No Leakage Without State Change: Repurposing Configurable CPU Exceptions to Prevent Microarchitectural Attacks

Daniel Weber*, Leonard Niemann†, Lukas Gerlach*, Jan Reineke‡, Michael Schwarz*
*CISPA Helmholtz Center for Information Security
†usd AG
‡Saarland University

***Abstract*—Microarchitectural side-channel attacks have become significant threats to computer system security. While writing side-channel-resistant code can mitigate these attacks, it is time-consuming and error-prone. Detection approaches provide an alternative by monitoring the system for signs of ongoing attacks. However, distinguishing between malicious and benign processes is complex, error-prone, and ineffective against sophisticated attacks.**

In this paper, we propose a novel approach, IRQGuard, which shifts the focus to proactive mitigation. IRQGuard enables the victim to monitor its own microarchitectural events resulting from microarchitectural state changes. Leveraging existing CPU features, IRQGuard uses interrupt requests (IRQs) triggered by victim-specific microarchitectural state changes within predefined code regions. This self-monitoring eliminates noise of unrelated applications, enabling immediate detection and response to potential attacks. Our proof-of-concept implementation demonstrates that IRQGuard stops information leakage in under 200 CPU cycles, outperforming current methods significantly. We evaluate IRQGuard on both cryptographic (OpenSSL) and non-cryptographic (toilet command-line utility) applications. We demonstrate IRQGuard's real-world viability by protecting an OpenSSH server from cache attacks. IRQGuard offers a practical, low-overhead solution for mitigating a wide range of microarchitectural attacks on Intel, AMD, and Arm CPUs.

I. Introduction

Recent years have shown that microarchitectural side-channel attacks are a danger to the security of computer systems. Mitigating microarchitectural side-channel attacks is challenging both at the software and the hardware level. While software can be written to resist side-channel attacks [46], [27], this is typically only done for cryptographic implementations [49], [13], [6] due to performance overhead and non-negligible complexity of such code. State-of-the-art techniques for defense in depth and protecting general applications from side-channel attacks rely on detection at runtime [66], [40], [41], [43], [55]. Existing proposals mainly rely on continuously sampling performance-monitoring counters (PMCs) that count microarchitectural events. However, these techniques suffer from significant limitations. First, the detection is *asynchronous*, requiring constant evaluation of PMC readings and still leaving a small but powerful window of opportunity for an attack. This window is often in the range of milliseconds—enough to leak an entire cryptographic key to an attacker before the attack is stopped. The constant evaluation often also requires non-negligible resources, such as an entire CPU core or dedicated machines [66], [50]. Second, the detection is typically applied in an *unfocused way*, including the entire system, i.e., various applications that do not have to be protected. As a trade-off, the detection either suffers from many false positives, as, e.g., memory-intensive parts of applications can easily be misclassified as cache attacks [16], or has to use a high threshold that still allows camouflaged attacks [31]. Third, most proposals do not discuss what the detection approach does on a *(false) positive*. Kosasih et al. [31] also identified these problems in concurrent work. They develop new camouflaged attacks that mask their malicious execution patterns behind benign program execution, allowing them to evade detection effectively. They conclude that it is uncertain whether real-time cache side-channel attack detection systems can be deemed effective for practical use in real-world scenarios. Thus, monitoring the entire system with performance counters appears to be a dead end.

In this paper, we emphasize that the *victim needs to monitor itself*, i.e., only its own microarchitectural events, instead of relying on a third party that can only monitor the microarchitectural events of the entire system. As a result, there is significantly less noise in the observed events, and an attacker can neither hide in the noise nor attack at such a low speed that it would drown in the noise. In other areas, it is already realized that it is most effective when the victim monitors itself to prevent leakage. For example, tamper-resistant security modules destroy their data when detecting tampering [48]. This is also true outside of computer science, where, e.g., ATMs destroy money on attacks using the intelligent banknote neutralization system that dyes banknotes during an attempted theft. In these cases, it is clear that the attack target has to monitor itself instead of a third party monitoring the environment, as the attack target can identify an attack quicker and thus react immediately.

To apply the same idea to microarchitectural attacks, we propose IRQGuard[1], a technique that leverages existing configurable CPU features for *proactively mitigating* microarchitectural side-channel attacks. We request the CPU to issue an

[1]IRQ stands for interrupt request, the type of CPU exception we use in our proof-of-concept implementation to immediately stop a victim

exception in the form of interrupt requests (IRQs) based on victim-specific configurable *microarchitectural state changes within a specified code region of the victim*. The main intuition is that microarchitectural attackers need to observe changes in the microarchitectural state caused by secret-dependent operations in the victim. Thus, if the victim stops before changing the state, there is no observable leakage for the attacker. With IRQGuard, the CPU issues an interrupt when a predefined number of microarchitectural state changes is reached. This diverts the control flow of the victim to a predefined handler, stopping the leakage almost instantly. Contrary to previous work, *victim-observable* state changes within a critical *code section* instead of *attacker-created* state changes *in the entire system* within a *time window* are used for attack mitigation. As the threshold for microarchitectural events is measured over a fixed code snippet of the victim, IRQGuard makes camouflaged attacks [31] difficult to impossible. Thus, IRQGuard is not a passive monitoring, but a proactive mitigation approach.

We demonstrate our PoC implementations on Intel, while showing that the same approach works for AMD and Arm CPUs, which makes our approach applicable to a wide range of systems. As the entire logic for triggering interrupts is handled in hardware, IRQGuard does not require any software-based sampling component that has to run all the time. IRQGuard simply provides transaction-style constructs to protect code against various side-channel attacks. These transactions are similar to Cloak [17], but configurable in terms of events and number of events. Thus, unlike Cloak, IRQGuard is neither limited to cache misses nor to Intel CPUs. IRQGuard can stop the leakage after fewer than 200 CPU cycles, thus outperforming all related approaches, besides Cloak, by orders of magnitude.

We demonstrate IRQGuard on OpenSSL, an application regularly used to demonstrate side-channel attacks [5], [42], [18]. To demonstrate the real-world applicability, we protect the OpenSSL library inside OpenSSH and demonstrate that OpenSSH still works as long as no attack is mounted against it. Furthermore, by hardening the `toilet` command-line utility, which is used to render text as ASCII art, we show that IRQGuard can protect non-cryptographic targets from leaking their access patterns. To demonstrate the mitigation of TLB-based attacks, we show that IRQGuard can mitigate TLBleed [15]. For all evaluated attacks, IRQGuard effectively mitigates exploitation by stopping the leakage entirely or at least significantly reducing the leakage rate.

IRQGuard is a practical mitigation for a series of microarchitectural side-channel attacks. Due to its generic hardware-based concept, IRQGuard can mitigate a wide range of microarchitectural attacks as long as the CPU features a method to raise an exception (e.g., an interrupt) when a configurable number of microarchitectural events is reached for the victim. Note that transient execution attacks are out of the scope of this work. The most related tool to our approach, as it also uses a hardware-based approach to protect applications from microarchitectural side-channel attacks, is Cloak [17]. However, Cloak is based on Intel TSX, a now deprecated instruction-set extension on Intel CPUs, whereas our approach works on Intel, AMD, and Arm CPUs. Whereas Cloak is limited to stopping a program after the *first* cache miss, IRQGuard allows the developer to choose to either behave the same or configure an additional error margin. The ability to rapidly abort a program under attack, together with the low runtime overhead, makes IRQGuard a practical solution that can be widely deployed on CPUs.

To summarize, we make the following contributions:

1) We introduce a novel approach to use interrupt requests sent by the CPU to actively *mitigate* ongoing microarchitectural attacks by observing microarchitectural events per victim-instruction sequence instead of globally over time.
2) We present IRQGuard, an open-source[2] proof-of-concept (PoC) implementation showing that our approach mitigates Flush+Reload, Prime+Probe, and TLBleed.
3) We demonstrate IRQGuard's real-world applicability by hardening the OpenSSL library and protecting an OpenSSH server.
4) We show IRQGuard's versatility by hardening the non-cryptographic `toilet` command-line utility.

II. Background

In this section, we introduce the necessary background on side-channel attacks, transient-execution attacks, and PMCs.

A. Microarchitectural Side-Channel Attacks

Microarchitectural side-channel attacks are a class of security vulnerabilities that exploit implementation details in the microarchitecture of modern computer systems. By mounting side-channel attacks, attackers can extract sensitive information, such as cryptographic keys [61], [21], [33], [37], [63], [5], [42], [18], [28], [36]. These attacks exploit that different CPU operations have different performance characteristics, e.g., varying execution times or power consumption.

