Finding and Exploiting CPU Features using MSR Templating

Andreas Kogler* Daniel Weber† Martin Haubenwallner* Moritz Lipp‡ Daniel Gruss* Michael Schwarz†

*Graz University of Technology †CISPA Helmholtz Center for Information Security ‡Amazon Web Services

2022 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-1316-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/SP46214.2022.9833599

***Abstract*—To ensure backward compatibility while adding new features to CPUs, CPU vendors enable a limited CPU configuration via so-called model-specific registers (MSRs). These MSRs have been introduced for various features, such as debugging, performance monitoring, or security. While many MSRs are documented, there is still a plethora of undocumented or sparsely documented MSRs in modern CPUs. Furthermore, with multiple hundred MSRs, each providing up to 64 configuration bits, it is tedious to find specific configuration options.**

In this paper, we show that MSRs and their configuration bits can be detected automatically on Intel and AMD CPUs. We introduce MSRevelio, a framework to automatically detect bits that influence the behavior of instructions and semi-automatically find bits controlled by BIOS settings. We show that previously overlooked bits can harden systems against microarchitectural attacks such as Medusa, CrossTalk, and software-prefetch attacks. Additionally, we show that an undocumented lock bit allows disabling AES-NI at runtime, forcing mbedTLS to fall back to an AES implementation vulnerable to cache attacks. Exploiting this fallback inside an SGX enclave, we fully recover the AES key used by the enclave. With our detection approach, we show that security features retrofitted with microcode updates can be easily detected, even before the public documentation of the underlying vulnerability. In our analysis of the Xen hypervisor, we show that Xen's handling of MSRs was flawed for a long time, allowing guests to access undocumented and unhandled MSRs and fingerprint specific Xen versions. Using automated correlation analysis between documented and undocumented MSRs, we discover a previously undocumented MSR correlating with the CPU's timestamp counter. This MSR is also accessible from Xen guests, and we demonstrate a Foreshadow attack when all other timers are unavailable or artificially deteriorated. Our results highlight that transparency is crucial for features interacting closely with CPU internals.

I. Introduction

With nearly every new CPU generation, CPU vendors add new features to their CPUs. While some of these features are architectural, such as new instruction-set extensions [20], [31], often features are more related to the microarchitecture, such as mitigation options for transient-execution attacks [35], [37]. Such microarchitectural features can often even be retrofitted to existing CPUs using microcode updates [20]. The non-architectural features typically require some form of interaction with the CPU. Typically, these CPU features are exposed via model-specific registers (MSRs). MSRs are special registers that can be read from and written to by privileged code, *i.e.*, the operating system (OS). Every MSR has a unique 32-bit address and a size of 64 bits. Generally, MSRs are used for interaction with the CPU, such as enabling and disabling CPU features, debugging, and performance monitoring.

While CPU vendors publicly document many MSRs, there are also undocumented MSRs or bits inside documented MSRs that are not documented. These MSRs might only be used internally to debug or reveal information that CPU vendors do not want to disclose [28], [24]. Undocumented MSRs have been shown to undermine CPU security. The AMD K8 CPU provided an MSR that enabled a debug mode [21]. Similarly, Domas [24] found an MSR on the VIA C3 CPU that allows enabling a so-called "god mode". When enabling this mode, unprivileged applications can execute special CISC instructions that circumvent all privilege checks of the CPU. Some of these undocumented MSRs are mentioned in patents, but there is no clear description of what they do or how they can be used. Moreover, even for documented MSRs, not all bits are fully documented, *i.e.*, reserved bits that have effects.

MSRs can also be used to add security features to CPUs. For example, mitigations for Spectre [49], Foreshadow [72], ZombieLoad [70], RIDL [75], or CrossTalk [64] have been implemented using MSRs [37]. These MSRs control the speculation behavior and provide the possibility to clear several caches and buffers. All of them have been introduced with microcode updates to retrofit mitigations to older CPUs. In the case of OS support, it can query this functionality via the `cpuid` instruction or the IA32_ARCH_CAPABILITIES MSR and then use the features via the corresponding MSRs [37].

In this paper, we introduce MSRevelio[1], a framework that automatically detects available MSRs, regardless of whether they are documented or not. Our approach generates a list of readable and writable MSRs for a specific CPU. We compare the list of detected MSRs with the documented MSRs on Intel and AMD CPUs and classify the MSRs into documented, partly documented, and undocumented. This analysis reveals that all tested CPUs have a large number of undocumented MSRs. On the evaluated AMD CPUs, the number of undocumented MSRs even exceeds the number of documented MSRs. In addition to scanning MSRs, we also automatically analyze the detected undocumented MSRs. We sample the values of both documented and undocumented MSRs. Based on these samples, we automatically correlate undocumented with documented MSRs for a probabilistic classification of

[1]Find MSRevelio's source code at https://github.com/IAIK/msrevelio

DOI 10.1109/SP46214.2022.00110

undocumented MSRs. Our approach is more robust than the timing-based approach suggested by Domas [24] that assumes similar MSRs expose similar access times.

We also use MSRevelio (Section III) for a semi-automated analysis of different BIOS versions (Section IV). In this use case, we search for BIOS settings influencing the value of such (partly) undocumented MSRs, and indicating the MSR's purpose. Using MSRevelio, we scan for changes in undocumented MSR bits when modifying BIOS settings and measure potential impacts on instructions. By grouping instructions and collecting performance-counter readings, we identify possible effects of MSR bits on the group's instructions. This approach can also be used to search for all MSR configuration bits that impact a specific instruction. Based on these results, we present six security-relevant case studies.

We demonstrate that MSR bits influencing instruction behavior can mitigate but also introduce new security issues. While these bits could also be found in a manual analysis, MSRevelio alleviates the analysis substantially. We discover an MSR bit converting software-prefetch instructions to no-operations, mitigating software-prefetch attacks on AMD [27], [51]. Additionally, our approach finds a bit to trap `cpuid`, reducing the attack surface for CrossTalk [64]. Moreover, by templating BIOS features, we detect that fast-string support can be disabled at runtime, reducing the impact of Medusa [61]. By unsetting the undocumented AES-NI lock bit, we can disable AES-NI at an arbitrary time within the SGX threat model, leading to a time-of-check-to-time-of-use vulnerability forcing mbedTLS [8] to fall back to a vulnerable AES implementation exploitable by a side-channel attack. We show the feasibility of this attack by recovering the full AES key from a single memory-access trace.

We use the found MSRs as a template for tracking the change of MSRs over different CPU microcode versions. Microcode cannot only modify but also add entirely new MSRs. We automatically detect which MSRs have been added in specific microcode versions and whether the MSR was, later on, removed again with a microcode update. Based on our analysis, we can clearly detect which microcode version added mitigations for transient-execution attacks. We even discover CPUs for which such MSRs have been introduced months before the vulnerability was publicly disclosed and the MSR was documented. We cross-check all detected MSRs with official documentations to discover that all added MSRs are related to security features, showing that this approach can leak information about potential embargoed security vulnerabilities.

We also show that the Xen hypervisor just recently prevented the guest OS from accessing undocumented MSRs [17], [62]. Instead of using an allow list for MSRs that should be accessible to a guest, Xen relied on a block list to allow access to all MSRs except a few. With our automated approach, we show how this blocklist evolved over different versions of Xen. This allows fingerprinting of the Xen version even if the hypervisor prevents access to that information or if anti-VM detection methods are applied [48]. In the final case study, we show that the blocklist-based approach allows guests on older Xen versions to potentially access security-relevant MSRs of the host system. Our correlation analysis reveals a previously unknown MSR available to Xen guests that correlates with known timers. Leveraging this MSR yields a high-resolution timer, even if other timers are unavailable, e.g., because the hypervisor uses Fuzzy time [30], [76] or restricts parallel execution, preventing counting threads. To verify that the discovered MSR can act as a timer, we demonstrate a Foreshadow attack [72] using this MSR.

Our results show that MSRs directly impact the system's security. Access to certain MSRs can have negative consequences in the cloud, as they might re-enable attacks that were thought mitigated. Other MSRs, however, can also be used to mitigate attacks for which only costly software workarounds are available, such as prefetch-based attacks [27], [51]. As we found these MSRs on all systems affected by the corresponding vulnerability, they can be used as a short-term solution until the vulnerability is fixed in hardware.

To summarize, we make the following contributions:

1) We demonstrate an automated approach to detect undocumented MSRs and MSR bits on Intel and AMD CPUs and their impact on instructions and system functionality.
2) We show how our detected MSRs harden systems against microarchitectural attacks but also enable new attacks.
3) We show that the block-list approach used in the Xen hypervisor poses a risk to the system security by allowing guest access to undocumented MSRs, demonstrating a new timing primitive for side-channel attacks.
4) We analyze the evolution of MSRs over microcode versions, showing silent additions of security-related MSRs.

Responsible Disclosure: We responsibly disclosed our findings to Intel on August 3rd, 2021. Intel acknowledged our findings.

Outline: Section II provides background. Section III introduces MSRevelio, a framework to automatically find and classify MSRs. Section IV extends MSRevelio with "BIOS templating" to pinpoint features in MSRs. Section V demonstrates security implications of MSRs in six case studies. Section VI discusses limitations. Section VII concludes.

II. Background

In this section, we provide background about MSRs, Intel SGX, microcode, and transient-execution attacks.

A. Model Specific Register (MSR)

MSRs are special CPU registers, allowing interaction with low-level CPU features and advanced configuration of the CPU's behavior. Modern x86 CPUs have hundreds of MSRs [40], [5]. However, there is usually only sparse public documentation [24], *i.e.*, many MSRs are not publicly documented, and for many MSRs, the function of specific bits is not mentioned or not precisely defined. MSRs are accessed using the two privileged instructions `rdmsr` and `wrmsr` for reading and writing the 64-bit MSRs. Each register is addressed using a unique 32-bit address. Hypervisors restrict MSRs to prevent the guest systems from taking over control of the host. As

MSRs are typically implemented in microcode, they can be removed or added, and their behavior can be updated via CPU microcode updates. For instance, recently, MSRs have been used to add security mitigations against Spectre [49], Foreshadow [72], ZombieLoad [70], and CrossTalk [64] attacks.

B. Intel SGX

Intel SGX (Software Guard Extensions) is an instruction set extension providing a trusted-execution environment (TEE) for Intel CPUs. The SGX threat model, similar to other TEEs, assumes that even privileged software such as the OS, administrative users, and peripheral hardware may be compromised and behave maliciously. The trusted code is separated from the untrusted code into a so-called enclave. Enclaves operate within an encrypted and isolated memory region so that even the OS or a physical attacker cannot access the unencrypted memory contents. However, Intel considers vulnerabilities in enclaves the responsibility of the enclave developer, including software side channels [11], [69], and software bugs like race conditions [80], [68]. Enclaves are launched within a regular application and can be interrupted by the OS at any point.

C. Micro-op Performance Profiling

With the rising complexity of out-of-order execution CPUs, profiling the performance of actual executed code is non-trivial. To get insight into the resources allocated and events triggered inside a CPU, vendors introduced Performance Monitoring Counters (PMCs). With these counters, a user can monitor the execution of instructions more precisely. However, mapping the observed events to a given instruction is complex, as the CPU splits instructions into smaller micro-ops.

NanoBench [2] is a framework designed to measure the exact PMCs of single instructions. The framework compiles measurement code from a given assembly snippet which allows minimizing the external measurement noise. In addition to the automatic measurement code generations, the framework also handles filling the CPU pipeline with a known state to allow for the same base conditions for all measurements.

