contribute to performance overhead. The detection granularity defines how efficiently the detection mechanism profiles hardware events and makes detection decisions. To this end, the detection module used in the Kingsguard mechanism incurs a $1-2\%$ performance degradation to the victim process in terms of makespan. These results are achieved with the highest sampling frequency, i.e., with a sample after every 10 bits being encrypted in the case of F+R on RSA and every 10 encryptions being performed in the case of F+F and P+P on AES. We embed ML models, after training, inside the encryption libraries, which greatly reduces the footprint of these models. Our experimental results, as illustrated in Table 4, show that the selected models take a fractional amount of time in performing their binary classification compared to the total encryption time taken by both RSA and AES cryptosystems under various load conditions as illustrated in Table 5. For instance, both LR and SVM models take roughly 55μs on average to classify an attack scenario under a no load condition whereas, under the same load conditions, RSA takes 7604μs while under F+R attacks, and AES takes 1395μs and 763μs while under F+F and P+P attacks, respectively. As the load conditions vary, there is no significant change in the measured results. Thus, the implementation of ML models does not significantly contribute to performance overhead.

4.3 Case Study 2: Flush+Flush Attack on AES

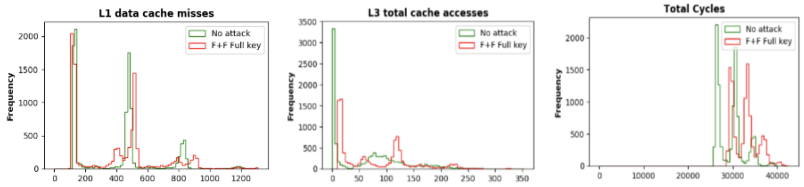

Fig. 8: Selected hardware events under FL conditions for AES encryption: With & Without Flush+Flush Attacks.

According to [15], it is virtually impossible to detect the attacker thread responsible for Flush+Flush attacks due to the absence of any abnormality in cache misses and hits generated by the attacker. However, in this work, we

illustrate that this attack is detectable from the victim's perspective as this results in more cache misses and accesses because of high speed flushing from the attacker.

Detection Accuracy Table 6 shows the detection accuracy of all machine learning models for Flush+Flush attacks. LDA shows very high accuracy under all load conditions for detection of Flush+Flush attacks on AES. The high inaccuracy of LR and SVM models under FL conditions can be explained with the help of Figure 8, which shows the behavior of used hardware events under attack and no-attack for FL conditions. It is evident that all the features start to overlap under attack and no-attack scenarios as shown in Figure 8. This behavior of overlapping features makes it harder for machine learning models to properly discern attack scenarios from no-attack scenarios. However, it is interesting to see that the LDA model is still able to show high accuracy in the case of FL conditions (95.20%).

Table 6: Detection results using LDA, LR and SVM models for Flush+Flush attacks on the AES Cryptosystem

Model	**Load**	**Accuracy(%)**	**Speed(%)**	**FP(%)**	**FN(%)**	**Overhead(%)**
LDA	NL	99.970	25.000	0.075	0.025	1.180
	AL	98.740	25.000	1.200	0.140	
	FL	95.200	12.500	4.600	0.230	
LR	NL	91.730	12.500	0.000	9.300	1.103
	AL	83.100	25.000	10.900	2.000	
	FL	75.860	25.000	98.390	1.610	
SVM	NL	97.420	12.500	0.000	100	0.790
	AL	70.640	12.500	94.560	5.440	
	FL	63.160	12.500	98.140	1.860	

Table 7: Detection results for Prime+Probe attacks on AES

Model	**Load**	**Accuracy(%)**	**Speed(%)**	**FP(%)**	**FN(%)**	**Overhead(%)**
LDA	NL	95.150	2.100	0.000	4.850	3.480
	AL	97.470	2.100	0.000	2.530	
	FL	100.000	1.100	0.000	0.000	
LR	NL	99.890	2.100	0.110	0.000	3.230
	AL	99.970	2.100	0.030	0.000	
	FL	99.920	2.100	0.080	0.000	
SVM	NL	100.000	2.100	0.000	0.000	5.080
	AL	100.000	2.100	0.000	0.000	
	FL	99.990	2.100	0.000	0.010	