Various microarchitectural components have been exploited in side-channel attacks [64], [44], [7], [2], [14], [45], [65]. Flush+Reload [64] and Prime+Probe [44] are two prominent examples of cache-based side-channel attacks (or cache attacks). Both attacks yield a primitive to observe whether a victim process has accessed memory location M. While for Flush+Reload, M is a shared memory address between the attacker and the victim process, this is not needed for Prime+Probe as long as the attacker and victim execute on the same CPU. In Flush+Reload, the attacker first flushes M from the CPU cache using a cache maintenance instruction, such as `clflush`. Next, the attacker waits for the victim to finish its sensitive computation. Lastly, the attacker times the access to M. As the access time decreases for cached memory, the attacker can reason about the cache state of M. In Prime+Probe, the attacker exploits that a memory address M can only be stored at one of n positions in the CPU cache. Accessing these n memory addresses and thereby caching all of them and evicting the memory address M is the first step of Prime+

[2]https://github.com/cispa/irqguard

Probe. Next, the attacker waits for the sensitive computation of the victim. Afterward, the attacker times the access to all n memory addresses. If all accesses are fast, the victim did not access a memory address colliding with M, as accessing one of these addresses evicts one of the n memory addresses used in the first step.

An attacker exploiting TLBleed [15] performs steps similar to a Prime+Probe attacker. The difference is that the attacker spies on accesses for an entire memory page P and targets the TLB instead of the CPU cache. First, the attacker evicts the virtual-to-physical address mapping from page P by accessing memory pages colliding with the TLB set of P. Then, the attacker accesses the memory pages and measures the access time.

An effective defense against side-channel attacks is constant-time programming [27], which ensures that control- and data flow are independent of secret values. While constant-time programming is used for state-of-the-art cryptographic implementations, it is tedious and error-prone to write programs in such a way.

B. Performance-Monitoring Counters

Performance-monitoring counters (PMCs) are hardware counters built into modern CPUs to count the number of specific microarchitectural events in the system. Such events include the number of executed instructions, cache misses, and mispredicted branches [22].

On Intel architectures, PMCs are implemented as registers and are split into two groups: *programmable* and *fixed* counters. While the fixed counters `FIXED_CTRx` count predefined events, a user can utilize programmable counters `PMCx` to count a specific event by modifying the corresponding control register `PERFEVTSELx`. Similar mechanism exist for AMD [1, Chapter 17] and Arm [3, Chapter G8.4]. As PMCs may overflow, it is possible to configure a control register to generate an interrupt on a counter overflow.

III. Threat Model

We assume that the attacker and victim execute code on the same machine. The attacker can execute native code with user-level privileges. The victim deploys a program executing computations protected by our approach. We refer to this program as the *protected program* for the remainder of the paper. The attacker can execute their attacks on a co-located CPU core, including the sibling thread of the victim program's logical core, i.e., the hyperthread.

We consider all side-channel attacks discussed in Section V to be in scope. We exclude attacks directly exploiting hardware vulnerabilities instead of victim behavior, such as Meltdown-type attacks and microarchitectural fault attacks (e.g., Rowhammer), as well as attacks that require hardware access by the attacker. As our approach can be applied to some of these attacks given further assumptions, we discuss under which assumptions IRQGuard can be used for further microarchitectural attacks in Section VII.

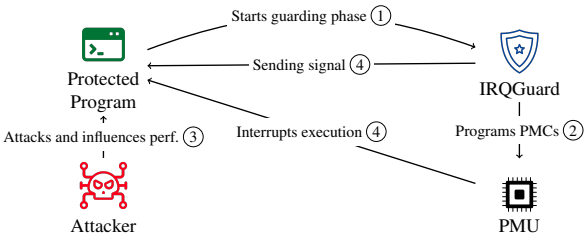

Fig. 1: After starting the guarding phase, the core executing the victim program is interrupted by a threshold violation of the PMCs, and IRQGuard signals the protected program.

IV. IRQGuard

In this section, we describe the concepts and the implementation details of IRQGuard. We introduce the main idea of letting the victim configure threshold for its expected microarchitectural state changes. For tight thresholds, we demonstrate that IRQGuard can profile the expected performance characteristics of the victim program. During said profiling phase, IRQGuard also keeps track of memory accesses done by the victim program. This allows IRQGuard to prefetch the expected memory accesses during later phases to minimize the noise further, thus allowing even tighter detection thresholds. Finally, this section discusses how the PMC interrupts stop the victim immediately and what the post-abort actions of the victim are.

A. Design Idea

The design of IRQGuard relies on the fact that side-channel attacks require the *victim* to interact with the microarchitecture, which can be observed by monitoring the hardware during the victim's execution. For example, side channels can be described as an interplay of 3 types of sequences: reset, trigger, and measurement sequence [59]. In this setting, the reset and measurement sequence are executed by the attacker and can thus be hidden from detection mechanisms by, e.g., artificially slowing down the attacker's computations and thus effectively hiding the computation in the noise of all running processes [31]. However, the trigger sequence must be executed by the victim process and thus cannot be hidden that easily if only the victim process is monitored. The observation is that for side-channel attacks, leakage only exists if the victim *changes* the microarchitectural state to a different state than the reset state induced by the attacker. This modification by the victim can be captured using PMCs on the victim side. For example, in a cache attack, the attacker sets a cache line to uncached and waits for the victim to change it to the cached state. Based on this interaction of a victim process with the microarchitecture, the idea is that we can detect the anomalies on the victim side. We know that an attacker may have altered the state of the microarchitecture to force the victim to change the microarchitectural state, resulting in information leakage.

As PMCs expose the internal state of the microarchitecture, they can record fine-grained information about, e.g., the number of cache misses, retired micro-ops, or pipeline flushes.

In our approach, the victim program is protected by profiling its behavior under normal circumstances, i.e., without an ongoing microarchitectural attack. Profiling means recording the influence of the program's confidential computations on PMCs programmed with different events discussed in Section IV-C. Based on this profile, for each PMC, we define a threshold, i.e., an event count that, if exceeded, indicates an ongoing attack with a high probability. Assume a PMC's maximal value is $PMC_{maxvalue}$. The supported bit-width for PMCs on the system defines this value. We define a threshold t and program each PMC to $PMC_{maxvalue} - t$. These thresholds are devised such that they should never be exceeded in a normal execution but only when a malicious actor is mounting a microarchitectural attack against the program. We configure the PMC to send an interrupt upon overflowing. Consequently, the interrupt immediately stops and notifies the protected program if a threshold violation (and thus likely an attack) occurs. The remainder of this section explains this approach in more detail based on our PoC implementation.

B. IRQGuard Overview

IRQGuard allows programs to alter their control flow if an attack is ongoing, effectively preventing further leakage. Our approach applies to cryptographic and non-cryptographic programs. Our PoC implementation consists of a kernel module, which implements the main logic of the tool, and a C library, which acts as an API. Alternatively, a userspace application can be built that interacts with the Linux `perf` interface. While we use such an implementation for Arm and AMD (cf. Section VII), we observe better results using our kernel module implementation. The developer annotates the critical part of their code, e.g., a decryption routine or a secret-dependent array lookup, by inserting calls to functions of the IRQGuard API. We refer to these code parts as the guarded section.

IRQGuard comprises two offline phases and an online phase. The first offline phase (the *profiling phase*) is to find thresholds for a set of performance-monitoring events during the execution of the guarded section (Section IV-D). The second offline phase is the recording of expected memory accesses of the guarded section to prefetch these memory accesses at runtime, thus reducing the number of unrelated events in PMCs (Section IV-E). The offline phases are only required once. The online phase (the *guarding phase*), leverages the PMCs to mitigate attacks on the guarded section of the protected program (Section IV-F). If the CPU triggers an interrupt, it is forwarded to the protected program, thus enabling the programmer to act accordingly, e.g., abort the execution (Section IV-G).

Figure 1 shows an overview of IRQGuard protecting a program. The program can be shipped to a system with (unprivileged) untrusted users. The previously generated thresholds, together with the annotations of the critical functions, allow IRQGuard to program the PMCs of the shared system. IRQGuard immediately halts the execution of the protected program when attacked during the guarded section, thus mitigating the ongoing attack.

C. Preparation: Choosing the Optimal Performance-Monitoring Events

IRQGuard relies on PMCs to handle anomalies. We have to choose events that correlate with the side channels that we want to defend against. Choosing the optimal set of events to protect against the largest possible set of side-channel attacks is challenging. Previous research already came up with approaches to detect specific attacks based on PMCs [67], [29], [43], [66], [10], [41], [40], [55], [56]. For our work, we use a combination of PMCs where previous work identified that they correlate well with side-channel attacks. Modern CPUs typically feature 4 or 8 programmable PMCs [24, Chapter 18]. Typically 3 fixed-function PMCs are accessible [24, Chapter 18].

The entire list of configured events we use for our PoC implementation is shown in Appendix A together with the attacks associated with each event. Our configuration combines PMCs for the L1 and last-level cache (LLC) with TLB misses and branch mispredictions. Note that L2 cache attacks also leave traces in the L1 cache, so we do not require specific PMCs for them. Furthermore, note that this configuration should be seen as a default configuration. Developers using IRQGuard can extend or adapt the configuration to their own needs. Side-channel attacks found in the future might need additional performance events for IRQGuard to defend against their exploitation.