D. Transient-execution Attacks

With out-of-order and speculative execution, a CPU can lazily handle exceptions and predict the outcome of computations, e.g., the target of an indirect jump, to reduce pipeline stalls. When the CPU has to handle an exception or mispredict a computation, the pipeline's state is rolled back to the instruction causing the exception or misprediction. As rolled back instructions are never committed to the architecture, they are referred to as transiently executed [16], [44].

Spectre [49] and Meltdown [54] showed that attackers can abuse transiently-executed instructions to leak data, *i.e.*, so-called transient-execution attacks [16]. An attacker encodes the results of a transient computation into a microarchitectural element that is not rolled back, e.g., the CPU cache. After discovering transient-execution attacks, multiple such attacks have been published [29], [16], [56], [61], [70], [75], [14], [72], [81], [64], [73]. Microarchitectural data sampling (MDS) attacks [70], [75], [14], [61], [64] are a subclass of transient-execution attacks leaking values from internal components of the CPU, e.g., the line fill buffer. In these attacks, an attacker brings the CPU into a state where transiently executed instructions compute with stale or incorrect values of internal buffers or caches. Leaking the values of these components allows leaking data across all security boundaries.

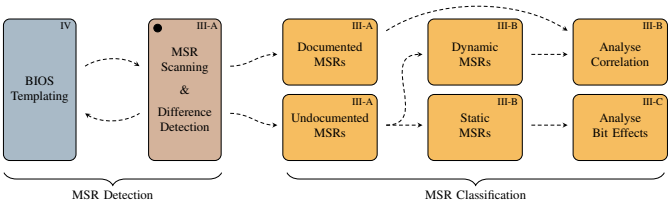

Fig. 1: The general structure of MSRevelio's analysis steps.

E. Microcode

Modern CPUs frequently receive updates to react to security concerns or bugs. Hence, manufacturers need a mechanism to update the behavior of CPU instructions or components. The microcode is an additional layer of abstraction between the actual hardware and the Instruction Set Architecture (ISA), which allows altering the internal behavior of CPU instruction to a certain extent [50]. Furthermore, some complex instructions require so-called microcode assists, which then execute a sequence of micro-ops read from the microcode [20], [70].

III. MSRevelio

This section describes MSRevelio, a framework to automatically find, classify, and analyze MSRs. MSRevelio aims to find undocumented features of MSRs that ultimately impact the security of the system. The overview of the steps of MSRevelio is shown in Figure 1. First, MSRevelio scans the potential MSR address space (cf. Section III-A). This scan obtains a list of available MSRs that are automatically classified into read-only, write-only, and read-writable MSRs. This list is filtered based on the official documentation of the CPU, resulting in documented, undocumented, or partly documented, *i.e.*, some bits are undocumented, MSRs. The MSRs that are not or only partly documented are recorded to classify them into two groups. Dynamic MSRs that change over time are correlated with documented MSRs to find similarities (Section III-B), or even aliased MSRs. For static, unchanging MSRs, MSRevelio analyzes the bits to determine whether toggling the bit impacts the behavior of instructions (Section III-C). In such a case, we can manually investigate whether such a change is security-relevant. Finally, we extend MSRevelio to find the influence of certain BIOS configurations on MSRs (Section IV) to gain additional information on undocumented MSR bits.

A. Detecting Undocumented MSRs

The MSR range of modern CPUs is continuously extended to provide additional functionality or adapt to new security flaws. Due to the large 32-bit address space, most of the

addresses are either not used and do not provide any functionality or are reserved for future extensions. However, there are more MSRs available than officially documented. To find these undocumented MSRs, MSRevelio uses both the `rdmsr` and `wrmsr` instruction. Both instructions raise a General Protection Fault (GP fault) if the CPU does not physically back the MSR address. The reason that we use both instructions is that there are 4 different MSR types: read- and writable, read-only, write-only, and non-present MSRs. By combining reads and writes to these MSRs, MSRevelio can detect all types of MSRs over the entire 32-bit address space of the MSRs.

1) Design: Detecting the presence of an MSR is not influenced by the MSR scope or the current privilege level. The `rdmsr` and `wrmsr` instructions already require OS privileges, and each core has the same set of accessible MSRs [39]. Our approach cannot access certain restricted MSRs that are only readable in SMM mode, e.g., MSR `0x9e`. However, as not even a ring-0 attacker can use them, we do not consider this a huge limitation. As every CPU core exposes the same MSRs, MSRevelio can search the MSR address space in parallel, significantly decreasing the execution time of the profiling. The framework first tries to read an MSR and catches any generated GP faults. The second test is a write to the MSR, again catching generated GP faults. Based on occurred faults, MSRevelio distinguishes between read-only, read- and writable, write-only, and non-existing MSRs. The framework stores the addresses of existing MSRs for later analysis.

To compare the discovered list of available MSRs to the official documented MSRs, MSRevelio additionally implements a PDF parser for the PDF-only documentation. As the structure of the MSR tables in both the Intel and AMD documentation is consistent, we can automatically find and extract the information from these tables. In addition to all documented MSRs, the parser extracts undocumented and reserved bits of documented MSRs. MSRevelio compares the discovered MSRs with the parsed documentation to determine undocumented or only partly documented MSRs.

2) Implementation: MSRevelio is implemented as a Linux kernel module with an additional user-space library. It uses the `rdmsrl_safe` and `wrmsrl_safe` kernel functions to catch GP faults when reading and writing MSRs. MSRevelio tries not to alter the content of the MSRs by writing the value that was read before. Only for write-only MSRs, this is not possible and MSRevelio conservatively tries to write '0' to such MSRs. For most of the writeable MSRs, the documentation [41], [5] states that when writing to the MSR, undocumented bits must be '0' to prevent a GP fault. This behavior is also necessary, as MSRs can only be overwritten with a 64-bit value and not bitwise. A complete scan when using multiple cores on our *AMD Ryzen Threadripper 1920x* with 5244 accessible MSRs takes $5.74\,\text{min}$ ($\sigma_{\bar{x}} = 0.0005$, $n = 10$).

B. Classifying MSRs

The classification functionality of MSRevelio is divided into two parts, namely the detection of static and dynamic MSRs and the further analysis of undocumented dynamic MSRs

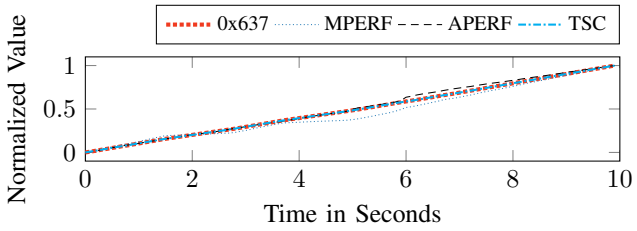

Fig. 2: Normalized values of the undocumented MSR(`0x637`) and the highly-correlating MSRs MSR(`0xe7`) (MPERF), MSR(`0xe8`) (APERF), and MSR(`0x10`) (TSC) (all monotonic counters) over $10\,\text{s}$.

based on the correlation with other documented dynamic MSRs. Note that for this analysis, we can only use MSRs that are readable. We define *static MSR* as an MSR with a fixed value that only changes if the MSR is actively written to, e.g., MSRs containing configuration bits. A *dynamic MSR* is an MSR that is continuously updated by the CPU, e.g., counters or sensor values. To distinguish static from dynamic MSRs, MSRevelio samples the values of the respective MSRs for a certain amount of time to detect if the value changes at some point. While an MSR classified as a dynamic MSR is always a dynamic MSR, MSRevelio might classify some dynamic MSRs as static MSRs if the value updates only with a very low frequency. However, for analyzing the impact on instruction behavior (cf. Section III-C), we are only interested in static MSRs as well as all write-only MSRs, as they are static in its nature, *i.e.*, they do not change their value. Hence, as static MSRs are always classified as static MSRs, this only results in some additionally tested MSRs, but no missed MSRs.

We further classify the found undocumented dynamic MSRs by cross-correlating them with all documented dynamic MSRs. For the correlation, we continuously sample all, *i.e.*, documented and undocumented, dynamic MSRs for $10\,\text{s}$ while executing a CPU stress test in parallel. The stress test triggers spikes in electricity and temperature sensor readings and triggers changes in the power states. As a result, undocumented MSRs exposing such states are easier to correlate with existing documented MSRs, as they contain more features for the correlation. Each resulting sample set of every undocumented dynamic MSR is then correlated with every documented dynamic MSR using the Spearman and the Pearson coefficient. For every undocumented MSR, MSRevelio generates a list of documented MSRs, sorted by the correlation coefficient, and a plot of all the sampled values. Figure 2 illustrates such a result, showing the undocumented MSR(`0x637`) and the documented MSR(`0xe7`) (IA32_MPERF), MSR(`0xe8`) (IA32_APERF) and MSR(`0x10`) (IA32_TIME_STAMP_COUNTER). While there are more computationally expensive methods to compare time series [10], [65], we do not require such complex algorithms, as the recorded data points are aligned. Hence, a simple correlation analysis is sufficient. The correlation analysis results in a list of similar MSRs, allowing to judge the likely information source of the MSR, e.g., whether it contains

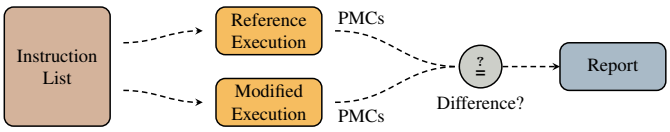

Fig. 3: MSRevelio changes bits in the undocumented MSR and measures various instruction groups' performance counters to see if the bit influenced the instructions.

thermal readings or a counter. In Section III-D, we show that this approach can find undocumented MSRs and detect the type of values exposed by the found MSR.

C. Impact on Instruction Behavior

To further analyze static MSRs, MSRevelio analyzes the impact of MSR bits on instructions. The goal of this scan is to find undocumented or reserved bits that influence the behavior of instructions. The framework performs the scan on the static MSRs (cf. Section III-B), as fluctuating MSR bits are usually not used as feature-control bits.

1) Design: To automatically detect changes in instruction behavior, MSRevelio uses performance counters for templating. To ensure that MSRevelio not only finds effects triggered by single bit flips but instead also finds effects that result from enum MSR fields, *i.e.*, groups of bits inside an MSR that belong to the same configuration option, we rely on optimized *flipping masks*. The masks tests all possible values of all possible enum fields with size $\leq W$ within an MSR by only performing 2^W writes to the MSR. For each observed effect the MSR is further analysed to find the exact enum field. Figure 3 shows the general concept of MSRevelio's bit scan. In the first step, a ground truth is recorded: MSRevelio executes a set of instructions on the CPU and records instruction-related performance counters. If for a set of instructions there is a change in the performance-counter values, it is an indication that one of the altered MSR bits affect the instruction. An alternative design could iterate over all possible bit values of an MSR instead of only considering the bounded enum fields. We did not choose this approach for two reasons.

First, such a design would require significantly more measurements to be performed. For example, on the tested Intel Core i5-4570, we found 3612 writable bits spread across 177 undocumented MSRs. In this case, our approach (with $W = 4$) only needs to test 2832 MSR values and additionally 496 tests per observed side effect to further find the root cause of the effect, whereas a full search would require $16\,853 \times 10^{12}$ tests (the calculations are shown in Appendix A). Testing the entire search space for the writable reserved bits of the documented MSRs on this processor would require only $16\,712 \times 10^6$ tests. One could argue that for the latter amount of tests, it is feasible to parallelize the tests using a cloud provider's resources. However, doing so is costly. At the time of writing, the cheapest on-demand CPU that AWS offers for its data center in Frankfurt costs $\$\,0.0047$/hour [6]. As our current implementation takes around 3 seconds per test, we can run 1200 tests for $\$\,0.0047$ and hence the complete search space enumeration for the reserved bits would cost $\frac{16712 \cdot 10^6}{1200} \cdot \$\,0.0047 = \$\,65\,455$. Note that these costs are only for testing a single CPU and abstract away the challenge to find a cloud provider offering the target CPU.