Detection Speed On implementation of Flush+Flush on AES [15], at least 350-400 encryptions need to be performed to complete the attack. Thus, the detection of Flush+Flush attacks would only be useful if it is performed before the completion of 400 encryptions of AES. Therefore, for Flush+Flush attacks on AES, the detection speed is defined in terms of the number of encryptions needed to detect the attack taken as a percentage of 400 encryptions (upper

bound). As an example, a detection speed of 12.5% would mean that detection is achieved within the first 50 encryptions. The Kingsguard mechanism detects Flush+Flush attacks within the first 50 encryptions in most cases.

Confusion Matrix Table 6 shows the breakdown of misclassifications of all machine learning models into FPs and FNs while detecting Flush+Flush attacks on AES. For most of the cases, the majority of mispredictions falls into FPs. A few cases (SVM and LR under NL), where the majority of errors falls into the false negatives category, have very high accuracy and the actual number of false negatives and positives for them is very low.

Performance Overhead All three machine learning models incur a small profiling and detection overhead for the implementations of Flush+Flush attacks as shown in Table 6. Results show a maximum overhead of 1.18 in the case of LDA, which is considerably small.

4.4 Case Study 3: Prime+Probe Attacks on AES

Detection Accuracy Table 7 shows the detection accuracy of the selected ML models while detecting P+P attacks on AES. The detection accuracy is very high for all ML models (close to 100%) under all load conditions. The only exception is LDA under NL and AL, where it still shows a detection accuracy above 95%. In order to explain this high accuracy of all ML models, we can examine Figure 9, which shows the distribution of hardware events. As shown in this figure, all used features clearly show distinctive behavior under FL conditions (shown in Figure 9), the used hardware events start to overlap around two features (L3's total cache accesses and total cache misses) but still exhibit distinctive behavior leading to the good performance of ML models.

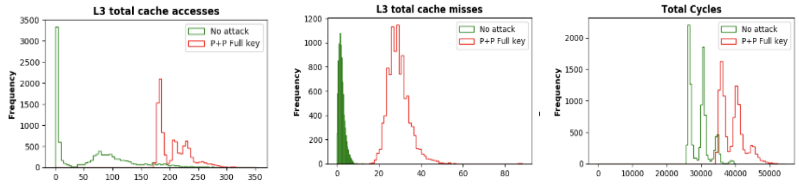

Fig. 9: Selected hardware events under FL condition for AES encryption: With & Without Prime+Probe Attacks

Detection Speed In order to reliably estimate the upper 4-bits of a secret key byte, a Prime+Probe attack needs at least 4800 AES encryptions [15]. Therefore, the detection of Prime+Probe would be useful only if it is achieved before completion of 4800 encryptions. Here, we define the detection speed as the number of encryptions needed to detect the attack, taken as a percentage of 4800 encryptions (i.e., the upper bound). For instance, a detection speed of 2.1% would mean that detection is achieved within the first 100 encryptions. Table 7 shows the runtime detection speed achieved by all ML models. Our detection module is able to detect the attack within the first 100 encryptions, which is well ahead of 4800 AES encryptions under all load conditions.

Confusion Matrix Table 7 shows the detection inaccuracy of our ML models under all load conditions for P+P attacks on AES. The percentage of false positives and negatives out of the evaluated samples is very close to 0 in almost all cases.

Performance Overhead Table 7 shows that the performance overhead for detecting P+P attacks is generally low for our detection module. We sample hardware events every 50 encryptions to make detection decisions. Since the detection speed is already very high, the sampling frequency of counters can be relaxed, which would lead to a further reduction in performance overhead.