D. Offline Phase: Profiling Phase

Based on the selected list of performance-monitoring events, IRQGuard generates thresholds for these events that are exceeded in the presence of microarchitectural attacks. For this, the developer adds the IRQGuard API calls to the program. All that is needed are calls to 4 IRQGuard API functions: At program start, IRQGuard must be initialized with a call to `iguard_init`. Afterward, the function `iguard_parse_pmc_config` is given a filepath to a configuration file containing the configurations of the PMCs. Note that in this step, the configuration does not contain any thresholds yet. The code to be protected (i.e., the guarded section), e.g., the decryption routine of a cryptographic algorithm, has to be annotated by calling the function `iguard_start_protection` before and the function `iguard_stop_protection` after that code, resembling the usage of Cloak [17]. While the efficacy of our approach decreases with longer and more complicated code inside the guarded section, there are no theoretical restrictions on the code.

Developers should wrap the code processing confidential information, e.g., cryptographic keys or secret-dependant control flow, inside the guarded section. For example, an RSA signing application wraps the decryption routine inside the guarded section to prevent leaking of the private key handled by the function.

For profiling, the developer signals IRQGuard to run in profiling mode and executes the program multiple times in common scenarios, allowing IRQGuard to generate a baseline profile of the PMC values during executions without an

ongoing attack. After profiling, the thresholds in the config are updated. A naïve but promising approach is to take the maximum observed value and add an error margin. We observe that for our PoC implementation, $20\,\%$ is a sufficient error margin for most cases (cf. Section V). While in the default case, the program is profiled on the same system that is used in the guarding phase, Section V also shows that thresholds can be ported to another similar system. Requiring a threshold to distinguish between benign and malicious activity is a common endeavor for our tool and related work [17], [50], [43]. While our PoC implementation only uses a fixed threshold, the approach also works with more sophisticated detection mechanisms, e.g., the change-point detection method proposed by Zhang et al. [66]. After a PMC overflows and transfers the execution from the protected program to IRQGuard, IRQGuard can decide whether it notifies or signals the protected program based on further metrics.

E. (Optional) Offline Phase: Prefetching Memory Accesses

For an optimal efficacy of IRQGuard, an accurate distinction from microarchitectural events introduced by an ongoing attack compared to the expected events is key. Hence, IRQGuard supports automated prefetching of the expected memory accesses of the victim program, as shown to be effective by Cloak [17]. This prefetching reduces the number of microarchitectural state changes in the absence of an attacker. Thus, for a normal operation, most memory accesses are cached by the prefetching, resulting in few to no cache misses in the guarding phase as long as no attacker tampers with the microarchitectural state. The only manual interaction is that the user needs to create a prefetching profile for the protected program. This is done by executing a single script, which requires no additional user interaction.

Our implementation uses Intel Pin [26] to record the memory access pattern of the protected program. In our case, we mark the location of the guarded section, i.e., between `iguard_(start/stop)_protection`, with a unique instruction such that we can identify the protected section from our custom Intel Pin plugin. We choose the instruction `verr` as it is a legacy x86 instruction without side effects that modern compilers are not expected to emit. Once the guarded section is identified, all memory accesses of it are logged. As memory accesses tend to change depending on the control flow of a function, we further heuristically fill gaps in the previously created memory address trace by adding dummy memory accesses whenever two addresses are less than 4 cache lines apart. The resulting list of memory accesses, encoded as relative offsets in the binary, is stored in a file and loaded into memory in the API function `iguard_init`. Eventually, every call to `iguard_start_protection` prefetches all these memory addresses to prevent them from causing cache and TLB misses during the guarded section. Additionally, we prefetch stack addresses during runtime by prefetching cache lines close to the current stack pointer. The treatment of stack addresses differs as these may change for each function invocation and hence cannot be predicted as easily.

F. Online Phase: Guarding Phase

With the profile and the prefetching offsets, the developer can ship the program to a production system where attacks can occur, e.g., a server running untrusted code. The production system also needs to run an instance of IRQGuard, i.e., the kernel module. The developer needs to add the thresholds from the profiling phase (cf. Section IV-D) to the PMC configuration file, whose filepath is specified as argument in `iguard_parse_pmc_config`.

Once the profiling mode of IRQGuard is disabled, its behavior changes as follows. During the guarding phase, the PMCs are set to interrupt mode, resulting in program interruption immediately once a threshold violation occurs. The actual monitoring of IRQGuard only takes place for the guarded section, i.e., the code between the functions `iguard_(start/stop)_protection`. IRQGuard resets and enables the PMCs in `iguard_start_protection` and stops them in `iguard_stop_protection`. More precisely, IRQGuard writes the value $PMC_{maxvalue} - threshold_{PMCn}$ into each PMC, where $threshold_{PMCn}$ corresponds to the configured threshold and $PMC_{maxvalue}$ is the maximum value a PMC before overflowing. Additionally, IRQGuard programs the counters such that an overflow raises a performance-monitoring interrupt.

Note that the PMCs are only programmed on the CPU core that the protected program runs on. Furthermore, note that a more sophisticated implementation can stop the PMCs when the operating system switches the context away from the protected program. Alternatively, the operating system can store the PMC value on each context switch and restore it if the program continues, effectively eliminating events from other applications. While our PoC implementation does not support that feature, we stress that a production-ready version can implement it in the same way as context switches are handled for the `perf` utility. While this feature requires either a custom kernel or support by the Linux kernel, it would further improve the precision of the IRQGuard approach, as such an extension would significantly reduce the impact of all unrelated programs on a system without any negative consequences.

IRQGuard registers an interrupt handler for the performance-monitoring interrupt (PMI) using the Linux kernel function `register_nmi_handler`. If the threshold for one of the PMCs is exceeded, i.e., a PMC overflows, the hardware triggers said PMI. This interrupt triggers the handler installed by the kernel module, which immediately sends a signal to the protected program. To make it convenient for the programmer, the signal is handled by a call to `setjmp`. This construct is wrapped into a macro that can be used inside `if`-conditions. Thus, all the programmer needs to do is to put the sensitive computations after an `if(!iguard_detected_attack)`. In the `else`-case, the threshold violation should be handled.

```
iguard_init();
iguard_parse_pmc_config("./pmc_config.iguard");
iguard_start_protection();
uint8_t secret = 0b01101001;
size_t idx = 0;
int i = 0;
while (i < N) {
  if (!iguard_detected_attack();){
    sync_with_attacker();
    if ((secret >> idx) & 1)
      for (int r = 0; r < REPS; r++) one_bit();
    else
      for (int r = 0; r < REPS; r++) zero_bit();
    idx = (idx + 1) % 8;
    i += 1;
  } else { break; }
}
iguard_stop_protection();
```

Listing 1: The victim program. Lines highlighted in *orange and italic* are added for the protected variant of the program, i.e., when IRQGuard is used to harden the program.

G. Post-Abort Actions

When IRQGuard stops the application, it allows the developer of the victim application to control the resulting action. Depending on the concrete scenario, different actions may be chosen. One option would be to stop the victim program in case of an attack. While this yields a denial-of-service gadget for an attacker, it is also a strong defensive approach as it prevents any further attack vector from this point onwards. Note that an adversary with local code execution can typically abuse simpler ways to disrupt the running system, e.g., by exhausting the system's resources.

Another less strict option capable of handling false positives is to wait with further actions until the n^{th} violation. Note that retrying until n violations occur exposes the application to a greater leakage by the attacker. Thus, n has to be selected carefully, depending on how much leakage is acceptable. Also, the handler can employ more complex heuristics, as shown in previous work [8], [40], [55], [54]. The handler can then decide to abort or continue the computation.

Instead of stopping the protected program entirely, one can apply additional mitigations or defenses. For example, cloud applications could isolate the hardware components of different tenants on a IRQGuard-reported violation. Also, mitigations that yield more runtime overhead can be activated, e.g., page coloring or related approaches for detecting microarchitectural attacks [67], [29], [43], [66], [10], [41], [40], [55], [56]. This scenario is especially beneficial if one wants to combine stronger defense mechanisms, which induce high-performance penalties, with IRQGuard. This way, the performance-heavy defense mechanisms do not have to run all the time but can be started on a IRQGuard-reported violation. Note that the amount of leakage may increase depending on the chosen mitigation strategy.

V. Evaluation

We evaluate the effectiveness and performance impact of IRQGuard. As IRQGuard stops ongoing attacks by preventing further leakage, we primarily evaluate at which point in time we stop the ongoing attack, i.e., how much leakage remains.

We evaluate the effectiveness based on a vulnerable program, referred to as the *victim program*. We intentionally create a victim program exposing a redundant and clear memory access pattern to simulate a highly-vulnerable target. The victim's control flow depends on a secret value that the attacker can infer by leaking the control flow using a side-channel attack. We evaluate how Flush+Reload (cf. Section V-A), Prime+Probe (cf. Section V-A), and TLBleed (cf. Section V-B) can leak the control flow and how IRQGuard can mitigate this. Listing 1 shows the main part of the victim program.

The victim program iterates over the bitstring `01101001`. We choose this bitstring as it contains an equal number of ones and zeroes while being less trivial than a simple alternating pattern. As the control flow depends on the secret, side-channel attacks that differentiate calls to `one_bit` and `zero_bit` can recover the secret. To further strengthen the attacker, the function `sync_with_attacker` signals the attacker when a new iteration begins. This yields a strong attacker model with a synchronized attacker and victim. This is reasonable as if IRQGuard can defend against a strong attacker, it also stops more realistic but weaker attackers.