Second, it is reasonable to assume that flipping additional bits does not hide the effect of other bits in most cases. It is not beneficial to have one bit for enabling a feature and another for disabling it again. Instead, a practical implementation would use a single bit to toggle the activation of a feature, as the structure of most documented MSRs shows [41], [5]. An exception for this are enum MSR fields consisting of multiple bits to encode more than two values. Thus, we choose *flipping masks* to find enum MSR fields of up to W bits. This approach is not a perfect fit for every possible scenario, e.g., if an enum field is spread over more than W bits or contains unrelated bits in the middle. However, our approach acts as a trade-off between the scenarios we cover and the search space.

2) Implementation: To execute the instruction and record the performance counters, MSRevelio uses the nanoBench framework [2]. The nanoBench framework allows recording performance-counter events of a given assembly code snippet. The framework takes care of compiling the snippet and repeating it multiple times without introducing additional overhead. We test 124 common instructions divided into 16 groups based on their semantics. We show the groups and link the used performance-counter config in Appendix B. MSRevelio calls the nanoBench framework for each group separately. Hence, MSRevelio knows precisely which category of instructions the bit flip inside the MSR influenced. To cope with the large search space of undocumented MSR bits combined with the nanoBench framework invocation, MSRevelio pre-filters the undocumented bits in three phases based on their behavior.

Phase 1: Detecting Writable Bits. In the first phase, MSRevelio iterates over all the undocumented MSRs in parallel and records the bits inside the MSRs that can be toggled. This is done by reading the original MSR value once and then toggling one bit at a time and detecting if the `wrmsr` instruction executes successfully. While reserved or unimplemented bits typically raise a GP fault, some bits silently ignore the write. Hence, MSRevelio rereads the MSR value and checks if the bit was toggled. If we can successfully write to an MSR but reading from the MSR always faults, we consider the MSR as write-only and the bit as modifiable. Since setting arbitrary undocumented bits inside an MSRs can lead to various system freezes and undefined CPU states, the framework relies on a blocklist of such MSRs. This phase is also used to filter out bits that cause system freezes to enhance later execution time. Note that this process can also be fully automated using a remotely-controllable power switch [24].

Phase 2: Initial Recording of the Changed State. In the second phase, MSRevelio starts recording possible effects on the instruction groups. Our implementation tests all possible combinations of enum MSR fields consisting of up to 4 bits. We assume that multiple undocumented enum MSR fields are independent with respect to effects on instructions. With this assumption, this phase does not test each individual enum

TABLE I: MSRevelio's results for different microarchitectures, including the number of found and undocumented MSRs and the categorization of the undocumented MSRs. The number of similar MSRs indicates if an undocumented dynamic MSR can be correlated with a likeliness of more than 85% to a documented dynamic MSR.

CPU	μ-Arch	μ-Code	# Found (RW, RO, WO)	# Undocumented (RW, RO, WO)	# Static (RW, RO)	# Dynamic (RW, RO)	# Similar
AMD Ryzen Threadripper 1920X	Zen	0x8001137	5244 (5223, 17, 4)	4876 (4873, 2, 1)	4873 (4871, 2)	2 (2, 0)	0
Intel i7-6700k	Skylake	0x9e	477 (363, 108, 5)	105 (68, 35, 2)	99 (68, 31)	4 (0, 4)	2
Intel i7-8700k	Coffee Lake	0xb4	517 (388, 122, 7)	126 (89, 35, 2)	121 (89, 32)	3 (0, 3)	3
Intel i9-9900k	Coffee Lake	0xde	537 (413, 117, 7)	136 (99, 35, 2)	132 (99, 33)	2 (0, 2)	2
Intel Xeon Silver 4208	Cascade Lake	0x5003102	1109 (957, 142, 10)	647 (591, 52, 4)	601 (553, 48)	42 (38, 4)	42

MSR field, but instead alters the undocumented MSR by toggling different enum fields at the same time. In fact, it is possible to test all combinations of 4 consecutive bits at an arbitrary position inside the MSR using only 2^4 = 16 different MSR values. We use *flipping masks* containing a 1 in case a bit has to be flipped at that respective position in the MSR. We create the 16 masks by flipping the bits at position n after having generated $2^{n \bmod 4}$ masks, *i.e.*, we flip bit 0 after every generated mask, bit 1 every second, bit 2 after every fourth, and bit 3 after every eight masks, e.g., the second mask has the following representation: `0x1111111111111111`.

Due to this construction, given an arbitrary position of 4 consecutive bits, the 16 bitmasks represent all 16 values of these 4 consecutive bits (see Appendix C). This optimization reduces the search time by a factor of $\frac{16+60\cdot 8}{16}$ = 31 compared with an optimized sliding window, *i.e.*, testing all the enum values at a given position and then shifting the window by one while reusing previous results. The results of this phase are candidate MSRs where at least one of the changed bits affects the instructions. These MSRs are the basis for the third phase.

Phase 3: Finding the Origin of the Effects. After the second phase, MSRevelio has a list of MSR candidates that have observable effects on the instructions groups. In this phase, MSRevelio iterates over each of these candidates and sequentially tests all enum MSR fields to pin down the effects observed in the performance counters to a specific enum field and its value. This phase's results contain detailed information about the MSRs and which bits influence a certain instruction group. We further analyze the automated results of this scan in Section III-D and in six case studies in Section V.

D. Results

We conducted an exhaustive search for undocumented MSRs on a total of 5 CPUs. The overall results of these CPUs are shown in Table I. We found 5890 undocumented MSRs on AMD and Intel CPUs, with most of the discovered MSRs (4876) on AMD CPUs. However, 96.8 % of the found static, read-and-writeable, undocumented MSRs on AMD do not raise a GP-fault when written but ignore the written value, restricting further bit behavior analysis. We also observe similar behavior for 54.1 % of the Intel MSRs. We analyzed all these undocumented MSRs for correlations with documented MSRs. We found 53 undocumented MSRs that expose continuously changing values correlating with existing MSRs. For example, the dynamic MSR(`0x637`) exposes a monotonic counter correlating with documented counters, and we further explore this counter in Section V-F. For the static MSRs, we conducted the enum field search to find bits that influence specific instructions. We found 1 undocumented and 6 partially documented bits that affect instructions such as `cpuid` and `prefetch`. The effects of these MSR bits are analyzed in Section V. To determine the specific functionality of the MSR, manual analysis is necessary.

IV. Detecting OS-Configurable BIOS Features

In this section, we extend the MSR scanner of MSRevelio to detect differences in static MSRs that are caused by changes in BIOS settings. In line with Intel's documentation [34], BIOS refers to firmware, regardless if it is an actual BIOS or UEFI. The BIOS is responsible for configuring multiple CPU settings on boot, e.g., available features [20] and settings related to the power management. While some settings can only be changed by the BIOS at boot time, other features can also be modified by the OS. Many BIOS versions, e.g., on consumer systems, expose only a small subset of settings to the user. With this approach MSRevelio can template BIOS versions to hint the user on how to reenable *unlocked* features from the OS. Additionally, it can be used to analyze whether undocumented or poorly-described BIOS features impact MSRs.

A. BIOS Templating

Our approach produces a list of MSRs and bits inside these MSRs configured by the BIOS, including documented and undocumented bits. For this purpose, we template BIOS features that modify MSRs by changing a BIOS setting manually and automatically scanning all MSRs' values (cf. Section III-A). As we target features, we are only interested in static MSRs. After toggling a BIOS setting, we compare the values of all static MSRs to the values of the initial scan of all MSRs (cf. Section III-B). If an MSR has a different value, we assume that the BIOS setting led to the change of this MSR. We can increase the certainty that this MSR value depends on the BIOS setting by repeating the process multiple times.

To further focus on undocumented settings, MSRevelio uses the list of documented MSRs and their bits to check whether the changed bits are documented (cf. Section III-A). If either an undocumented MSR or an undocumented bit change, MSRevelio reports this as an undocumented feature. This undocumented feature is analyzed for impacts on the instruction behavior in the same way as MSRs obtained from a full MSR scan (cf. Section III-C). Again, to reduce the likelihood of uncorrelated changes, repeating the process multiple times increases the probability that the reported MSR bit is indeed related to the BIOS feature.

B. Setup

We evaluate 5 systems with BIOSes exposing a rich set of features. The tested CPUs are a Celeron J4005, Core i7-6700K, Core i7-8565U, and Core i7-10510U, with an Intel JYGKLCPX.86A.0053.2019.1015.1510, AMI 2.17.1246, HP R93 01.01.06, and an AMI 2.21.1277 BIOS, respectively. These BIOSes include options that are not documented in the BIOS manual and for which we did not even find any unofficial documentation, e.g., "Strong Weak Leaker" or "K1 off". We initially set all the BIOS values to their defaults and use MSRevelio to get the difference in the MSR availability and the MSR values after changing specific settings.

C. Results

Our scans reveal several BIOS options that directly affect MSRs. While several MSRs are locked after the BIOS initialization, some of them can be modified by the OS to emulate these BIOS settings. We also discovered BIOS settings that affect undocumented MSRs or MSR bits.

1) Documented Settings: Some of the changed MSRs are documented and simply expose the read-only status of the BIOS setting. Most settings, including VTx/VTd, turbo boost, fast-string support, or execute disable, are reflected in the MSR(`0x1a0`), documented as IA32_FEATURE_CONTROL. While the BIOS locks the configuration bits for VTx/VTd, the fast-string support and turbo boost can be changed by the OS. The execute-disable feature can theoretically also be modified by the OS. However, this only led to OS crashes on our machines. To point out the impact of the unlocked bits, we show in Section V-D that the OS can harden a system against Medusa [61] using the feature bit for fast-string support.

Another documented setting is the enabling and disabling of hardware prefetch features. For this setting, the BIOS simply modifies MSR(`0x1a4`) (MSR_MISC_FEATURE_CONTROL), which is also writable by the OS [77]. However, our tested BIOS versions only disabled the L2 prefetcher and not the L1 prefetcher when setting "Hardware Prefetcher" to "disable". While we consider this at least misleading, if not a bug, we do not see any security problem in that behavior.

2) Unofficial or Undocumented Settings: Our approach detected MSRs that are either entirely undocumented, not officially documented, or not documented for the microarchitecture on which we found the MSR. Such MSRs include MSR(`0x621`), MSR(`0x35`) (MSR_CORE_THREAD_COUNT), MSR(`0x7a`), and MSR(`0xe2`) (MSR_PKG_CST_CONFIG_CONTROL). MSR(`0x35`) provides information on the state of hyperthreading on Intel Xeon CPUs. However, while it is not documented for Intel Core CPUs, it also works on our Intel Core machines. MSR(`0x621`) is not publicly documented, but mentioned as MSR_UNCORE_PERF_STATUS in a book by Gough et al. [26] for Xeon E3/E5 CPUs without further details. Based on the description of the BIOS setting that modified bit 0 of this MSR, we learn that it is the state of Intel SpeedStep. We discovered another potential bug in one of our BIOS versions concerning bit 0 of MSR(`0x7a`). The BIOS provides an option "MachineCheck" which toggles exactly this bit. While we did not find any official documentation, the CoreBoot source [19] and a Linux kernel patch [66] suggest that this bit enables Intel SGX, which is also supported on our machine. A "Timed MWAIT" feature in our BIOS enables bit 31 in MSR(`0xe2`), which is officially reserved. We assume that this enables an `mwait` extension to continue execution after a specified number of CPU cycles have elapsed, similar to AMD's `mwaitx` instruction [4]. We leave it as future work to reverse engineer how this feature can be leveraged and its impact. When enabling and disabling AES-NI via the BIOS, it changes bit 1 in MSR(`0x13c`). For this MSR (MSR_FEATURE_CONFIG), the documentation states that 2 bits are used to represent the AES-NI state, without providing a detailed description. On all analyzed machines, bit 0 was always set to '1'. To showcase the security impact of this finding, Section V-A shows that bit 0 is actually a lock bit that can be exploited to attack SGX enclaves by disabling AES-NI at runtime.