Overall Performance Overhead of the Kingsguard The overall performance cost of the Kingsguard for performing detection and subsequent mitigation as compared to the key recovery time by potential attackers is a critical measure to evaluate the overhead. Table 8 illustrates the overall performance overhead incurred by the Kingsguard mechanism while performing different operations both in user- and kernel-space. For a sample set of 1000 iterations under variable load conditions, we have observed that the entire operation, from detection, collection of PIDs, evaluation of PIDs, killing untrusted Processes and resumption of service, takes 178μs, 199μs and 206μs on average for no load, average load and full load conditions, respectively. Compared to the time taken by the use-case attacks to recover the secret key in Table 1, one can notice that the entire mitigation mechanism takes only a fraction of time.

Table 8: Performance overhead at different stages for the Kingsguard mechanism while detecting Flush+Reload attacks on RSA

Load Type	**Detection(μs)**	**PID Collection(μs)**	**Mitigation(μs)**	**Total Overhead(μs)**
No Load	Min: 64	Min: 0.5	Min: 5	Min: 69.6
	Avg: 72	Avg: 1	Avg: 18	Avg: 91
	Max: 121	Max: 1.5	Max: 54	Max: 176
Av. Load	Min: 69	Min: 0.5	Min: 5	Min: 74.5
	Avg: 103	Avg: 1	Avg: 21	Avg: 125
	Max: 172	Max: 1.5	Max: 58	Max: 231
Full Load	Min: 70	Min: 1	Min: 6	Min: 77
	Avg: 138	Avg: 1.7	Avg: 25	Avg: 164
	Max: 208	Max: 2	Max: 108	Max: 318

Table 9: Mitigation accuracy of the Kingsguard under simultaneously occurring homogeneous attacks

Attack Type	**Attacking Processes**	**Detection Time (μs)**	**PID Collection Time (μs)**	**Mitigation Time (μs)**	**Mitigation Accuracy (%)**
F+R	2	103	1.000	18	99.010– 99.580
	3	104	0.990	19	99.660– 99.730
F+F	2	26	1.000	18	99.030– 99.950
	3	28	1.000	18	97.170–99.950
P+P	2	26	0.950	18	99.950– 99.990
	3	28	0.990	19	99.900–99.970

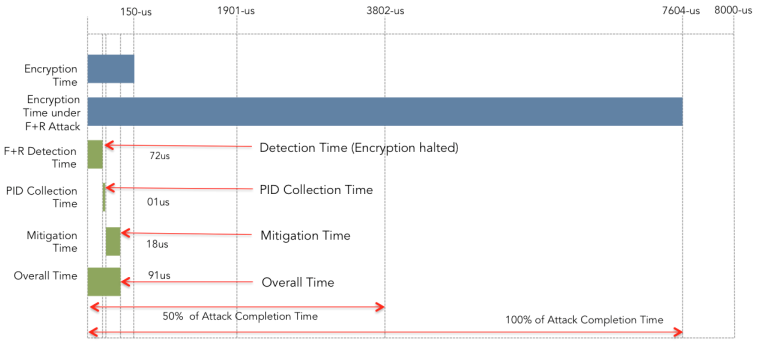

Fig. 10: Performance Overhead of the Kingsguard measured for Flush+Reload Attack running on RSA cryptosystem