In the following, we evaluate how the victim program performs in the presence of different side-channel attackers and how these attacks perform when the program is hardened by IRQGuard. Listing 1 includes the changes to the victim program when it is protected by IRQGuard as shown in orange font and italic. As described in Section IV, the guarded section of the program is annotated by calls to `iguard_(start/stop)_protection`. While one can further optimize this by only guarding the loop's body, this would require additional mental effort by the developer. If there is an attack, as indicated by `iguard_detected_attack` returning `true`, the victim program breaks its loop, avoiding further information leaks.

A. Cache Attacks

We demonstrate multiple side-channel attacks based on the CPU cache, i.e., cache attacks, and how IRQGuard mitigates them. We instantiate the victim program with `N=1000` and `REPS=1000`. As post-abort action (cf. Section IV-G), we choose to stop the program immediately. While this action may not be applicable for every scenario, we choose this action as any further leakage is completely dependent on the actions choosen for further mitigation. Furthermore, stopping the program restricts the evaluation on the leakage before IRQGuard notifies the system about the attack, which is the primary concern of this work.

Flush+Reload Flush+Reload [64] can leak the secret of the program by leaking from the I-cache. The attacker flushes a cache line contained in `one_bit` (or alternatively on `zero_bit`), waits, and accesses it, thus measuring its cache

TABLE I: Flush+Reload threshold stability experiments for varying CPUs.

CPU	Threshold	F-Score	Detection Time
Intel Xeon E-2176M	156 LLC mis.	1.0	2.09 bits
Intel Xeon E-2176M	175 LLC mis.	1.0	2.94 bits
Intel Xeon E-2176M	195 LLC mis.	1.0	3.92 bits
Intel Xeon E-2176M	214 LLC mis.	1.0	3.96 bits
Intel Xeon E-2176M	234 LLC mis.	0.99	3.97 bits
Intel Xeon E3-1505M v5	156 LLC mis.	1.0	2.66 bits
Intel Xeon E3-1505M v5	175 LLC mis.	1.0	2.98 bits
Intel Xeon E3-1505M v5	195 LLC mis.	0.98	2.94 bits
Intel Xeon E3-1505M v5	214 LLC mis.	1.0	3.03 bits
Intel Xeon E3-1505M v5	234 LLC mis.	0.98	4.42 bits
Intel Core i5-6400T	156 LLC mis.	0.95	3.22 bits
Intel Core i5-6400T	175 LLC mis.	0.96	3.96 bits
Intel Core i5-6400T	195 LLC mis.	0.95	4.0 bits
Intel Core i5-6400T	214 LLC mis.	0.97	4.09 bits
Intel Core i5-6400T	234 LLC mis.	0.98	4.76 bits

state. For improved accuracy, the attacker can mount the attack multiple times per bit.

After a threshold violation, the loop is exited, and no further cache accesses are leaked. For evaluation, we execute the victim in the unprotected scenario and the protected scenario (cf. Listing 1). The threshold is generated by taking the maximum of multiple runs without an ongoing attack and adding a $20\,\%$ error margin (cf. Section IV-D). In our experiment, this yields a threshold T of 195 LLC misses on an Intel Xeon E3-1505M v5 running Ubuntu 20.04 with Linux kernel 5.4.0. To evaluate different thresholds, we repeat the experiments with $T - 20\%$, $T - 10\%$, $T + 10\%$, and $T + 20\%$. We repeat the experiment on two different machines, one with a different microarchitecture. The exact thresholds and results are shown in Table I. We observe that when the threshold is violated, the leakage signal vanishes. When executing the attack 100 times, we observe that average leakage rate is reduced to the first 5 bits out of 1000, i.e., we reduce the attack's leakage by $99.5\,\%$. To evaluate the efficacy of IRQGuard, we additionally execute the experiment 100 times in which we remove the reload and flush step of the attacker, i.e., removing the minimal amount of attacker code that needs to be removed to disable the attack. We label IRQGuard stopping the attack in this case as a false positive, thus allowing us to calculate the F-scores for each threshold as shown in Table I. Furthermore, Table I shows how the threshold derived on one CPU acts when transfered to a different CPU. While, we observe that the threshold is slightly less effective on the Intel Core i5-6400T, it is still effective in stopping the attack after the first 5 bits. We conclude that while thresholds are transferable between CPUs, it is beneficial to generate thresholds on the target system to maximize the efficacy of IRQGuard. Note that this is still a realistic scenario as large cloud providers typically have large sets of homogeneous systems.

Prime+Probe Another cache-based attack that can be mounted against the victim program is Prime+Probe [44]. First, the attacker finds an eviction set for the function `one_bit` (or alternatively `zero_bit`). Afterward, the attacker repeatedly measures the time it takes to access the eviction set. If the access time increases, the attacker learns that `one_bit` was executed. Hence, the attacker can infer the current secret bit. We evaluate Prime+Probe on the shared L3 cache. The signal defaults to zero when the loop is aborted.

We evaluate this attack on the same Intel Xeon E3-1505M v5. After multiple profiling runs, we take the maximum observed value and add the $20\,\%$ margin, resulting in a threshold of 461 LLC misses. We execute the experiment 100 times. We observe that the attack is stopped after, on average, 163.94 of 1000 leaked bits, i.e., IRQGuard reduces the leakage by $83.6\,\%$. IRQGuard stops the attack in all 100 cases. Furthermore, we execute the attack 100 times without the prime step interfering with the victim's state, i.e., a benign version of the attack program. In this case, IRQGuard never reports a violation, resulting in an F-score of 1. While a remaining leakage of $16.39\,\%$ intuitively sounds high, an attack on a $2048\,\mathrm{bit}$ RSA key would still require breaking a $1712\,\mathrm{bit}$ key with this leakage. Note that while attacks like Heninger et al. [20] exist that recover an entire key from a partial key, even these attacks require more than $16.39\,\%$ of the key. Furthermore, such attacks require very small exponents, i.e., $e \leq 3$, which are not commonly used in practice.

To analyze the performance of our setting on a different machine, we execute the experiment a second time on an Intel Xeon E-2176M running Ubuntu 20.04 with Linux kernel 5.4.0. We again observe an F-score of 1, but IRQGuard stops the attack after, on average, 2 bits. Thus, we reduce the leakage by $99.8\,\%$. We conclude that IRQGuard reliably stops cache attacks on the LLC.

B. TLB Attacks

IRQGuard can defend programs against TLB-based attacks such as TLBleed [15]. We instantiate the victim program with `REPS=1000` and `N=1000`. To mitigate such attacks, IRQGuard uses performance-monitoring events correlating with TLB misses. On the tested Intel CPUs, TLB misses are counted by the `DTLB_LOAD_MISSES.WALK_STLB_HIT` performance-monitoring event. The attacker repeatedly evicts the TLB set mapping to the address of `one_bit` while observing the eviction time. If `one_bit` is accessed by the victim, this access evicts an entry of the attacker eviction set from the TLB. Thus, an attacker can infer from the access time whether the access has occurred. Note that the attacker executes on the hyperthread sibling of the victim's core.

We evaluate our attack on the same Intel Xeon E3-1505M v5. We run the experiment 100 times with the enabled attack and 100 times with a benign attacker program. We take the default threshold with the $20\,\%$ error margin described in Section IV. This procedure results in 137 TLB misses as a threshold. For the malicious attacker program, instead of a leakage of 1000 bits on the unprotected binary, the binary protected by IRQGuard does not leak any bits, i.e., the leakage is reduced by $100\,\%$. The reason we directly abort during the measurements of the first bit is that due to the noisy signal transmitted through the TLB, the attacker evicts the TLB multiple times during each run of the victim. For the benign

attacker program, we observe no false positives, resulting in an F-score of 1.

To evaluate the threshold on a second machine with the same microarchitecture, we rerun the experiment on an Intel Xeon E-2176M running Ubuntu 20.04 with Linux kernel 5.4.0. We also observe an F-score of 1. However, the attack is only mitigated after an average leakage of 23.97 of 1000 bits. Note that with a threshold of 137, even a theoretically ideal attacker can at most leak 137 '1'-bits, as at least one TLB miss is required for learning that a processed bit is a '1'-bit.

C. Reliability

We evaluate IRQGuard's reliability with two experiments. First, we measure how many instructions are still architecturally executed after the threshold is exceeded, i.e., a PMC overflowed. Second, we measure how long instructions may transiently execute afterward. Both these experiments evaluate how fast IRQGuard can stop the program after detecting a threshold violation.

Abort Timing We measure the time between exceeding the threshold of a PMC and aborting the normal control flow of the protected program. To reliably measure this, we use two PMCs. While the first PMC is used by IRQGuard and is configured with a threshold, the second PMC is used to monitor the progress of the program.