V. Case Studies

This section presents six case studies demonstrating the security impact of previously overlooked MSR bits. We show that undocumented MSRs can prevent existing attacks and re-enable mitigated attacks in certain scenarios. We also detect security-relevant MSRs in microcode distributed before the vulnerability is disclosed. Finally, we show that specific hypervisors version expose distinguishable MSR fingerprints and provide access to security-relevant undocumented MSRs.

A. Exploiting the AES-NI Lock Bit

MSRevelio's BIOS templating (cf. Section IV) revealed that MSR(`0x13c`) (MSR_FEATURE_CONFIG)'s lowest two bits, which enable AES-NI, either contain the value '1' or '3'. The Intel SDM [41] documents that if these bits are '3', the AES instructions are not available until the next reset, otherwise, they are available. Also, the SDM notes that if the bits are not equal to '1', the instructions can be misconfigured. However, the individual behavior of these bits is not documented.

We observe that the MSR value cannot be changed via the `wrmsr` instruction from within the OS. This indicates that the BIOS locks the MSR after finishing the initialization. As bit 0 is always set on all tested machines, we assume that this bit is the lock bit of the MSR, set by the BIOS to restrict further changes. We verify this assumption by modifying the BIOS to not set the bit in this MSR, which is in line with the threat model of SGX [20]. With the unlocked MSR, we show that an attacker can modify the feature-detection mechanism of a securely-designed SGX enclave using mbedTLS to fall back to an insecure cryptographic-algorithm implementation, allowing the full extraction of the AES key via Prime+Probe.

1) Threat Model: Where available, the AES-NI instructions are used in cryptographic libraries to implement the AES algorithm securely and efficiently [31], [8]. These libraries are often combined with Trusted Execution Environments (TEEs)

to protect the implementation of cryptographic algorithms and establish secure communication with other parties. Furthermore, the threat model of SGX protects an enclave from a malicious OS and even malicious BIOS firmware [20]. In this threat model, an attacker can modify the BIOS [9].

We consider two distinct attack scenarios. First, we consider a system under complete attacker control. Here, the attacker tries to extract a secret key used by a targeted enclave. In this scenario, the attacker modifies the BIOS only on the attacker's machine to remove the lock bit. Second, we envision a scenario where the MSR is not initialized at all, e.g., because the BIOS developer was not aware of this MSR. While we did not encounter such a BIOS on any of our tested machines, there is a chance that such a BIOS exists due to a large number of BIOS vendors and the wide variety of BIOS versions.

2) BIOS Modification: To verify that the first bit of MSR(`0x13c`) is the actual lock bit of the AES-NI instructions, we patch the BIOS of our test system. For this case, we use a *MINISFORUM X35G* mini-PC with an *AMI* BIOS and an *Intel Core i3-1005G1* CPU. Dumping and flashing the BIOS is possible via the official AMI Firmware Update tool [7]. Alternatively, an attacker can simply use a low-price SPI flasher such as a CH341A if no software tool is available or if such a tool does not allow flashing a modified image. We extract all 253 binaries of the BIOS image using *UefiTool* [67] and disassemble them using Ghidra with the firmware utilities plugin [46]. In our BIOS, the MSR(`0x13c`) is initialized in the silicon init (SiInitFsp) module. Depending on the BIOS version (cf. Appendix D), the `wrmsr` is either inlined or encapsulated in a wrapper function. In both cases, we simply patch the initialization of the `EAX` register to not set bit 0. The patch for multiple BIOS versions is provided in Appendix D. For the second scenario, we replace the `wrmsr` (or the call to the wrapper) with `NOP` to leave the MSR uninitialized.

3) Behavior Verification: After booting both images, bit 0 of the MSR reads as '0'. For both BIOS modifications, the `cpuid` instruction still reports that AES-NI is available. Writes with the `wrmsr` instruction to the second bit of MSR(`0x13c`) are reflected by the `rdmsr` instruction, verifying that with a bit 0 cleared, bit 1 is not locked. Furthermore, setting bit 0 using `wrmsr` prevents any subsequent changes to the MSR. Hence, the bit 0 is indeed the lock bit of this MSR. In addition to acting as a lock bit, bit 0 is also the "apply" bit. Changes to the second bit only take effect after the lock bit is set. Therefore, the two BIOS modifications behave the same because the CPU ignores the second bit until the lock bit is set. If both bits are set, AES-NI is disabled, and the instructions raises an illegal instruction exception as expected.

4) AES-NI inside Intel SGX: We demonstrate the security implications of changing the AES-NI availability at runtime on the mbedTLS library [8]. Due to its small codebase and side-channel resistant AES-NI implementation, it is often used inside Intel SGX [8]. In mbedTLS, the CPU feature detection is performed over the `cpuid` instruction. Due to the restricted SGX environment, the `cpuid` instruction is not available inside enclaves. Therefore, the SGX-SDK uses an OCALL to retrieve the CPUID information from outside the enclave [33], leading to a potential attack vector manipulating the read CPUID leaf. To enable a robust CPU feature check, enclave developers can rely on executing a potentially not supported instruction and configuring an exception handler to catch the exception [58]. If the instruction executes without raising an exception, the hardware supports the given CPU instruction, otherwise, the exception handler is used to continue execution safely. With this mechanism, the feature detection is encapsulated inside the enclave and does not rely on untrusted data.

Second, a developer might know about the limitations of mbedTLS's fallback algorithm and check the availability of AES-NI with the secure feature-detection in the enclave's initialization phase and abort if AES-NI is not enabled. Furthermore, we assume developers do not account for changing feature bits like the AES-NI-enable bit during runtime, as the possibility of such behavior is not documented.

5) Proof-of-concept Attack: The attacker enables AES-NI in the BIOS and leaves it enabled during the initialization of the enclave. Hence, any feature check for the availability of AES-NI, be it through an OCALL to `cpuid` or using the trusted CPUID library [58], detects the availability of AES-NI. Even if the presence of AES-NI is enforced through some kind of attestation, the default enabled AES-NI state without the lock bit passes this check. However, an attacker can disable the AES-NI instruction set at any point by simply interrupting the enclave and modifying the MSR. With precise execution control of SGX enclaves, e.g., using SGX-Step [74], an attacker can target a specific instruction after which the AES-NI instructions are disabled. As a result, this leads to a time-of-check-to-time-of-use vulnerability for SGX enclaves that check for AES-NI and later on use it, as is the case for the mbedTLS library (Version 2.26.0). In case AES-NI is not available, mbedTLS falls back to a software-based AES implementation, which is not side-channel resistant [71], as it uses key-dependent memory accesses (cf. Appendix E).

6) Prime+Probe on SGX: This case study shows that an attacker can extract secret information via a cache attack from an SGX enclave. Since the SGX environment ensures flushing of the L1 cache during an enclave exit, and the SGX attestation can verify that hyperthreading is disabled, an attack using the L1-cache is unlikely. Therefore, an attacker needs to perform Prime+Probe on the last-level cache (LLC). We assume that an attacker uses precise execution control, e.g., SGX-Step [74], for Prime+Probe on the LLC [60], [11].

We simulate a Prime+Probe attack on mbedTLS, to show that a single trace suffices to exploit the AES-NI misconfiguration and leak the secret key. We accurately simulate cache sets, using the Intel Pin tool [32] to record the memory accesses and extract the corresponding cache set. As we only consider virtual addresses, we only extract the lower 6 bits of the cache set. However, this is already sufficient to recover the key. Figure 4 shows the cache-set accesses for two AES keys.

MbedTLS' AES implementation leaks the key in two different functions. First, in the `mbedtls_aes_setkey_enc` function responsible for the key schedule. Second, in the

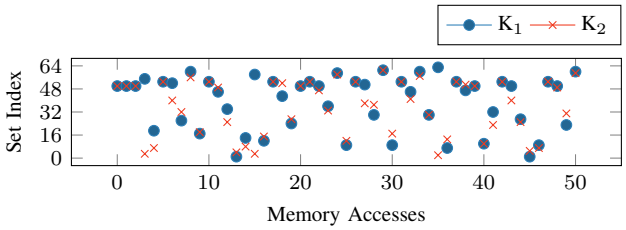

Fig. 4: Truncated secret-dependent cache-set accesses of mbedTLS' AES-128 implementation for two different keys.

`mbedtls_aes_encrypt_internal` function performing the T-table-based encryption. We consider two attacks, both with a *single* simulated Prime+Probe trace. If the attacker only records the encryption function, we require a known plaintext to recover the key. However, if the key schedule is included, we extend the attack to even recover parts of the plaintext. Note that we can exchange the encryption and known-plaintext attack with the decryption counterparts.

We analyze the trace with the Z3 solver [22] over 10 000 simulated Prime+Probe attacks on mbedTLS' AES-128 implementation with randomly generated keys and plaintexts. The solver recovers all keys using a known-plaintext attack, where each recovery only takes seconds. With unknown plaintext, the solver needs at most 74.08 min ($\sigma_{\bar{x}}$ = 0.367, n = 1200) to recover the key and additionally 10 bytes of the plaintext. The solver can recover even more plaintext bytes, however, the performance depends on the used key and plaintext. The performance evaluation used as the known plaintext's bytes, the higher bytes 10 to 15. The solver always finds the correct key without additional candidates. Hence, if an attacker can disable AES-NI at runtime, they can force mbedTLS to a path vulnerable to side-channel attacks, and extract the key.

B. Mitigating Software Prefetch Attacks on AMD

In this case study, we present the first software mitigation on AMD systems against prefetch-based side-channel attacks [27], [51]. Prefetch-based KASLR breaks exploit the runtime difference of the prefetch instruction for mapped and unmapped addresses, effectively derandomizing the kernel location. The prefetch KASLR break is an important part in the recent "Spectre in the Wild" exploit [79] to find the addresses of targeted kernel structures. As this is the only known full microarchitectural KASLR break on AMD CPUs, it is desirable to prevent this type of attack. Furthermore, Lipp et al. [51] exploit prefetch on AMD to break fine-grained KASLR, monitor kernel activity and to leak kernel memory using Spectre. With MSRevelio, we search and discover an MSR that disables the prefetch instructions on AMD systems. The prefetch-disabling bits can be set by the OS or a privileged user to prevent all prefetch-based side-channel attacks and, therefore, remove a building block for sophisticated attacks.

1) Threat Model: We assume a system without software bugs in the kernel and enabled KASLR. We further assume an unprivileged attacker with code execution on the system.

TABLE II: The bits of MSR 0xc0011029 were found with the instruction analysis of MSRevelio.

MSR 0xc0011029	Description	Effect
Bit 2	disable `PREFETCHNTA`	-1 LdDispatch
Bit 3	disable `PREFETCHT0`	-1 LdDispatch
Bit 4	disable `PREFETCHT1`	-1 LdDispatch
Bit 5	disable `PREFETCHT2`	-1 LdDispatch
Bit 6	disable `PREFETCHW`	-1 LdDispatch
Bit 7	disable `PREFETCH`	-1 LdDispatch

The system does not expose the kernel offset via system interfaces, e.g., `/proc/kallsysms`. Thus, the attacker uses the prefetch-based KASLR break to mount attacks or extract data from the kernel, e.g., using a Spectre attack [79], [51].