Figure 10 provides clarity on the numbers presented in Table 8. For instance, victim executing encryption (RSA) takes 150μs on average under isolated conditions. Under no load conditions, Flush+Reload attacks on the RSA cryptosystem would require at least 7604μs to complete, whereas the Kingsguard mechanism can detect this attack in 72μs on average for No Load conditions (Table 8). Once an attack is detected, the encryption service is immediately halted by the OS, i.e., in the first 72μs in this case. In the next step, the PIDs of all processes using an encryption service are collected in the user space and this information is relayed to the kernel space for subsequent mitigation which takes 88μs on average. Based on this information, the mitigation module is activated, it differentiates between trusted and non-trusted processes, kills untrusted processes and resumes trusted execution, which takes 18μs on average in this case. Thus, the overall performance overhead of running the Kingsguard, from detection to mitigation, is measured at 178μs on average. As the mitigation mechanism is performing scheduling to kill untrusted processes, it can be nondeterministic under different load conditions. The mitigation mechanism should perform before 50% completion of the secret key so that the attacker is not able to build the key. For this security sensitive part, we made sure that once an attack is detected, the encryption service is halted and is only resumed when trusted execution resumes. This ensures that an attacker will not be able to execute an attack after 72μs. This time, in terms of secret key computation, refers to Table 3 which shows that the attack is detected at 9 bits out of 1024 secret key bits in this case. At 9 bits, the encryption is stopped and the threat is mitigated, which is far less than 50% of key computation. It is impossible for the detection mechanism to become stalled because it is working at fine granularity i.e. in the case of Flush+Reload (RSA), the detection framework samples the events after every 10 bits, whereas, the attacker needs 1024 bits to recover the secret key. Even if we consider that the detection was stalled for some reason, it will still detect the attack in the next 10 bits, which is well before the completion of the attack (1024 bits), because the detection works at very fine-granularity while providing early stage detection. In the case of Flush+Flush and Prime+Probe

attacks on AES, the mitigation overhead is relatively much less compared to the attack completion time as illustrated in Tables 1 and 8. These results are similar to the ones presented in Table 8.

5 Discussion

This section presents limitations, open problems and future research opportunities associated with the application domain of detection-based mitigation solutions. When we talk about CSCAs, either they are known or unknown, they leave their imprints on the caches. These imprints can be observed by carefully profiling the behavior of victim process without any initial knowledge of the type of attack. To help, hardware events have played an important role in the detection of such behaviors but it is important to note that hardware events can prove to be imprecise, non-deterministic and limited in number which can lead to increased inaccuracies in the results (FPs & FNs). A detailed insight on the challenges, perils and pitfalls of using HPCs for security can be found in [11]. Vulnerability assessment of system components can be performed by using simulators such as gem5 [12] to capture the behavior of smarter attacks and to better understand the microarchitectural components for security reasons. Moreover, an important step can be the design of new hardware events for security that can help in the behavioral analysis of systems under attack for known/unknown attacks.

We have provided a proof of concept as the Kingsguard which is able to capture the behaviors of multiple CSCAs and mitigate them in any temporal order. In our case we have demonstrated results on Flush+Reload, Flush+Flush and Prime+Probe attacks. It will be interesting to deploy a large repository of CSCAs as well as multiple variants of Transient Execution Attacks i.e., Spectre and Meltdown which rely on cache behaviors. As pointed out in the previous paragraph, as soon as the repository allows the detection of multiple vulnerabilities at the same time, we may face issues of multiplexing for HPCs (where multiple hardware events are scheduled on a single register). It can lead to inaccuracy and imprecision of results. Automated vulnerability assessment is one potential direction which involves a deep understanding of microarchitectural components to automatically detect known and unknown vulnerabilities in the systems.

Throughout the discussion, we elaborated that threshold-based solutions are not efficient enough to distinguish abnormal behavior from normal behavior because CSCAs are stealthy and contain a high bandwidth. Attacks can happen in any temporal order and sophisticated methods are required to profile data coming from hardware and later on classify them as attack/no-attack. For this reason, we have experimented with 12 ML models which provide empirical evidence to strengthen the belief that ML can be helpful in classifying anomalous behaviors. Hence, ML in security can be considered as a sophisti-

cated and resilient direction toward modern computing systems. Linear models in our case have proven to be successful on known CSCAs. There is more exploration required in this domain to determine if deep learning models and reinforcement learning can be helpful to reduce inaccuracies in the results (reduction of FPs & FNs). Another potential direction is to deploy hardware detection components which rely on using the ML models in hardware. Such a detection mechanism can be robust, precise and accurate while showing even lower overhead with a very early stage detection. Furthermore, a future exploration of adversarial attacks to corrupt data coming from hardware events and misclassify ML models can be an interesting direction to follow.