We configure the first PMC to track the performance-monitoring event `LONGEST_LAT_CACHE.MISS`, i.e., the event responsible for tracking the number of LLC misses. We configure IRQGuard to abort the normal protected program's control flow after 5 LLC misses. The sample program causes 5 cache misses followed by a sled of multiple `divps` instructions. The code for this can be found in Appendix A. We program the second PMC with the event `FP_ARITH_INST_RETIRED.128B_PACKED_SINGLE` to count the number of retired `divps` instructions. Thus, the second PMC reports how many instructions were executed after the threshold violation.

We run the code 3 times, two times as warm-up runs, to make sure that everything besides the intended cache misses results in a cache hit and a third time to measure the impact that we want to observe. We execute this experiment 1000 times on an Intel Xeon E-2176M running Ubuntu 20.04 with Linux kernel 5.4.0. For all runs, we observe that the event `FP_ARITH_INST_RETIRED.128B_PACKED_SINGLE` counts on average 3.5 (with a median of 0) retired instructions after the threshold violation. If we add an `mfence` instruction before the `divps` sled, we do not observe any retired `divps` instructions anymore, as expected. Note that this experiment suffers from the design limitation that we cannot reason about the potential retirement of all possible instructions, such as `nop` instruction. We conclude that, on a violation, IRQGuard diverts the instruction stream of the protected program almost immediately and orders of magnitude faster than related approaches (cf. Section V-E).

Transient Window After Violation To determine the number of transiently executed instructions after a threshold violation, we configure the performance-monitoring event `ARITH.DIVIDER_ACTIVE` on the second PMC while keeping the remainder of the previous experiment setup. `ARITH.DIVIDER_ACTIVE` counts the number of cycles the CPU's divider unit executes, including transient instructions. Note that as `ARITH.DIVIDER_ACTIVE` includes the workload of transiently executed instructions, it allows us to measure the length of the transient window in cycles. In line with the abort-timing experiment, we use the same machine and repeat the measurements 1000 times. For these 1000 runs, we observe an average activity of 192.49 cycles (with a median of 167). We conclude that IRQGuard can rapidly abort the protected program within only a few cycles after a threshold violation. This stopping time aligns with the Intel TSX-based attack mitigation of Gruss et al. [17].

D. Performance Impact

In this section, we evaluate the performance impact of IRQGuard on an Intel Xeon E3-1505M v5 with Ubuntu 20.04 running Linux kernel 5.4.0. First, we measure the overhead that the API calls introduce for the previously discussed victim program. Second, we evaluate the overhead introduced for other applications on the same system.

We measure the execution time of our victim program (cf. Listing 1) when IRQGuard is added and compare that to the execution time without IRQGuard. We run the victim program with varying values of N, i.e., we iterate over a different number of bits. We repeat each measurement 1000 times and observe that IRQGuard's performance overhead is below $0.64\,\%$ for the protected program. The precise numbers are shown in Appendix A.

We also evaluate how IRQGuard impacts the performance of unrelated programs using the performance benchmark SPEC CPU 2017. To create a realistic scenario for IRQGuard, we create a protected program requesting IRQGuard to program each PMC to the highest possible threshold. This threshold is unlikely to be reached by the protected program but suffices to show the performance overhead that active PMCs have on a system. As the execution time of SPEC CPU may still lead to an overflow of the PMCs, we just enable the profiling mode of IRQGuard. This prevents the program from stopping when an overflow is reached but expresses the same performance overhead. In parallel, we execute the SPEC CPU benchmarking suite. We execute SPEC CPU once with and once without IRQGuard running on the system. On average, IRQGuard introduces an overhead of $0.84\,\%$. Details are shown in Appendix A.

E. Comparison to Similar Tools

We analyze how IRQGuard compares to related tools from previous works that are based on PMCs. Related tools that are designed to protect applications against microarchitectural attacks are typically based on PMCs [31] and can be split into 4 major categories based on their underlying detection approach. These categories are static analysis of binaries [29], [32], ML-based anomaly detection systems [34], [10], [41], [40], [19],

[55], [54], threshold-based detection systems [43], [17], [66], and detection systems that mix the previous categories [50], [8].

Kosasih et al. [31] showed that most PMC-based mitigations are evaluated insufficiently and concluded that such mitigations can be evaluated by comparing the effectiveness, overhead, detection speed, and underlying threat model. While the effectiveness of most tools is evaluated using PoC implementations of microarchitectural attacks, the evaluation details strongly differ between these works, and most papers do not provide artifacts, making it hard to draw further conclusions. Note that the reported effectiveness of the best related tools align with our approach. Thus, we focus further comparison on the remaining aspects.

Table II provides a summary of approaches related to IRQGuard for protecting applications from microarchitectural attacks without requiring custom hardware. A crucial aspect of *proactively* protecting against attacks is how fast an approach can react to an attacker [31]. This is determined by the sampling rate and the time it takes to stop the execution of the attacker or victim program. Compared to previous work, our approach excels in this category as we leverage CPU exceptions and do not rely on sampling rates. This allows IRQGuard to abort the program orders of magnitudes faster than previous works. Other threshold-based approaches like Payer et al. [43] and Zhang et al. [66] rely on sampling intervals, making them slower than hardware-based approaches like ours. While this can partially be overcome by lowering the sampling intervals, this immediately increases the overhead of said tools and raises the question whether such slow sampling intervals are even feasible in a realistic environment. To stop cache attacks in a similar time window as IRQGuard, they would need to sample on every single cache miss, thus increasing the overhead by a large amount.

The only other tool that is in the same order of magnitude, when it comes to stopping an attack, is Cloak [17]. An essential limitation of Cloak is that it requires Intel TSX, which is only available on some Intel CPUs and was already deprecated [25]. Hence, Cloak cannot be used anymore on CPUs with up-to-date microcode. While Intel TSX can be force-enabled outside SGX, doing so makes the system vulnerable to Meltdown-type attacks [51]. Furthermore, Cloak only supports stopping after the first cache miss occurred. While IRQGuard can be configured the same way, it is beneficial to configure an error margin, which allows for handling a handful of unexpected events occurring. This allows IRQGuard to protect considerably longer code snippets while maintaining reliable results. For example, for protecting an RSA implementation, Cloak has to be inlined in the exponentiation loop of the decryption routine as it cannot handle the entire decryption routine [17]. However, IRQGuard can handle such a routine, which is demonstrated in Appendix A. Performance-wise, we observe that the overhead of our tool is in the same order of magnitude as the best comparable tools.

Another crucial aspect is the underlying threat model of the tools [31]. Related approaches typically monitor the entire system to identify and stop malicious processes. The drawback is that they have to differentiate between *all possible* benign and malicious workloads based on the performance characteristics. Kosasih et al. [31] demonstrate that this leads to problems for advanced adversaries as they can hide in the noise of benign processes. Our approach shifts the focus from the entire system to specific parts of the victim process, which attackers cannot alter. Thus, defenders know ahead of time what performance characteristics to expect, making it hard to evade IRQGuard. For example, while the camouflaged attack discussed by Kosasih et al. [31] does evade related approaches, it only impacts the performance footprint of the attacker process. Thus, it does not affect the effectiveness of IRQGuard, as the victim's behavior is unchanged. For a more thorough discussion on camouflaged attacks, we refer the reader to Section VII-A.

We conclude that our approach allows stopping the information leakage multiple orders of magnitude faster while having an overhead that is on par with related tools. This is achieved because we do not rely on software-based runtime monitoring or ML classifiers but directly leverage CPU hardware. Furthermore, by focusing our detection on the victim side instead of the attacker side, we make it harder for attackers to evade our approach as they cannot hide their attack code in a complex but benign program.

TABLE II: Comparison of IRQGuard with different tools.

Method	Tool	Overhead	Sampling Rate	Time to Stop
Static	[29]	none	n.a	n.a
	[32]	none	n.a	n.a
Machine Learning	[34]	"low"[2]	100 ms	≥100 ms
	[10]	"negligible"[2]	≤ 3 µs	0.2-0.64 ms
	[41]	0.3-11.3 %	10-100 enc.	≥10 encryptions
	[40]	1-2 %	50-100 enc.	≥50-100 encryptions
	[19]	7.7-24.88 %	1 ms	50 ms
	[55]	n.a	50 µs	≥50 µs
	[54]	4-30 %	1-5000 µs	≥1-5000 µs
Threshold	[43]	0-5.92 %	1 s	≥ 1 s
	[17]	0-248 %	instant	≤ 500 CPU cycles
	IRQGuard	0.08-8.4 %	instant	192 CPU cycles
	[66]	<5 %	100-1000 µs	120-5110 µs
Other	[50]	none	n.a.	0.5-312 min
	[8]	<5 % CPU util.	100 µs	1 ms

[2] These papers did not provide a concrete number.

VI. Case Studies

In this section, we demonstrate that IRQGuard can protect cryptographic and non-cryptographic real-world applications. We protect an AES implementation used in OpenSSL (Section VI-A) and based on the protected OpenSSL library, harden OpenSSH (Section VI-B) to demonstrate a real-world scenario for IRQGuard. In Section VI-C, we show the applicability to non-cryptographic applications by demonstrating IRQGuard on the command-line utility `toilet`. We evaluate all case studies on an Intel Xeon E3-1505M v5 running Ubuntu 20.04 with Linux kernel 5.4.0.