2) MSR Discovery: With the knowledge that hardware prefetchers can be disabled [3], we suspect that there might also be a possibility to disable software prefetchers. Hence, we leverage MSRevelio to automatically find MSR configuration bits that influence the software-prefetch instructions (cf. Section III-C). On an AMD Ryzen Threadripper 1920X CPU, MSRevelio discovered the MSR(`0xC0011029`). As Table II shows, bits 2 to 7 inside the MSR alter the behavior of the prefetch instructions as detected by MSRevelio. MSRevelio found these bits due to the reduced `LsDispatch.LdDispatch` performance counter by exactly one load compared to the reference (cf. Section III-C). This MSR is a "tweak" MSR that is used in errata to circumvent CPU bugs [78]. Although this MSR is not documented for the Zen microarchitecture (family 17h), we find the MSR in the extensive list of documented MSRs for the Bulldozer microarchitecture (family 15h) [3] (page 590), where these bits are documented as disabling the software prefetch instructions. Hence, by setting these bits, the OS can disable each of the six variants of the `prefetch` instructions individually.

3) Mitigate Prefetch Attacks: For our experimental setup, we use the file-based Linux MSR interface to disable all the software prefetch instructions. For evaluation, we build a PoC implementation of a prefetch-based KASLR break. The kernel is located in one of 512 possible virtual address offsets [15]. The PoC measures the execution time of the `prefetch2` instruction for all these possible virtual addresses. For every address, the KASLR break measures the execution time of 10 000 prefetch invocations executed in a loop. The loop is repeated 100 times, and the minimum of all the tries is recorded. If the kernel is mapped at the prefetched location, the execution time is higher (cf. Figure 5).

We execute the KASLR break on an AMD Ryzen Threadripper 1920x @ 3.8GHz with Ubuntu 20.04 LTS and Linux 5.4.0-74, with prefetch instructions enabled and disabled. Figure 5 shows the difference between the two invocations of the KASLR break. We observe that the KASLR break can precisely locate the kernel (offset 88) in the enabled case and fails to locate the kernel otherwise. Furthermore, we compare the execution time of `prefetch` when disabled with the execution time of a single byte `nop` instruction. The `nop` instruction takes 0.886 cycles ($\sigma_{\bar{x}}$ = 0.0092, n = 512 000 000)

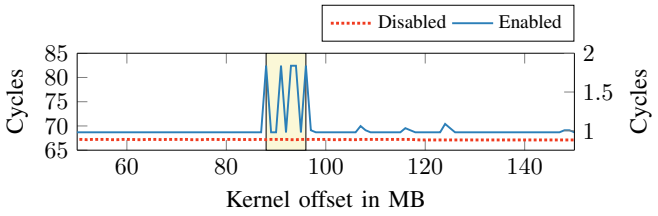

Fig. 5: The prefetch-based KASLR break with enabled (left axis) and disabled (right axis) prefetch instructions. The yellow box indicates the kernel's location at start offset 88.

on average, and the disabled `prefetch` instruction takes 0.885 cycles ($\sigma_{\bar{x}} = 0.0090$, $n = 512\,000\,000$) on average. From this experiment, we conclude that the disabled `prefetch` instruction is indeed equivalent to a `nop` instruction [3]. Thus, the disabled prefetch instruction stops loading data into the cache and does not translate the provided virtual address.

We compare the disabling of the prefetch instructions with the recent FLARE [15] KASLR mitigation, which hardens the kernel against microarchitectural KASLR breaks. However, as all other microarchitectural KASLR breaks only apply to Intel CPUs, disabling the prefetch instructions leads to similar security and performance guarantees as FLARE without modifying the kernel or additional memory overhead. Furthermore, a privileged user, such as a system administrator, can directly activate this mitigation over the MSR interface without any additional requirements. The runtime overhead is also not directly visible as most applications do not use prefetch instructions. On our Ubuntu 20.04 installation, less than 1 % of the installed binaries (300 out of 30 842) contain any software prefetch instructions. We evaluate the performance impact of disabling these instructions with the SPEC CPU 2017 benchmark. Table II shows the benchmark results, where the baseline is with the prefetch instructions enabled. The average performance overhead is only 0.04 %, and thus negligible.

C. Intercepting CPUID to Reduce CrossTalk Leakage

In this case study, we present a software-based mitigation to reduce CrossTalk [64] leakage. The CrossTalk attack leaks data from the staging buffer via the line-fill buffer. To get the data from the staging buffer to the line-fill buffer, CrossTalk uses `cpuid`, `rdrand`, `rdseed`, and `rdmsr` as *leaking primitives* leaking confidential data such as random numbers. However, the `rdmsr` instruction is only available in a privileged attacker model. Ragab et al. [64] assume that the used instructions cannot be trapped, but that it would hypothetically also hinder exploitation. We challenge this assumption by using MSRevelio to search for an MSR bit that can trap the `cpuid` instruction. Our evaluation shows that `cpuid` is the most reliable unprivileged leakage primitives even without TSX. Indeed, MSRevelio successfully discovered such a bit, leading to the first pure software mitigation to practically mitigate unprivileged CrossTalk attacks. By trapping `cpuid`, the OS can ensure that no confidential data, such as random numbers, are leaked from the staging buffer.

TABLE III: SPEC CPU 2017 benchmark for the performance overhead when disabling the software prefetch instructions.

Benchmark	SPEC Score		Overhead
	Baseline	No Prefetch	[%]
600.perlbench_s	4.02	4.02	0.00
602.gcc_s	7.32	7.34	−0.27
605.mcf_s	6.57	6.56	+0.15
620.omnetpp_s	2.89	2.88	+0.35
623.xalancbmk_s	4.34	4.33	+0.23
625.x264_s	4.79	4.79	0.00
631.deepsjeng_s	3.19	3.20	−0.31
641.leela_s	3.29	3.28	+0.30
648.exchange2_s	9.10	9.11	−0.11
Average			+0.04

1) Threat Model: For the pure software mitigation, we assume that an attacker can run unprivileged programs on a CPU affected by CrossTalk. The attacking program controls on which logical core the code is executed and can invoke the unprivileged `cpuid` instruction. The system does not deploy the microcode patch against CrossTalk [64], e.g., for performance reasons or because none is available.

2) MSR Discovery: MSRevelio found MSR(`0x140`) on an Intel i5-4570 using the instruction-behavior analysis (cf. Section III-C) that allows trapping the `cpuid` instruction. On this microarchitecture, this MSR is undocumented in the Intel SDM [41]. The only mention of this MSR is on the Xeon Phi, where it is referred to as `MISC_FEATURE_ENABLES`. Still, even on Intel Core, Xeon, and Celeron CPUs, setting bit 0 results in `cpuid` raising a GP fault. This feature was apparently introduced with the Intel Ivy Bridge microarchitecture as it does not exist on our tested Sandy Bridge machine (i5-2520M). On all our tested machines starting with Ivy Bridge (i5-3230M) to Ice Lake (i3-1005G1) and Jasper Lake (N4500), the MSR exists, and setting bit 0 allows trapping `cpuid`. Therefore, this allows us to harden systems against unprivileged CrossTalk attacks. While designing mitigations on top of undocumented features seems unwise, Intel CPUs with the 10^{th} generation already contain silicon fixes [42].

3) CrossTalk-Mitigation Implementation: We implemented a proof of concept to show that this attack is indeed prevented by trapping the `cpuid` instruction. Our PoC consists of a kernel module and a user-space shared library. The kernel module provides a single `ioctl` that is used by the user-space library to set and clear bit 0 of MSR(`0x140`), *i.e.*, to trap `cpuid` or allow it. The user-space library sets up the kernel module and transparently handles the GP faults. It is simply preloaded for binaries using the `LD_PRELOAD` environment variable. Note that this is just for the sake of simplicity for the proof-of-concept implementation. When implemented for a production system, the entire implementation would either be in the kernel, or partly in the kernel and partly in the dynamic linker and loader, such that it is applied to every binary on the system. The preloaded library installs a signal handler for GP faults. This signal handler analyzes the memory location at the faulting instruction pointer. If the address of the

instruction pointer is accessible (can be verified by abusing the `access` syscall [59]), the library checks if the opcode of `cpuid` (`0xA20F`) is found. If not, any potential other signal handler can be called. However, if the cause of the fault is the `cpuid` instruction, the library ensures that no sensitive data can be leaked from the staging buffer. We evaluated two variants: (1) only executing `cpuid` once before the application starts, and returning cached values for all other calls, (2) first overwriting the staging buffer with other information by calling `rdrand` and then executing the `cpuid` instruction. In both cases, `cpuid` does not transfer the targeted sensitive values from the staging buffer to the line-fill buffer. Moreover, caching the output of `cpuid` is not a functional problem, as the output typically does not change during the runtime.

4) Evaluation: We evaluated both variants' security and performance overhead on an Intel Xeon E3-1505M v5 with Ubuntu 20.04 and Linux 5.4.0-90. To verify that our method indeed hinders CrossTalk, we implemented the two PoCs from the paper, leaking the CPU brand string and the last generated `rdrand` random number. Both PoCs leak the targeted data from the staging buffer. We verified that preloading our library without enabling the trap does not negatively impact the PoC. With the `cpuid` trap enabled, we do not observe any leakage anymore. For the leakage mitigation, we do not observe any difference in the methods, *i.e.*, whether we use cached `cpuid` values or overwrite the staging buffer.

To measure the performance overhead of the `cpuid` trap, we used a microbenchmark that simply executes `cpuid` in a loop. In the normal case, *i.e.*, without the `cpuid` trap, the execution takes 182 cycles ($\sigma_{\bar{x}} = 0.6$, $n = 100\,000$). Trapping the `cpuid` instruction, overwriting the staging buffer, and re-executing it takes on average 9932 cycles ($\sigma_{\bar{x}} = 6.1$, $n = 100\,000$). Caching the `cpuid` instruction improves the performance slightly, with an average execution time of 8204 cycles ($\sigma_{\bar{x}} = 5.7$, $n = 100\,000$). As `cpuid` is typically only called at program startup, e.g., in the libc for feature detection [25], this overhead is negligible for the overall system performance. In contrast, the microcode patch for CrossTalk introduces an overhead of factor 12 for the `rdrand` instruction [64], which is usually used more often than the `cpuid` instruction.

5) Leakage Analysis: We validate that `cpuid` has the highest leakage rate of the unprivileged instructions, by extending the Crosstalk PoC to allow evaluation of the leakage rates when using either signal handling, TSX, or TAA as an exception-suppression method. First, we evaluate leaking `rdrand` with the `cpuid` instruction. Second, we exchange `cpuid` with `rdseed` in the attacker and evaluate the leakage again. Our evaluation found that exchanging `rdseed` with `rdrand` and vice versa did not influence the leakage rates. Therefore, we focused on leaking the more commonly used `rdrand` values. For the evaluation, we use an Intel i7-6700k with Ubuntu 20.04 and Linux 5.4.0-40 with disabled mitigations. The victim generates a random number every 187.5 ms and repeats this 100 times, generating overall 800 B of random data. We perform each experiment 10 times and count the correctly-leaked bytes and how often the entire eight-byte random number is successfully leaked.

TABLE IV: SPEC CPU 2017 benchmark performance overhead when disabling the fast-string optimization.