The Kingsguard mitigation is demonstrated in Linux, which uses the Completely Fair Scheduler (CFS) to schedule different applications. The scheduler runs with a range of specific time periods i.e., 5-10 ms in which different applications are scheduled for different resources. Reducing the scheduler time also introduces significant performance degradation and some applications may also be starved of resources such as the CPU, i/o etc. Scheduler-based mitigations now have to respect this scheduling time (5-10 ms) in order to schedule. But this time is very high from the encryption point of view. The state of the art reports that typically 50% of the key bits are enough to reconstruct the remaining secret key. To avoid this specific timing of the scheduler, the Kingsguard detection and mitigation is performed on the rounds of encryption instead of the scheduler time. This is why the Kingsguard is able to detect and subsequently mitigate rather stealthy attacks at a very early stage. Nowadays, we are experiencing attacks which are not even cryptosystem dependant and we may experience attacks in the future which are even stealthier in nature. Thus, scheduler-based mitigation may not remain a better fit for modern attacks until we reduce the scheduler timings and deal with a performance degradation caused by security issue.

6 Conclusion and Future Work

This paper proposes a novel OS-level run-time detection-based mitigation mechanism against CSCAs, called the Kingsguard, which enhances the security and privacy capabilities in general-purpose operating systems. The Kingsguard mechanism uses multiple machine learning models for run-time detection and relies on the profiling of concurrent processes, which are collected directly through the hardware events using HPCs in near real-time. We demonstrate that the Kingsguard is capable of detecting and subsequently mitigating Prime+Probe, Flush+Reload and Flush+Flush attacks on AES and RSA cryptosystems while running under Linux general-purpose distribution. We also demonstrate that the proposed mechanism is resilient to noise generated by the system under various loads. Our results show that the Kingsguard can mitigate known CSCAs with an accuracy of $> 95\%$ in most cases. To the best of our knowledge, this is the first research work that provides a run-time detection-based mitigation against Cache SCAs for Linux general-purpose dis-

tributions. Our proof of concept is also scalable in terms of vulnerabilities and deployability. This framework can also be used to detect other microarchitectural attacks such as Transient Execution Attacks i.e., Spectre and Meltdown because they rely on the same principle of CSCAs. The Kingsguard can be readily deployable on a number of exsisting Linux distributions such as Debian, Kali etc.