A. Prime+Probe on OpenSSL AES T-Tables

In this case study, we demonstrate IRQGuard on the AES T-table implementation of OpenSSL 1.0.1. This shows that IRQ-

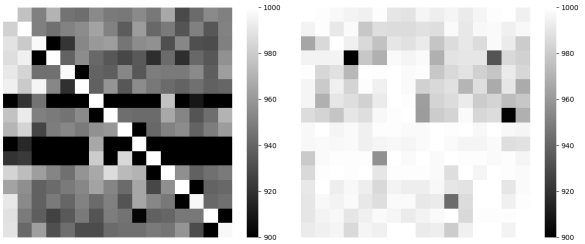

Fig. 2: Heatmaps of the leakage when attacking the unmodified and the protected variant of the OpenSSL AES encryption via Prime+Probe. The X-axis shows the different plaintext bytes, while the Y-axis shows the different cache lines. The key is encoded in the diagonal.

Guard can protect realistic libraries against microarchitectural attacks. This library is commonly used to demonstrate side-channel attacks [5], [42], [18], [28], [35], [16], [11], [53], [36], [9], [47]. The attacker exploits Prime+Probe to leak the secret AES key from the encryption routine. We add IRQGuard to the encryption routine and analyze how the attacker's performance decreases. Therefore, the attacker runs on the same CPU as the victim.

Overview The attacker mounts a Prime+Probe attack on the T-tables. Previous work has demonstrated that such attacks are feasible by leaking the T-table access patterns of every encryption round [5], [42], [18], [28], [35], [16], [11], [53], [36], [9], [47]. In line with previous work [65], we restrict the attack to the first encryption round. We add the IRQGuard API calls directly to the library at the start and end of the function `AES_encrypt`. Alternatively, developers could deploy IRQGuard from within their own program, i.e., wrapping the library calls. We mount the attack on the protected and unprotected library and use `LLC misses` as the event. Note that due to the prefetching of IRQGuard, the CPU caches all data apart from the single primed cache line of the attack. Without an ongoing attack, thus the entire working set is cached. Hence, we set the threshold to 1 LLC miss.

Results Figure 2 shows the leakage heatmaps when mounting the attack on the unmodified and the IRQGuard-protected variant of the AES encryption. For visualization purposes, we use the secret key consisting of all zeroes, which results in a visually diagonal access pattern. The key leakage for the unmodified version is clear enough for an attacker to infer the correct key bytes. In contrast, the protected library variant does not show a relevant access pattern. The runtime of the encryption increases by an average of $8.4\,\%$ or $4.25\,\mathrm{ms}$ ($n = 100$) for the protected variant. While this runtime overhead seems impactful, we stress that this is due to the low runtime of the operation and that the overhead is only in the orders of a few milliseconds. IRQGuard successfully mitigates the ongoing attack with a precision of 1.0 and a recall of 0.99. We conclude that IRQGuard can successfully be applied to real-world cryptographic libraries.

B. Hardening OpenSSH using IRQGuard

Based on the previous case study, we show that IRQGuard is capable of protecting real-world applications by hardening an OpenSSH server. We use the protected variant of OpenSSL (cf. Section VI-A), which OpenSSH uses to implement the AES ciphermode. We choose OpenSSH version 7.5 due to its compatibility with OpenSSL 1.0.1.

Overview As the peers of an SSH connection send and receive messages, we need to protect both the `AES_encrypt` and `AES_decrypt` functions. We take the setup of the previous case study and additionally add the IRQGuard API calls to the `AES_decrypt` function. Due to the additional overhead of the OpenSSH process compared to the previous case study, we increase the threshold to 18 LLC misses. The OpenSSH server sandboxes the processes responsible for receiving and handling incoming clients to protect against memory corruption attacks. As the default configuration of this sandbox prevents calls to the IRQGuard API, these calls either have to be allowlisted or the sandbox has to be disabled. For the sake of simplicity, we choose to turn off the sandbox for our experiment. A production-ready variant instead requires to allowlist the syscalls to the IRQGuard API in the Seccomp configuration of OpenSSH and to mount the device file of IRQGuard into the `chroot` jail.

Results To verify that the OpenSSH server works, we connect a client to the hardened OpenSSH server and execute the following commands: `id; ls /etc;cat /etc/passwd`. Each invocation of this command, including establishing the SSH connection, invokes the AES functions 878 times. After 50 repetitions, we observe that the hardened variant executes all commands successfully with an average runtime of $645\,\mathrm{ms}$. As the unprotected variant of OpenSSH takes an average runtime of $616\,\mathrm{ms}$, IRQGuard results in an overhead of $4.5\,\%$. We observe no false positive for any of the $43\,900$ AES invocations. We verify that a violation of the threshold stops the SSH connection, which results in a disconnect of the client with varying messages depending on whether the violation occurs during the connection buildup or afterwards. We conclude that IRQGuard is capable of protecting real-world software in a realistic scenario.

C. Flush+Reload on Text Rendering

In this case study, we show that IRQGuard is not limited to protecting cryptographic implementations. We protect the `toilet` command-line utility, which renders text as ASCII art. Additionally to being a non-cryptographic application, the victim also uses syscalls within the protected code snippet, demonstrating that this is not a problem for IRQGuard. The attacker mounts Flush+Reload on `libcaca`, the shared library used by `toilet` for rendering. The attacker and victim only have to share the LLC.

Overview The attacker tries to extract the input of `toilet`, i.e., the string that is rendered as ASCII art. Therefore, the attacker targets the function `caca_put_figchar`, the function responsible for rendering a glyph to an internal buffer that is subsequently written to the standard output. Within

this function, a loop iterates through all glyphs and exits early if the requested glyph is found. Hence, an attacker can mount Flush+Reload on this loop. The number of cache hits correlates with the glyph index in the font file. We use performance degradation on the victim to ensure the attacker can reliably distinguish the number of loop iterations. Although a library function causes the leakage, we apply IRQGuard in the main function of the `toilet` application. We add the calls to IRQGuard around the `render_list` function, which is responsible for the entire rendering, including creating the canvas, rendering the string, outputting the canvas, and cleaning up. We choose LLC misses as the performance-monitoring event. After profiling the application, we choose a threshold of 5000 LLC misses. On an attack, we directly abort the application instead of retrying to render the string.

Results Without an ongoing attack, the ASCII art is successfully rendered in $100\,\%$ of the executions ($n = 1000$). Hence, the regular usage of the tool is unaffected. We observe that the attack can distinguish any provided randomly chosen 3 letters in $100\,\%$ of the runs ($n = 100$). With IRQGuard, we never observe a successful attack. We conclude that IRQGuard can effectively mitigate side-channel attacks even on generic applications containing syscalls.

VII. Discussion

In this section, we discuss the limitations of IRQGuard, as well as the applicability to other systems.

A. Defending Against Camouflaged Attacks

As side-channel attacks are not actively exploited in the wild, we evaluate our tool on our implementation of these attacks. While this is in line with related work [31], real-world attackers can use evasion techniques. For example, Kosasih et al. [31] show that previous approaches can be circumvented by hiding the malicious payload in the performance overhead of a large process. Due to the design of our approach, i.e., the focus on the victim process instead of the attacker process, such an evasion has no influence on IRQGuard. Note that using the same approach for the victim process does not work, as the attacker cannot influence the victim's code. Alternatively, an attacker could slow down their attack to minimize its performance footprint. While this limits the capabilities of an attacker, due to the longer attack window, it makes it harder to detect the attack. Compared to previous tools that cannot detect an attack as soon as the attack is slow enough, our approach can still detect such attacks if the configured threshold is low enough. Depending on the scenario, the threshold for IRQGuard can be minimal, e.g., in Section VI-A, which makes it impossible for an attacker to mount an attack, no matter how stealthy.

B. Defending Against Further Attacks

We demonstrate the applicability of our approach for protecting against several types of microarchitectural attacks. While we do not explicitly demonstrate it, the design of IRQGuard allows protecting against further attacks, such as further variations of cache attacks and Spectre attacks. For example, to harden a program against Spectre-PHT attacks, developers can configure IRQGuard to monitor the state of the event `Branch Misses Retired`.

Two other attack types we do not discuss in the paper are Rowhammer and Meltdown-type attacks. While these attacks can be detected using PMCs [29], [39], the problem lies in the threat model of these attacks. For both attack types, the attacker attacks the system directly without interacting with a victim process, which IRQGuard can protect. This is in contrast to attacks in which the attacker targets the code of a victim process, which can be protected by IRQGuard. Thus, defending against these requires restrictions on the attacker's capabilities, e.g., when the attacker's code is restricted to a sandbox environment protected by IRQGuard. Further, it is challenging to come up with a meaningful profiling step.

PMCs have been abused for attacks [52], [12], [7], [60]. However, our approach only requires unprivileged access to PMC values of the own process, which is already possible on some distributions, e.g., Arch Linux. If only own counters can be configured, leaking PMCs using Meltdown 3a [23], [4], [60] is also not a security problem.