Benchmark	SPEC Score		Overhead
	Baseline	No Fast-Strings	[%]
600.perlbench_s	5.17	5.16	+0.19
602.gcc_s	8.76	8.06	+8.06
605.mcf_s	6.95	6.91	+0.58
620.omnetpp_s	3.61	3.62	−0.28
623.xalancbmk_s	4.50	4.51	−0.37
625.x264_s	4.47	4.46	+0.15
631.deepsjeng_s	3.92	3.67	+6.37
641.leela_s	3.56	3.57	−0.19
648.exchange2_s	9.82	9.82	0.00
Average			+1.61

For `cpuid`, we observe a leakage of 711.3 B ($\sigma_{\bar{x}} = 4.185$ B) with signal handling, 741.3 B ($\sigma_{\bar{x}} = 4.770$ B) with TSX, and 416.3 B ($\sigma_{\bar{x}} = 3.556$ B) with TAA. With `rdseed` as primitive, we observe a leakage of 3.4 B ($\sigma_{\bar{x}} = 0.544$ B) with signal handling, 2.1 B ($\sigma_{\bar{x}} = 0.368$ B) with TSX, and 553.3 B ($\sigma_{\bar{x}} = 18.296$ B) with TAA. Furthermore, when using signal handling, `cpuid` leaks on average 51.8 times the entire random number whereas `rdseed` is unable to leak the entire random number. For the overall byte-wise leakage rate, the leakage rate of `cpuid` is 211.4 times higher than `rdseed`'s when using signal handling. With recent microcode patches disabling TSX and, therefore, mitigating TAA, the `cpuid` trap is a viable option to further harden systems against unprivileged CrossTalk attacks.

D. Disabling Fast-String Support to Reduce Medusa Leakage

In this case study, we show a software-based approach to reduce the leakage of the Medusa attack [61]. Medusa is a Microarchitectural Data Sampling (MDS) attack, leaking data from the line-fill buffer on Intel CPUs. Medusa leaks data from Write Combining (WC) operations or memory operations backed by WC memory. These write-combining instructions use a part of the line-fill buffer to combine writes to the same cache line to reduce requests sent over the memory bus.

The Medusa attack uses implicit WC instructions like non-temporal moves, `rep movs`, `rep stos` instructions, or explicit WC memory to leak data from the WC buffer. However, as WC memory requires a special memory type, an attacker needs privileges to acquire it, which is only realistic when attacking SGX. We focus on implicit WC operations, available to an unprivileged attacker. By reducing the likelihood that sensitive data ends up in the WC buffer, the probability of a successful Medusa attack is also reduced.

1) Threat Model: We assume an unprivileged attacker is exploiting implicit WC instructions for the Medusa attack on an affected CPU. The OS does not mitigate the Medusa attack, e.g., with the adapted `verw` instruction to clear the fill buffer or group scheduling [61]. Therefore, an attacker can be co-located on the same physical core as the victim program, sharing the core's fill buffer with the victim.

2) MSR Discovery: With the BIOS templating approach (cf. Section IV), MSRevelio automatically detected the documented fast-string enable bit (bit 0) inside MSR(`0x1a0`) (IA32_FEATURE_CONTROL). Intel [41] documents this bit as fast-string enable bit, but the internal effects are only sparsely documented [38], [40]. Based on the instruction-behavior analysis of MSRevelio, we see that clearing the fast-string-enable bit changes the associated instructions to no longer perform WC memory writes. Therefore, this bit is suitable to reduce leakage of the unprivileged Medusa attacks.

3) Evaluation: We evaluate the impact of the fast-string enable bit on the Medusa attack on an Intel Core i7-6700K CPU that is affected by Medusa with Ubuntu 20.04 LTS and Linux 5.4.0-40. For the evaluation, we rely on the public Medusa PoCs [61]. Specifically, we focus on the PoC variants using fast-string operations that an unprivileged attacker can use [61]. We run the victim using fast-string operations with sensible data and test against these attack variants. We fix the test system's frequency to $3\,\mathrm{GHz}$ and pin the attacker and victim applications to sibling hyperthreads. We first verify that the PoC successfully leaks the targeted data. When disabling fast-string operations using MSR(`0x1a0`), the leakage is entirely gone, successfully preventing these variants of Medusa. We evaluate the performance overhead when disabling fast-string operations with the SPEC CPU 2017 benchmark and observe an average performance overhead of $1.61\,\%$. Table IV shows the benchmark results, where the baseline is with the optimized fast strings operations enabled.

4) Discussion: Similar to the CrossTalk mitigation (cf. Section V-C), this is only a short-term solution for affected CPUs. Newer CPUs, e.g., 10^{th} generation, are not affected by Medusa anymore, hence this mitigation is only necessary on older CPUs for which this MSR bit successfully reduces the leakage. Additionally, CPUs received security updates by repurposing the `verw` instruction to clear microarchitectural buffers on context switches [61]. While `verw` does not prevent attacks from a hyperthread, disabling the write combining instruction mitigates these attacks. Finally, as the leakable data for an attacker depends on the instructions executed within a victim application, the only remaining sources for leakage are non-temporal moves, as well as the upper $128\,\mathrm{bit}$ of AVX stores, which can be disabled via the `XCR0` register.

E. Tracing Microcode-introduced MSRs

In this case study, we show that MSRevelio can trace the evolution of MSRs of a CPU over multiple microcode versions. Tracing the addition of MSRs allows determining the microcode version where vendors deployed patches relying on additional MSRs. Moreover, we can analyze the time between deployment and documentation of MSRs. We show that all MSRs retrofitted using microcode on our tested machines are security-related. Thus, detecting MSRs before they are documented hints that there is a CPU vulnerability currently under embargo, as the fixes should, in the best case, already be deployed when the embargo ends. If a security patch introduces undocumented MSRs, we assume that the undocumented MSR exposes additional configuration bits for the mitigation. With this information, an adversary can determine the effects corresponding to the added MSR and potentially infer the reason for the security patch. This approach is similar to *patch diffing* where an attacker extracts the vulnerability by inspecting the patches provided to a system or component.

Since the discovery of Spectre [49] and Meltdown [54], many security patches have included new MSRs to mitigate vulnerabilities. To help OSs and hypervisors mitigate the impact of Spectre, a microcode update added MSR(`0x49`) (IA32_PRED_CMD) and MSR(`0x48`) (IA32_SPEC_CTR). These MSRs allow configuring the branch predictor and flushing it's state [37]. Similarly, due to Foreshadow [72], the MSR(`0x10B`) (IA32_FLUSH_CMD) was introduced to flush the L1 cache [35]. Recently, Intel also introduced MSR(`0x123`) (IA32_MCU_OPT_CTRL) with a microcode update to change the behavior of `rdseed` and `rdrand`, mitigating the CrossTalk attack [64].

1) Threat Model: We assume a sophisticated attacker is tracking the evolution of MSRs over multiple released microcode versions for different CPU generations to find new undocumented MSRs. The attacker uses the classification approaches shown in Section III-B and Section III-C to determine the effects of the MSR and potentially the vulnerability's source. This leaves the attacker with additional time until the public disclosure to mount attacks on unpatched systems.

2) Implementation: To trace the evolution of the MSRs over the microcode version, we use the late-loading mechanism of Linux [83]. The late-loading mechanism updates the CPU microcode to a newer version without rebooting the system. As a source for the microcodes, we rely on two GitHub repositories. First, the official Intel microcode repository [45], containing the microcode versions back to March 2019. Second, a collection of microcode versions from Plato Mavropoulos [57] dating back to 1996. A signature from Intel ensures the integrity of all microcode files. For each microcode version available for our CPU, MSRevelio extracts the list of MSRs available after applying the microcode update. As microcode can only be replaced by microcode with a newer version, we start at the initial microcode version hardcoded in the BIOS, and gradually update to newer versions. As a test system, we chose a CPU with the Sandy Bridge microarchitecture, which is the oldest second-generation Intel Core microarchitecture, released in 2011. Additionally, we use a CPU with the Broadwell microarchitecture (released 2014) and a CPU with the Coffee Lake microarchitecture (released 2017). For all published transient-execution attacks [42], [16], at least one of these microarchitectures is affected. With 28 microcode versions, ranging from 2011 to 2021, we found a large number of different microcodes to test.

3) Results: As expected, MSRevelio detects the introduction of MSR(`0x48`), MSR(`0x49`), MSR(`0x10B`) and MSR(`0x123`). Interestingly, there is a significant time difference between the first occurrence of these MSRs and their documentation. For the Sandy Bridge machine, MSR(`0x48`) and MSR(`0x49`) were introduced with microcode 0x2d on

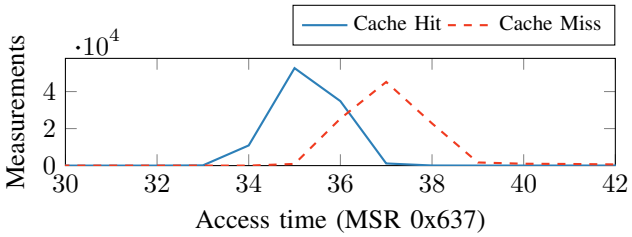

Fig. 6: Cache hits and misses measured with MSR(0x637).

February 7th, 2018, more than a month after the disclosure of Spectre [49]. In contrast, on Broadwell, these MSRs were introduced with microcode version 0x28 already on November 17th, 2017, *i.e.*, nearly 7 weeks *before* the public disclosure. On the same machine, MSR(0x10B) was introduced with microcode 0x2b on March 22nd, 2018 nearly 5 months *before* the public disclosure of Foreshadow on August 14th, 2018. The Sandy Bridge machine received this MSR with microcode 0x2e on May 10^{th}, 2018, 3 months *before* the disclosure. MSR(0x123) was introduced on the Broadwell with microcode 0x2f from November 12th, 2019, while the public disclosure was on June 9th, 2020. Skylake-S and Coffee Lake received the MSR with the microcode update from January 9th, 2020. We use MSRevelio to analyze the CrossTalk mitigation's MSR(0x123) and observe differences for `rdrand` and `rdseed`, directly revealing the affected instructions. These results show that microcode updates containing new MSRs are distributed before the MSR is officially documented. None of our tested microcode updates introduced any non-security-related MSRs. We assume that new non-security-related MSRs are only introduced with new microarchitectures. Hence, by using MSRevelio, it is possible to reveal the existence of CPU vulnerabilities or errata before they are publicly documented.

F. Exploiting Xen

In this section, we show that MSRevelio is also applicable in cloud environments to enumerate MSRs accessible to guest virtual machines. This does not only expose access to MSRs that can constitute a security threat but also allows fingerprinting the hypervisor. The Xen hypervisor checks the availability of an MSR inside the `guest_rdmsr` and `guest_wrmsr` functions and decides if access should be *simulated*, *allowed*, or trigger a *GP-fault* [82]. Until recently, the Xen hypervisor used a blocklist prohibiting guests from accessing MSRs. Due to the nature of a blocklist approach, the hypervisor does not restrict access to undocumented MSRs and, thus, allows guests to read and write them. With version 4.15 (April 2021), Xen changed the default behavior from a blocklist to an allowlist [17], [62], preventing access to undocumented MSRs.

1) Threat Model: We assume a privileged attacker running in a Xen VM re-enacting a scenario of a low-priced cloud provider offering single-core virtual machines. We assume that the attacker's virtual machine is pinned to a logical core of the machine sharing the physical core with other guests. Furthermore, we assume that the Xen hypervisor disables access to MSRs enabling power side-channel attacks [53] and traps calls to the timestamp counter to reduce its resolution [76], [55]. As the victim machine only runs on a single core, alternative timing approaches like counting threads [52] are unavailable.

TABLE V: Accessible MSRs in different Xen versions.