References

1. Performance application programming interface. In: *http : //icl.cs.utk.edu/papi/* (2018)
2. Aciiçmez, O.: Yet Another MicroArchitectural Attack: Exploiting I-Cache. In: Proceedings of the 2007 ACM Workshop on Computer Security Architecture, CSAW '07, pp. 11–18. ACM, New York, NY, USA (2007). DOI 10.1145/1314466.1314469
3. Akram, A., Mushtaq, M., Bhatti, M.K., Lapotre, V., Gogniat, G.: Meet the sherlock holmes' of side channel leakage: A survey of cache sca detection techniques. IEEE Access **8**, 70,836–70,860 (2020)
4. Alam, M., Bhattacharya, S., Mukhopadhyay, D., Bhattacharya, S.: Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel-attacks. Cryptology ePrint Archive, Report 2017/564 (2017)
5. Allaf, Z., Adda, M., Gegov, A.: A comparison study on flush+reload and prime+probe attacks on aes using machine learning approachess. UK Workshop on Computational Intelligence pp. 203—-213 (2017)
6. Askarov, A., Zhang, D., Myers, A.C.: Predictive black-box mitigation of timing channels. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS '10, pp. 297–307. ACM, New York, NY, USA (2010). DOI 10.1145/1866307.1866341. URL `http://doi.acm.org/10.1145/1866307.1866341`
7. Bazm, M.M., Sautereau, T., Lacoste, M., Sudholt, M., Menaud, J.M.: Cache-based side-channel attacks detection through intel cache monitoring technology and hardware performance counters. In: Fog and Mobile Edge Computing (FMEC), 2018 Third International Conference on, pp. 7–12. IEEE (2018)
8. Berard, D.: https://github.com/polymorf/misc-cache-attacks/
9. Briongos, S., Irazoqui, G., Malagón, P., Eisenbarth, T.: Cacheshield: Detecting cache attacks through self-observation. In: Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, pp. 224–235. ACM (2018)
10. Chiappetta, M., Savas, E., Yilmaz, C.: Real time detection of cache-based side-channel attacks using hardware performance counters. Appl. Soft Comput. **49**(C), 1162–1174 (2016). DOI 10.1016/j.asoc.2016.09.014
11. Das, S., Werner, J., Antonakakis, M., Polychronakis, M., Monrose, F.: Sok: The challenges, pitfalls, and perils of using hardware performance counters for security. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 20–38. IEEE (2019)
12. France, L., Mushtaq, M., Bruguier, F., Novo, D., Benoit, P.: Vulnerability assessment of the rowhammer attack using machine learning and the gem5 simulator-work in progress. In: Proceedings of the 2021 ACM Workshop on Secure and Trustworthy Cyber-Physical Systems, pp. 104–109 (2021)
13. Ge, Q., Yarom, Y., Cock, D., Heiser, G.: A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. Journal of Cryptographic Engineering pp. 1–27 (2016). DOI $10.1007/s13389-016-0141-6$
14. Gruss, D.: https://github.com/iaik/flush_flush
15. Gruss, D., Maurice, C., Wagner, K., Mangard, S.: Flush+Flush: A Fast and Stealthy Cache Attack. In: Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721, DIMVA 2016, pp. 279–299. Springer-Verlag New York, Inc., NY, USA (2016)
16. Gülmezoğlu, B., İnci, M.S., Irazoqui, G., Eisenbarth, T., Sunar, B.: A faster and more realistic flush+reload attack on aes. In: Revised Selected Papers of the 6th International Workshop on Constructive Side-Channel Analysis and Secure Design - Volume 9064,