IRQGuard can only mitigate microarchitectural attacks if there is a PMC for the exploited event that can be used on the victim. However, for example, power side channels, such as PLATYPUS [37] or Hertzbleed [57], are not detectable using PMCs. Nevertheless, this limitation is not as severe as the similar limitation for PMC-based detections. For example, Flush+Flush [16] claims to be a stealthy side-channel attack, as it does not trigger a large number of cache-miss events in the *attacker*. Still, IRQGuard protects against Flush+Flush, as the *victim* suffers from cache misses.

C. Limitations of Performance Counters

While IRQGuard is a generic approach for mitigating microarchitectural attacks, the approach has certain limitations.

Performance-counter Reliability IRQGuard fundamentally relies on the correctness and precision of the underlying PMCs. Hence, if the used PMCs do not reliably count the microarchitectural events, the effectiveness of IRQGuard also suffers. The same problem is also inherent in other approaches using PMCs. Weaver et al. [58] showed that specific PMCs on some systems tend to overcount the counted event. Similarly, PMCs also count events in transient execution [24, Chapter 18, 19]. However, in combination with IRQGuard, this is not a security problem, as overcounting only leads to false positives and, thus, unnecessary retries of the protected code. Only undercounting could lead to a security problem, but we are unaware of such behavior on any CPU.

Performance-counter Selection The efficacy of IRQGuard fundamentally relies on the correct selection of performance-monitoring events. IRQGuard can only mitigate attacks that the selected events can reliably monitor. The best combination of events for this task is still an open research problem. While previous work has evaluated a large set of PMCs for detecting microarchitectural attacks [67], [29], [43], [66], [10],

[41], [40], [55], [56], the counters are often chosen based on intuition and experience instead of a systematic evaluation. For our PoC implementation, we also rely on these previously identified counters as they lead to good results. We leave a generic methodology for finding the most related PMCs for future work.

D. Other Microarchitectures

While we evaluate IRQGuard on Intel CPUs, the approach is not limited to them. It only has 3 requirements for the underlying microarchitecture. First, a mechanism must exist to monitor microarchitectural elements during attacks. Second, this monitoring must be able to raise an interrupt on a specific event. Third, it must be possible to configure this event from software. These basic requirements are fulfilled for various (micro-) architectures. We experimentally verify that PMC overflows can be used to raise an interrupt that can be caught in userspace, thus allowing a port of IRQGuard for the given architecture, on AMD Zen to Zen 3, Intel Sandy Bridge to Alder Lake, and an Arm Cortex A73. These properties are also documented for ARMv8 [3] and have been unofficially documented based on the Apple M1 [30]. Hence, IRQGuard can be implemented on a wide range of CPUs. While our PoCs rely on PMCs, IRQGuard could theoretically also work with other mechanisms. For example, a CPU could build such an interface specifically for IRQGuard.

We also test IRQGuard inside virtual machines. KVM with QEMU 6.2.0 emulates all required properties of PMCs in passthrough mode, allowing IRQGuard to work from within VM guests. Neither on AWS nor on Hyper-V could we get IRQGuard running. However, hypervisors support the required functionality, as verified with KVM, making it possible to use this approach in the cloud if supported by the cloud provider. By saving and restoring PMCs on VM exits and entries, respectively, hypervisors can ensure that no performance information from the hypervisor nor other VMs is exposed and that the IRQ can be forwarded to the correct VM.

VIII. Conclusion

We presented a novel approach, IRQGuard, which allows victims to proactively mitigate microarchitectural side-channel attacks by monitoring their own microarchitectural events. By detecting specific microarchitectural state changes in defined code regions and triggering interrupt requests, IRQGuard prevents information leakage almost instantly. We showed that our proof-of-concept implementation induces minimal overhead and works on Intel, AMD, and Arm. We showed IRQGuard's effectiveness on realistic code bases by applying it to the OpenSSL library and hardening an OpenSSH server. We further demonstrated IRQGuard on a generic Linux utility, including syscalls. With its novel use of interrupts, IRQGuard is a practical, low-overhead solution for mitigating a wide range of microarchitectural attacks, even against camouflaged attackers.

Acknowledgments

We want to thank our anonymous reviewers for their comments and suggestions. We thank Ruiyi Zhang for fruitful discussions and his help with the AES case study. We thank William Kosasih, Yusi Feng, Chitchanok Chuengsatiansup, Yuval Yarom, and Ziyuan Zhu for providing the code of their camouflaged side-channel attacks. This work was partly supported by the Semiconductor Research Corporation (SRC) Hardware Security Program (HWS) and by the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101020415).

References

[1] AMD64 Architecture Programmer's Manual, 2024.
[2] Onur Acıiçmez, Jean-Pierre Seifert, and Çetin Kaya Koç. Predicting secret keys via branch prediction. In *CT-RSA*, 2007.
[3] ARM. *ARM Architecture Reference Manual ARMv8*. ARM Limited, 2013.
[4] ARM. Cache Speculation Side-channels, 2020. Version 2.5.
[5] Daniel J. Bernstein. Cache-Timing Attacks on AES, 2005. URL: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.
[6] Daniel J Bernstein, Tung Chou, and Peter Schwabe. McBits: fast constant-time code-based cryptography. In *CHES*, 2013.
[7] Sarani Bhattacharya and Debdeep Mukhopadhyay. Who watches the watchmen?: Utilizing Performance Monitors for Compromising keys of RSA on Intel Platforms. *Cryptology ePrint Archive, Report 2015/621*, 2015.
[8] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. Cacheshield: Detecting cache attacks through self-observation. In *CODASPY*, 2018.
[9] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks. In *USENIX Security Symposium*, 2020.
[10] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. ePrint 2015/1034, 2015.
[11] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *USENIX Security Symposium*, 2017.
[12] Lizzie Dixon. Breaking KASLR with perf, 2017. URL: https://blog.lizzie.io/kaslr-and-perf.html.
[13] Vinodh Gopal, James Guilford, Erdinc Ozturk, Wajdi Feghali, Gil Wolrich, and Martin Dixon. Fast and Constant-Time Implementation of Modular Exponentiation. *Embedded Systems and Communications Security*, 2009.
[14] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures. In *NDSS*, 2020.
[15] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium*, 2018.
[16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*, 2016.
[17] Daniel Gruss, Felix Schuster, Olya Ohrimenko, Istvan Haller, Julian Lettner, and Manuel Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *USENIX Security Symposium*, 2017.
[18] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium*, 2015.
[19] Berk Gulmezoglu, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. FortuneTeller: Predicting Microarchitectural Attacks via Unsupervised Deep Learning. *arXiv:1907.03651*, 2019.
[20] Nadia Heninger and Hovav Shacham. *Reconstructing RSA Private Keys from Random Key Bits*. 2009.
[21] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX. In *CHES*, 2020.