Xen Version	# Accessible	# Static	# Dynamic
4.7 (IBM cloud)	618	600	18
4.11.4 (Ubuntu)	521	496	25
4.11.4 (Debian)	505	486	19
4.14.1	452	434	18
4.15	203	202	1

2) Alternative Timer in Xen: With MSRevelio, we discovered the undocumented MSR(0x637) on Intel CPUs that continuously increments its value and correlates with the timestamp counter (cf. Figure 2). While not officially documented in Intel's SDM [41], we found a reference to MSR(0x637) in the coreboot project [18] as MSR_COUNTER_24_MHZ. We exploit this timer to mount a Foreshadow attack [36], [81] from a Xen virtual machine. We evaluate the attack on an Intel Core i7-8650U running Ubuntu 20.04.2 LTS with Linux 5.4.0-52 and Xen 4.14.2. The attacker runs in a PVH virtual machine with Debian 10 and Linux 4.19.0-16 with a single virtual CPU assigned to a logical core.

In our experiment, the attacker uses the MSR(0x637) as a timestamp to distinguish cache hits from misses. In contrast to the cycle-accurate timestamp accessible via `rdtsc`, this timestamp has a lower resolution of only $41.67\,\mathrm{ns}$. Thus, when measuring the access time to cached and uncached memory using the MSR, there is a slight overlap, as shown in Figure 6.

In our attack, the attacker sets the PFN of one of its EPT pages to a physical location of a victim page and marks the page as non-present, triggering an L1 Terminal Fault [36] on access. As the page is not present, the CPU aborts the page translation and uses the attacker-controlled PFN to lookup data in the L1 cache. We use Intel TSX to suppress the fault and encode the leaked values by caching corresponding addresses in a look-up table. Using Flush+Reload with the MSR as a timestamp, we successfully recover the leaked values.

In our attack, we leak a 50-byte string from a victim running on the sibling hyperthread. Our unoptimized proof-of-concept implementation has an average runtime of $107.4\,\mathrm{ms}$ ($\sigma_{\bar{x}} = 1000$, $n = 1.756$) and an average leakage rate of 214 B/s ($\sigma_{\bar{x}} = 1000$, $n = 4.176$). When using `rdtsc` as the timestamp, we achieve an average runtime of $0.38\,\mathrm{ms}$ ($\sigma_{\bar{x}} = 1000$, $n = 0.003$) and an average leakage rate of 49 147 B/s ($\sigma_{\bar{x}} = 1000$, $n = 498.66$). The difference is mainly caused by the measurement imprecision caused by the lower resolution and also because querying the MSR value has a heavy performance impact [1].

3) Fingerprinting Xen Versions: In our analysis of various Xen versions, we detected that the number of visible MSRs is different for each Xen version, allowing an attacker to infer the Xen version if this information is (partly) blocked, as e.g., on the IBM cloud. We evaluated MSRevelio within different Xen

versions on an Intel i7-8650U CPU running Ubuntu 20.04.2 LTS with Linux 5.40-52 and on the IBM cloud.

For our evaluation, we first fingerprint the hypervisor by using the MSR detection mechanism of MSRevelio, including the analysis for static and dynamic MSRs. We show the results of the detected MSRs for 5 different XEN versions in Table V. It is noticeable that with an increasing version number, the number of exposed MSRs decreases as the blocklist is extended with additional entries. The implemented allowlist approach of version 4.15 significantly reduces the number of MSRs from 452 (version 4.14.1) to 203.

It is worth highlighting that while reporting the same Xen version (4.11.4), the number of accessible MSRs between the latest available version on Ubuntu and Debian differs by 16 MSRs. By comparing the detected MSRs, we observed that the MSRs exploited by the Platypus attack [53] are still accessible on the Ubuntu installation as the security patches of Xen have not been applied to Ubuntu. Thus, despite reporting the same version number, virtual machines can be exposed to different security risks depending on the patch level.

VI. Discussion and Limitations

A. Related Work

One of the first MSR scanners dates back to 2001 [47] using the file-based Linux MSR interface to search for readable MSRs. In contrast to MSRevelio, this scanner can only detect readable MSRs. Furthermore, recent changes to the Linux kernel restrict accesses to MSRs from userspace [63], making this approach less reliable.

Domas [24], [23] focused on the execution time of reading MSRs to identify ones with unique functionality to detect a potential CPU backdoor. With MSRevelio, we do not focus on unique MSRs changing the ISA, *i.e.*, introducing new instructions or changing the architectural functionality of instructions.

Haruspex [12] scans the x86 instruction set with speculative execution and performance counters. Bölük also applied this approach to detect undocumented MSRs [13]. While the performance of this approach is unclear, the stated detected MSRs match our findings, e.g., for MSR(`0x2e6`) mentioned as `LT_LOCK_MEMORY_MSR` in an Intel errata [43]. In addition, MSRevelio found that one can only write 0 to this MSR.

B. Indirect Effects

There are also MSR bits that affect the system without directly affecting instructions. Examples are the configuration of hardware prefetchers or fixes for CPU errata. These bits only have a measurable effect in corner cases that cannot be triggered automatically. To detect the effect of such bits, targeted test cases would be required. If such a targeted test case exists, e.g., because someone has the intuition that an undocumented MSR affects a specific feature (e.g., disabling hardware prefetchers), MSRevelio can also be extended to analyze MSR bits concerning this particular test.

VII. Conclusion

With MSRevelio, we automatically detect undocumented MSRs and their effects on Intel and AMD CPUs. We demonstrate that undocumented MSRs can not only hint at the existence of CPU vulnerabilities but that they can also introduce new attack vectors or re-enable mitigated attacks. Furthermore, we show that undocumented MSRs can also be used to mitigate various microarchitectural attacks.

In this paper, we show that undocumented or sparsely documented MSRs have a non-negligible effect on system security, not only on native systems but also in the cloud.

Acknowledgments

We thank the anonymous reviewers for their guidance, comments, and suggestions. Additional funding was provided by a generous gift from Amazon. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties. Daniel Weber thanks the Saarbrücken Graduate School of Computer Science. Andreas Kogler thanks Claudio Canella, Lukas Giner, and Martin Schwarzl for the discussions and feedback.

References

[1] A. Abel and J. Reineke, "uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures," in *ASPLOS*, 2019.

[2] ——, "nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems," in *ISPASS*, 2020.

[3] Advanced Micro Devices Inc., *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors*, 2013.

[4] *AMD64 Architecture Programmer's Manual*, Advanced Micro Devices Inc., 2017.

[5] *Open-Source Register Reference For AMD Family 17h Processors Models 00h-2Fh*, 3rd ed., Advanced Micro Devices Inc., 7 2018.

[6] Amazon, "Amazon EC2 On-Demand Pricing," 2021. [Online]. Available: https://aws.amazon.com/ec2/pricing/on-demand/

[7] AMI, "BIOS/UEFI Utilities," 2021. [Online]. Available: https://www.ami.com/products/firmware-tools-and-utilities/bios-uefi-utilities/

[8] ARM, "mbed TLS," 2020. [Online]. Available: https:///tls.mbed.org

[9] J.-P. Aumasson and L. Merino, "SGX Secure Enclaves in Practice: Security and Crypto Review," in *Black Hat Briefings*, 2016.

[10] D. J. Berndt and J. Clifford, "Using Dynamic Time Warping to Find Patterns in Time Series," in *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, 1994.

[11] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software Grand Exposure: SGX Cache Attacks Are Practical," in *WOOT*, 2017.

[12] C. Bölük, "Haruspex," 2021. [Online]. Available: https://github.com/can1357/haruspex/

[13] ——, "Undocumented MSRs with Haruspex," 2021. [Online]. Available: https://twitter.com/_can1357/status/1427511999550959628

[14] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking Data on Meltdown-resistant CPUs," in *CCS*, 2019.

[15] C. Canella, M. Schwarz, M. Haubenwallner, M. Schwarzl, and D. Gruss, "KASLR: Break It, Fix It, Repeat," in *AsiaCCS*, 2020.

[16] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," in *USENIX Security Symposium*, 2019, extended classification tree and PoCs at https://transient.fail/.

[17] A. Cooper, "x86/hvm: disallow access to unknown MSRs," 2020. [Online]. Available: https://xenbits.xen.org/gitweb/?p=xen.git;a=commitdiff;h=84e848fd7a162f669cf8248ce502ca864f869447

[18] coreboot, "coreboot: Fast, secure and flexible OpenSource firmware," 2019. [Online]. Available: https://www.coreboot.org/

[19] CoreBoot, "CoreBoot - Bios Update Trigger," 2021. [Online]. Available: https://github.com/coreboot/coreboot/blob/master/src/soc/intel/common/block/include/intelblocks/msr.h#L17

[20] V. Costan and S. Devadas, "Intel SGX Explained," *Cryptology ePrint Archive, Report 2016/086*, 2016.

[21] Czernobyl, "Super-secret debug capabilities of AMD processors !" 2014. [Online]. Available: http://www.woodmann.com/collaborative/knowledge/index.php/Super-secret_debug_capabilities_of_AMD_processors_!

[22] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[23] C. Domas, "Breaking the x86 ISA, v. 2017-07-27," *Black Hat US*, 2017.

[24] ——, "Hardware Backdoors in x86 CPUs," *Black Hat US*, 2018.

[25] F. S. Foundation, "GLIBC Feature Detection," 2021. [Online]. Available: https://github.com/bminor/glibc/blob/master/sysdeps/x86/cpu-features.c

[26] C. Gough, I. Steiner, and W. Saunders, *Energy Efficient Servers*. Apress, 2015.

[27] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR," in *CCS*, 2016.

[28] G. G. Henry and T. Parks, "Apparatus and method for limiting access to model specific registers in a microprocessor," December 2012, uS Patent 8,341,419 B2.

[29] J. Horn, "speculative execution, variant 4: speculative store bypass," 2018.

[30] W.-M. Hu, "Reducing Timing Channels with Fuzzy Time," *Journal of Computer Security*, 1992.

[31] Intel, "Advanced Encryption Standard (AES) Instructions Set: White Paper," 2008.

[32] Intel, "Pin - A Dynamic Binary Instrumentation Tool," 2012. [Online]. Available: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

[33] ——, "Intel Software Guard Extensions SDK for Linux OS Developer Reference," May 2016, rev 1.5.

[34] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture," 2016.

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

[36] Intel, "L1 Terminal Fault SA-00161," 2018. [Online]. Available: https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault

[37] ——, "Deep Dive: CPUID Enumeration and Architectural MSRs," May 2019. [Online]. Available: https://software.intel.com/security-software-guidance/insights/deep-dive-cpuid-enumeration-and-architectural-msrs#MDS-CPUID

[38] Intel, "Intel 64 and IA-32 Architectures Optimization Reference Manual," 2019.

[39] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z," 2019.

[40] ——, "Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide," 2019.

[41] ——, "Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers," May 2019.

[42] ——, "Affected Processors: Transient Execution Attacks," 2020. [Online]. Available: https://software.intel.com/security-software-guidance/processors-affected-transient-execution-attack-mitigation-product-cpu-model

[43] Intel, "Intel Xeon Processor Scalable Family," 2020.

[44] Intel, "Refined Speculative Execution Terminology," 2020. [Online]. Available: https://software.intel.com/security-software-guidance/insights/refined-speculative-execution-terminology

[45] Intel, "Intel Linux Processor Microcode Data Files," 2021. [Online]. Available: https://github.com/intel/Intel-Linux-Processor-Microcode-Data-Files

[46] A. James, "ghidra-firmware-utils," 2021. [Online]. Available: https://github.com/al3xtjames/ghidra-firmware-utils

[47] D. Jone, "Dave Jone's MSR scanner," 2001. [Online]. Available: https://www.mail-archive.com/linuxbios@listman.lanl.gov/msg02813.html

[48] P. Kemkes, "Techniques: Current use of virtual machine detection methods," 2020. [Online]. Available: https://www.gdatasoftware.com/blog/2020/05/36068-current-use-of-virtual-machine-detection-methods

[49] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *S&P*, 2019.