COSADE 2015, pp. 111–126. Springer-Verlag New York, Inc., New York, NY, USA (2015). DOI 10.1007/978-3-319-21476-4_8. URL http://dx.doi.org/10.1007/978-3-319-21476-4_8
17. He, K.K.: Kernel korner. why and how to use netlink socket. *https : //www.linuxjournal.com/article/*7356 (2005)
18. Irazoqui, G., Inci, M.S., Eisenbarth, T., Sunar, B.: Wait a minute! A fast, Cross-VM attack on AES. In: International Workshop on Recent Advances in Intrusion Detection, pp. 299–319. Springer (2014)
19. Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. CoRR **abs/1801.01203** (2018)
20. Kulah, Y., Dincer, B., Yilmaz, C., Savas, E.: Spydetector: An approach for detecting side-channel attacks at runtime (2018)
21. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown. CoRR **abs/1801.01207** (2018)
22. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-level cache side-channel attacks are practical. In: Proceedings of the 2015 IEEE Symposium on Security and Privacy, SP '15, pp. 605–622. IEEE Computer Society, Washington, DC, USA (2015). DOI 10.1109/SP.2015.43. URL http://dx.doi.org/10.1109/SP.2015.43
23. Liu, L., Wang, A., Zang, W., Yu, M., Xiao, M., Chen, S.: Shuffler: Mitigate Cross-VM Side-Channel Attacks via Hypervisor Scheduling. In: International Conference on Security and Privacy in Communication Systems, pp. 491–511. Springer (2018)
24. Mushtaq, M.: Software-based Detection and Mitigation of Microarchitectural Attacks on Intel's x86 Architecture. Theses, Université de Bretagne Sud (2019). URL https://hal-univ-ubs.archives-ouvertes.fr/tel-03105715
25. Mushtaq, M., Akram, A., Bhatti, M., Rao, N.B.R., Lapotre, V., Gogniat, G.: Run-time detection of Prime+ Probe side-channel attack on AES encryption algorithm. In: Global Information Infrastructure and Networking Symposium (2018)
26. Mushtaq, M., Akram, A., Bhatti, M.K., Chaudhry, M., Lapotre, V., Gogniat, G.: Nightswatch: a cache-based side-channel intrusion detector using hardware performance counters. In: Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy, p. 1. ACM (2018)
27. Mushtaq, M., Akram, A., Bhatti, M.K., Chaudhry, M., Yousaf, M., Farooq, U., Lapotre, V., Gogniat, G.: Machine Learning For Security: The Case of Side-Channel Attack Detection at Run-time. In: 25th IEEE International Conference on Electronics Circuits and Systems, Bordeaux, FRANCE (2018)
28. Mushtaq, M., Bricq, J., Bhatti, M.K., Akram, A., Lapotre, V., Gogniat, G., Benoit, P.: Whisper: A tool for run-time detection of side-channel attacks. IEEE Access **8**, 83,871–83,900 (2020). DOI 10.1109/ACCESS.2020.2988370
29. Osvik, D.A., Shamir, A., Tromer, E.: Cache Attacks and Countermeasures: The Case of AES, pp. 1 − −20. Springer Berlin Heidelberg, Berlin, Heidelberg (2006). DOI $10.1007/11605805\_1$. URL $http://dx.doi.org/10.1007/11605805_1$
30. PENG, S.h., ZHOU, Q.f., ZHAO, J.l.: Detection of cache-based side channel attack based on performance counters. DEStech Trans. on Computer Science and Engg. (2017)
31. Sabbagh, M., Fei, Y., Wahl, T., Ding, A.A.: SCADET: a side-channel attack detection tool for tracking Prime+ Probe. In: ICCAD (2018)
32. Stefan, D., Buiras, P., Yang, E.Z., Levy, A., Terei, D., Russo, A., Mazières, D.: Eliminating cache-based timing attacks with instruction-based scheduling. In: European Symposium on Research in Computer Security, pp. 718–735. Springer (2013)
33. Tromer, E., Osvik, D.A., Shamir, A.: Efficient Cache Attacks on AES, and Countermeasures. Journal of Cryptology **23**(1), 37–71 (2010). DOI 10.1007/s00145-009-9049-y. URL http://dx.doi.org/10.1007/s00145-009-9049-y
34. Vattikonda, B.C., Das, S., Shacham, H.: Eliminating Fine Grained Timers in Xen. In: Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop, CCSW '11, pp. 41–46. ACM, New York, NY, USA (2011). DOI $10.1145/2046660.2046671$. URL $http://doi.acm.org/10.1145/2046660.2046671$
35. Yarom, Y., Benger, N.: Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack

36. Yarom, Y., Falkner, K.: FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In: Proceedings of the 23rd USENIX Conference on Security Symposium, SEC'14, pp. 719–732. USENIX Association, Berkeley, CA, USA (2014). URL http://dl.acm.org/citation.cfm?id=2671225.2671271
37. Yarom, Y., Genkin, D., Heninger, N.: CacheBleed: A Timing Attack on OpenSSL Constant Time RSA, pp. 346 − −367. Springer Berlin Heidelberg, Berlin, Heidelberg (2016). DOI $10.1007/978-3-662-53140-2_17$
38. Zhang, R., Su, X., Wang, J., Wang, C., Liu, W., Lau, R.W.H.: On mitigating the risk of cross-vm covert channels in a public cloud. IEEE Transactions on Parallel and Distributed Systems **26**(8), 2327–2339 (2015). DOI $10.1109/TPDS.2014.2346504$
39. Zhang, T., Zhang, Y., Lee, R.B.: Cloudradar: A real-time side-channel attack detection system in clouds. In: International Symposium on Research in Attacks, Intrusions, and Defenses, pp. 118–140. Springer (2016)
40. Zhang, Y., Juels, A., Reiter, M.K., Ristenpart, T.: Cross-vm side channels and their use to extract private keys. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS '12 (2012)
41. Zhang, Y., Reiter, M.K.: Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS '13, pp. 827–838 (2013)