[22] Intel. Intel 64 and IA32 Architectures Performance Monitoring Events, 2017. Revision 1.0.
[23] Intel. Instructions affected by rogue system register read, 2018. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/resources/instructions-affected-rogue-system-register-read.html.
[24] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide, 2023.
[25] Intel. Intel Transactional Synchronization Extensions (Intel TSX) Memory and Performance Monitoring Update for Intel Processors, April 2023. URL: https://www.intel.com/content/www/us/en/support/articles/000059422/processors.html.
[26] Intel Corporation. Pin - A Dynamic Binary Instrumentation Tool, 2012. URL: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.
[27] Intel Corporation. Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations, 2020. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html.
[28] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES. In *S&P*, 2015.
[29] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Mascat: Preventing microarchitectural attacks before distribution. In *CODASPY*, 2018.
[30] Dougall Johnson. Apple Firestorm/Icestorm CPU microarchitecture docs, 2021. URL: https://github.com/dougallj/applecpu.
[31] William Kosasih, Yusi Feng, Chitchanok Chuengsatiansup, Yuval Yarom, and Ziyuan Zhu. SoK: Can We Really Detect Cache Side-Channel Attacks by Monitoring Performance Counters? In *AsiaCCS*, 2024.
[32] Minkyung Lee and Jin Kwak. Detection Technique of Software-Induced Rowhammer Attacks. *CMC*, 2021.
[33] 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*, 2017.
[34] Congmiao Li and Jean-Luc Gaudiot. Detecting malicious attacks exploiting hardware vulnerabilities using performance counters. In *COMPSAC*, 2019.
[35] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*, 2016.
[36] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors. In *AsiaCCS*, 2020.
[37] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. PLATYPUS: Software-based Power Side-Channel Attacks on x86. In *S&P*, 2020.
[38] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*, 2015.
[39] Samira Mirbagher-Ajorpaz, Gilles Pokam, Esmaeil Mohammadian-Koruyeh, Elba Garza, Nael Abu-Ghazaleh, and Daniel A Jiménez. Perspectron: Detecting invariant footprints of microarchitectural attacks with perceptron. In *IEEE MICRO*, 2020.
[40] Maria Mushtaq, Ayaz Akram, Muhammad Khurram Bhatti, Maham Chaudhry, Vianney Lapotre, and Guy Gogniat. Nights-watch: A cache-based side-channel intrusion detector using hardware performance counters. In *HASP*, 2018.
[41] Maria Mushtaq, Jeremy Bricq, Muhammad Khurram Bhatti, Ayaz Akram, Vianney Lapotre, Guy Gogniat, and Pascal Benoit. WHISPER: A Tool for Run-time Detection of Side-Channel Attacks. *IEEE Access*, 2020.
[42] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA*, 2006.
[43] Matthias Payer. HexPADS: a platform to detect "stealth" attacks. In *ESSoS*, 2016.
[44] Colin Percival. Cache Missing for Fun and Profit. In *BSDCan*, 2005.
[45] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium*, 2016.
[46] Thomas Pornin. Why Constant-Time Crypto?, 2022. URL: https://www.bearssl.org/constanttime.html.
[47] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In *CCS*, 2021.
[48] Srivaths Ravi, Anand Raghunathan, and Srimat Chakradhar. Tamper resistance mechanisms for secure embedded systems. In *VLSI Design*, 2004.
[49] Chester Rebeiro, A. David Selvakumar, and A. S. L. Devi. Bitslice Implementation of AES. In *Cryptology and Network Security (CANS)*, 2006.
[50] Majid Sabbagh, Yunsi Fei, Thomas Wahl, and A. Adam Ding. SCADET: A Side-Channel Attack Detection Tool for Tracking Prime+Probe. In *ICCAD*, 2018.
[51] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*, 2019.
[52] Leif Uhsadel, Andy Georges, and Ingrid Verbauwhede. Exploiting hardware performance counters. In *5th Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC'08).*, 2008.
[53] Stephan Van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In *USENIX Security Symposium*, 2018.
[54] Han Wang, Hossein Sayadi, Gaurav Kolhe, Avesta Sasan, Setareh Rafatirad, and Houman Homayoun. Phased-guard: Multi-phase machine learning framework for detection and identification of zero-day microarchitectural side-channel attacks. In *ICCD*, 2020.
[55] Han Wang, Hossein Sayadi, Avesta Sasan, Setareh Rafatirad, and Houman Homayoun. Hybrid-shield: Accurate and efficient cross-layer countermeasure for run-time detection and mitigation of cache-based side-channel attacks. In *ICCAD*, 2020.
[56] Han Wang, Hossein Sayadi, Avesta Sasan, Setareh Rafatirad, Tinoosh Mohsenin, and Houman Homayoun. Comprehensive Evaluation of Machine Learning Countermeasures for Detecting Microarchitectural Side-Channel Attacks. In *GLSVLSI*, 2020.
[57] Yingchen Wang, Riccardo Paccagnella, Elizabeth He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. Hertzbleed: Turning power side-channel attacks into remote timing attacks on x86. In *USENIX Security Symposium*, 2022.
[58] Vincent M Weaver, Dan Terpstra, and Shirley Moore. Non-determinism and overcount on modern hardware performance counter implementations. In *ISPASS*, 2013.
[59] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. Osiris: Automated Discovery of Microarchitectural Side Channels. In *USENIX Security*, 2021.
[60] Daniel Weber, Fabian Thomas, Lukas Gerlach, Ruiyi Zhang, and Michael Schwarz. Reviving meltdown 3a. In *ESORICS*, 2023.
[61] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. Stacco: Differentially Analyzing Side-channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves. In *CCS*, 2017.
[62] Yuval Yarom. Mastik: A Micro-Architectural Side-Channel Toolkit, 2016. URL: https://cs.adelaide.edu.au/~yval/Mastik/.
[63] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA Nonces Using the FLUSH+ RELOAD Cache Side-channel Attack. *Cryptology ePrint Archive, Report 2014/140*, 2014.
[64] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*, 2014.
[65] Ruiyi Zhang, Taehyun Kim, Daniel Weber, and Michael Schwarz. (M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels. In *USENIX Security*, 2023.
[66] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. Cloudradar: A real-time side-channel attack detection system in clouds. In *RAID*, 2016.
[67] Yinqian Zhang and MK Reiter. Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *CCS*, 2013.

APPENDIX

Table III shows the evaluation results of the performance overhead of a system when IRQGuard enabled the PMCs on the system. Results are given by the SPEC CPU 2017 benchmark.

Table IV shows the default PMC configuration used in IRQGuard. Performance-monitoring events marked with

TABLE III: SPEC CPU 2017 benchmarking results.

Benchmark	SPEC Score		Overhead
	Baseline	IRQGuard	[%]
600.perlbench_s	5.71	5.71	0.00 %
602.gcc_s	8.36	8.37	0.12 %
605.mcf_s	6.42	6.54	1.87 %
620.omnetpp_s	3.88	3.88	0.00 %
623.xalancbmk_s	4.47	4.48	0.22 %
625.x264_s	4.71	4.71	0.00 %
631.deepsjeng_s	3.94	3.95	0.25 %
641.leela_s	3.59	3.59	0.00 %
648.exchange2_s	10.40	10.50	0.96 %
657.xz_s	2.23	2.33	4.95 %
Average			0.84 %

TABLE IV: Default PMC configuration used in IRQGuard.

PMC	Programmed Event	Associated Attacks
PMC0	LLC Misses (**A**)	(L3 and flush-based) Cache Attacks
PMC1	DTLB_LOAD_MISSES.WALK_STLB_HIT	TLB Attacks
PMC2	MEM_LOAD_RETIRED.L1_MISS	(L1) Cache Attacks

A are architecturally defined events; other events are microarchitecture-specific [24, Chapter 19], i.e., their exact name may differ for different microarchitectures.

Table V shows the runtime overhead of the code snippet in Listing 1 when executed with different values for N.

The code in Listing 2 measures the number of executed and retired instructions after a threshold violation. IRQGuard is configured to stop after 5 cache misses, i.e., in line 7. Afterward, the divider is used by the `divps` instructions. The divider usage is monitored via the PMC event `ARITH.DIVIDER_ACTIVE`.

In this case study, we show that IRQGuard also allows protecting more complex cryptographic implementations by protecting an older version of the library GnuPG (1.4.13) and its RSA implementation, which is vulnerable to Flush+Reload. Note that using exactly this GnuPG version aligns with related work [64], [62], [38] and thus eases comparison. The attacker targets an RSA decryption process that the attacker can query with arbitrary ciphertexts. While the attacker and victim use shared memory, they run on different physical CPU cores and only share the LLC.

TABLE V: The runtime overhead of the code in Listing 1.

Iterations (N in Listing 1)	**IRQGuard Overhead**	**Runtime Default** (median)	**Runtime with IRQGuard** (median)
10	0.64 % ± 1.64	321.52 ms	323.41 ms
20	0.29 % ± 1.60	322.53 ms	323.48 ms
40	0.33 % ± 1.62	323.32 ms	324.39 ms
60	−0.12 % ± 1.83	325.14 ms	324.89 ms
80	0.15 % ± 1.83	325.95 ms	326.27 ms
100	0.29 % ± 1.73	326.15 ms	327.03 ms
200	0.46 % ± 1.68	328.89 ms	330.61 ms
300	0.39 % ± 1.67	333.86 ms	334.60 ms
400	0.08 % ± 1.64	338.34 ms	338.26 ms
500	0.19 % ± 1.52	341.62 ms	342.22 ms

```
1  ; rax = pointer to 5 consecutive
2  ;       uncached memory pages
3  mov rbx, [rax]
4  mov rbx, [rax + 4096]
5  mov rbx, [rax + 8192]
6  mov rbx, [rax + 12288]
7  mov rbx, [rax + 16384]  ; <- 5th cache miss
8                          ;    violates threshold
9  divps xmm0, xmm1
10 divps xmm0, xmm1
11 divps xmm0, xmm1
12 [...]
```

Listing 2: Code to measure the number of executed and retired instructions after a threshold violation. IRQGuard is configured to stop the program after 5 cache misses.

Overview The attacker targets the GnuPG functions `mpih_sqr_n` and `mul_n`. As the `mul_n` function is only called if the currently-processed secret bit is '1', the attacker can leak the secret by monitoring the access to those functions. To mitigate this attack, we take the most straightforward approach for guarding the code of the library and start the guarding phase (cf. Section IV-F) directly after entering the function `secret`, which contains the decryption routine of the RSA implementation. For the error handling, we also choose the most simplistic variant and abort execution on an attack. For further possible actions, we refer to Section IV-G. We choose LLC misses as an appropriate performance-monitoring event to monitor. After profiling `secret` 100 times, we see a PMC value of 705 LLC misses with a maximum of 2586. We determine 3000 LLC misses as an appropriate threshold for IRQGuard.

Results For the unprotected library, we observe that our attack, on average, recovers 2030.95 bits, i.e., 99.6 % of the secret key ($\sigma = 6.8, n = 100$). When protected by IRQGuard, the leakage signal only exists at the beginning, i.e., before the configured threshold is violated, as the protected library stops the encryption on a violation. Consequently, an attacker can only leak 29.45 bits, on average, before exceeding the threshold ($\sigma = 0.5, n = 100$). IRQGuard mitigates the attack with a recall and a precision of 1. IRQGuard causes a runtime overhead for the RSA encryption routine of 0.3 %.