[50] P. Koppe, B. Kollenda, M. Fyrbiak, C. Kison, R. Gawlik, C. Paar, and T. Holz, "Reverse engineering x86 processor microcode," in *USENIX Security Symposium*, 2017.

[51] M. Lipp, D. Gruss, and M. Schwarz, "Amd prefetch attacks through power and time," in *USENIX Security Symposium*, 2022.

[52] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache Attacks on Mobile Devices," in *USENIX Security Symposium*, 2016.

[53] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, "PLATYPUS: Software-based Power Side-Channel Attacks on x86," in *S&P*, 2021.

[54] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *USENIX Security Symposium*, 2018.

[55] W. Liu, D. Gao, and M. K. Reiter, "On-demand time blurring to support side-channel defense," in *ESORICS*, 2017.

[56] G. Maisuradze and C. Rossow, "ret2spec: Speculative Execution Using Return Stack Buffers," in *CCS*, 2018.

[57] P. Mavropoulos, "CPUMicrocodes," 2021. [Online]. Available: https://github.com/platomav/CPUMicrocodes

[58] J. Mechalas, "Trusted CPU Feature Detection Library," 2019. [Online]. Available: https://github.com/intel/sgx-cpu-feature-detection

[59] M. Miller, "Safely searching process virtual address space," 2004.

[60] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "CacheZoom: How SGX amplifies the power of cache attacks," in *CHES*, 2017.

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

[62] R. Pau, "x86/pv: disallow access to unknown MSRs," 2020. [Online]. Available: https://xenbits.xen.org/gitweb/?p=xen.git;a=commitdiff;h=322ec7c89f6640ee2a99d1040b6f786cf04872cf

[63] B. Petkov, "[RFC PATCH] x86/msr: Filter MSR writes," 2020. [Online]. Available: https://lkml.org/lkml/2020/6/12/273

[64] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "CrossTalk: Speculative Data Leaks Across Cores Are Real," in *S&P*, 2021.

[65] D. E. Rumelhart, J. L. McClelland, and C. PDP Research Group, Eds., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. MIT Press, 1986.

[66] J. Sakkinen, "Add SGX Launch Control MSR definitions," 2018. [Online]. Available: https://patchwork.kernel.org/project/intel-sgx/patch/20181106134758.10572-14-jarkko.sakkinen@linux.intel.com/

[67] N. Schlej, "UEFI firmware image viewer and editor," 2020. [Online]. Available: https://github.com/LongSoft/UEFITool

[68] M. Schwarz, D. Gruss, M. Lipp, C. Maurice, T. Schuster, A. Fogh, and S. Mangard, "Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features," in *AsiaCCS*, 2018.

[69] M. Schwarz, D. Gruss, S. Weiser, C. Maurice, and S. Mangard, "Malware Guard Extension: Using SGX to Conceal Cache Attacks," in *DIMVA*, 2017.

[70] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-Privilege-Boundary Data Sampling," in *CCS*, 2019.

[71] S. Takarabt, A. Schaub, A. Facon, S. Guilley, L. Sauvage, Y. Souissi, and Y. Mathieu, "Cache-timing attacks still threaten iot devices," in *Codes, Cryptology and Information Security*, 2019.

[72] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *USENIX Security Symposium*, 2018.

[73] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *S&P*, 2020.

[74] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control," in *Workshop on System Software for Trusted Execution*, 2017.

[75] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue In-flight Data Load," in *S&P*, 2019.
[76] B. C. Vattikonda, S. Das, and H. Shacham, "Eliminating fine grained timers in Xen," in *CCSW*, 2011.
[77] V. Viswanathan, "Disclosure of hardware prefetcher control on some intel processors." [Online]. Available: https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors
[78] D. Vlasenko, "Better document AMD "tweak MSRs"," 2017. [Online]. Available: https://lore.kernel.org/patchwork/patch/783107/
[79] J. Voisin, "Spectre exploits in the "wild"," 2021. [Online]. Available: https://dustri.org/b/spectre-exploits-in-the-wild.html
[80] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza, "AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves," in *ESORICS*, 2016.
[81] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, "Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution," 2018. [Online]. Available: https://foreshadowattack.eu/foreshadow-NG.pdf
[82] XEN, "XEN's MSR handling," 2021. [Online]. Available: https://github.com/xen-project/xen/blob/RELEASE-4.15.0/xen/arch/x86/msr.c
[83] F. Yu and B. Petkov, "The Linux Microcode Loader," 2019. [Online]. Available: https://www.kernel.org/doc/html/latest/x86/microcode.html

APPENDIX A
NUMBER OF TESTS

Equation (1) shows the number of tests N_T with MSRevelio. For each writable MSR, we first perform 16 tests in *phase 2* to find an MSR with side-effects. To determine the exact position of the enum within the MSR, we perform the optimized sliding window search, where we re-use the previous results. For the case with $W = 4$, we require 496 tests per observed effect for *phase 3*.

$$N_T = \underbrace{\left(\sum_{writeable} 2^W\right)}_{\text{phase 2}} + \underbrace{\left(\sum_{observed} \left(2^W + (64-W)2^{W-1}\right)\right)}_{\text{phase 3}} \tag{1}$$

In comparison, Equation (2) shows the number of tests N_T required for an exhaustive search.

$$N_T = \sum_{msr \in writeable} 2^{writableBits(msr)} \tag{2}$$

APPENDIX B
INSTRUCTION GROUPS AND PMC EVENTS

Table VI shows the instructions groups used during the instruction behaviour analysis (cf. Section III-C). The full configuration groups including the detailed instructions are available in MSRevelio's GitHub repository[2]. For the performance counters we use the default configurations of *nanoBench* [2] for Intel[3] and AMD[4] CPUs. These configuration contain all performance counters for the given microarchitecture. However, most of the counters are also available on similar Intel or AMD microarchitectures, and we therefore relied on the Skylake configuration for all tested Intel CPUs (cf. Table I).

[2]MSRevelio's repository: https://github.com/IAIK/msrevelio

[3]Intel PMC config: https://github.com/andreas-abel/nanoBench/tree/fc038541dfd0de3428c7521131a548a85e923f7c/configs/cfg_Skylake_all.txt

[4]AMD PMC config: https://github.com/andreas-abel/nanoBench/tree/fc038541dfd0de3428c7521131a548a85e923f7c/configs/cfg_Zen_all.txt

TABLE VI: The used *instruction groups* for the instruction behavior analysis of the MSR bits (cf. Section III-C).

Group Name	# Instructions	Group Description
AES	6	AES-NI instructions
CPUID	1	`cpuid` instruction
FENCES	3	`sfecne`, `mfence`, and `lfence` instructions
FLUSH	1	`clflush` instruction
FP_ARITH	9	x87 floating-point instructions
FP_VARITH	7	AVX2 vector floating-point instructions
INT_ARITH	12	x86 integer instructions
INT_VARITH	11	AVX2 vector integer instructions
LOAD	1	AVX2 vector load instruction
STORE	1	AVX2 vector store instruction
MOVES	35	various memory `mov` instructions
MISC	11	`xchg`, `bswap` and string instructions
PREFETCH	6	`prefetch` instructions
RANDOM	1	`rdrand` instruction
STRIDED	17	*strided* memory loads
TIME	2	`rdtsc`, `rdtscp` instructions
Total	124	

TABLE VII: *Flipping masks* for enum fields with length of up to 4 bits. Selecting 4 consecutive bits within these 16 masks always covers all possible enum bit combinations by mimicking the `rotate` operation (cf. Section III-C).

Flipping Mask	Enum[3:0]	Enum[4:1]
0x0000000000000000	0b...0000	0b..0000.
0x1111111111111111	0b...0001	0b..1000.
0x2222222222222222	0b...0010	0b..0001.
0x3333333333333333	0b...0011	0b..1001.
0x4444444444444444	0b...0100	0b..0010.
⋮	⋮	⋮
0xffffffffffffffff	0b...1111	0b..1111.

APPENDIX C
FLIPPING MASKS

Table VII shows the *flipping masks* for enum fields with a length of up to 4 bits covering all possible 4 bit combinations, regardless of the position of the enum field within the 64 bit long MSR. When shifting the enum field inside the MSR, the *flipping mask* mimics the `rotate` operation.

APPENDIX D
BIOS PATCH

Figure 7 is the patch in the UEFIPatch[5] format. We tested it with two different BIOS images, an AMI Aptio V 2.21.1277 with build date 2020, and an AMI Aptio V 2.18.1263 with build date 2021. The motherboard of the all-in-one PC is the BESSTAR TECH IB9. As we chose the pattern-based patch format, it should also be applicable to other AMI BIOS versions. The difference in the two patches is that in 2.18.1263, there is a call to a function wrapping `wrmsr`, whereas in 2.21.1277, the `wrmsr` instruction is inlined.

```
# AMI Aptio V BIOS/UEFI 2.21.1277 (Core Version 1.010)
299D6F8B-2EC9-4E40-9EC6-DDAA7EBF5FD9 10 P:83C801EB0383C80389442410:83C800EB0383C80289442410
299D6F8B-2EC9-4E40-9EC6-DDAA7EBF5FD9 12 P:83C801EB0383C80389442410:83C800EB0383C80289442410

# AMI Aptio V BIOS/UEFI 2.18.1263 (Core Version 5.12)
299D6F8B-2EC9-4E40-9EC6-DDAA7EBF5FD9 10 P:83C801EB0383C8035250683C010000:83C800EB0383C8025250683C010000
299D6F8B-2EC9-4E40-9EC6-DDAA7EBF5FD9 12 P:83C801EB0383C8035250683C010000:83C800EB0383C8025250683C010000
```

Fig. 7: The patch for AMI Aptio V BIOS to disable the AES-NI lock bit. The patch can be applied to a BIOS image using UEFIPatch.

```
#if defined(MBEDTLS_AESNI_C) && defined(MBEDTLS_HAVE_X86_64)
    if( mbedtls_aesni_has_support( MBEDTLS_AESNI_AES ) )
        return( mbedtls_aesni_setkey_enc( (unsigned char *) ctx->rk, key, keybits ) );
#endif

    for( i = 0; i < ( keybits >> 5 ); i++ )
    {
        GET_UINT32_LE( RK[i], key, i << 2 );
    }

    switch( ctx->nr )
    {
        case 10:

            for( i = 0; i < 10; i++, RK += 4 )
            {
                RK[4]  = RK[0] ^ RCON[i] ^
                ( (uint32_t) FSb[ ( RK[3] >>  8 ) & 0xFF ]       ) ^
                ( (uint32_t) FSb[ ( RK[3] >> 16 ) & 0xFF ] <<  8 ) ^
                ( (uint32_t) FSb[ ( RK[3] >> 24 ) & 0xFF ] << 16 ) ^
                ( (uint32_t) FSb[ ( RK[3]       ) & 0xFF ] << 24 );

                RK[5]  = RK[1] ^ RK[4];
                RK[6]  = RK[2] ^ RK[5];
                RK[7]  = RK[3] ^ RK[6];
            }
            break;
        /* additional cases for different key lengths */
    }
```

Listing 1: An excerpt of the relevant code from the mbedTLS library that is susceptible to cache attacks. If AES-NI is not available (Line 2), mbedTLS falls back to a T-table implementation with secret-dependent key lookups in the array `FSb`.

APPENDIX E
MBEDTLS LEAK

Listing 1 shows an excerpt of the relevant code from the mbedTLS[6] library where the implementation falls back to the T-table implementation if AES-NI is not available.

[5]UEFIPatch tool: https://github.com/LongSoft/UEFITool/tree/master/UEFIPatch

[6]MbedTLS's source code: https://github.com/ARMmbed/mbedtls/blob/dd57b2f240c597e4cf6cc2492d5c03d067f234f9/library/aes.c#L587