FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 4 · Verteilte Systeme und Betriebssysteme

Henriette Paulina Hofmeier

Dynamic Reconfiguration of Hardware-Vulnerability Mitigations in the Linux Kernel

Masterarbeit im Fach Informatik

24. Juni 2022

Please cite as:
Henriette Paulina Hofmeier, “Dynamic Reconfiguration of Hardware-Vulnerability Mitigations in the Linux Kernel”, Master’s Thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Dept. of Computer Science, June 2022.

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Verteilte Systeme und Betriebssysteme
Martensstr. 1 · 91058 Erlangen · Germany

Dynamic Reconfiguration of Hardware-Vulnerability Mitigations in the Linux Kernel

Masterarbeit im Fach Informatik

vorgelegt von

Henriette Paulina Hofmeier

geb. am 12. Dezember 1992
in Bayreuth

angefertigt am

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**Prof. Dr.-Ing. habil. Wolfgang Preikschat-Schröder** **Stefan Reif, M.Sc.** **Luis Gerhorst, M.Sc.** **Christian Eichler, M.Sc.** **Prof. Dr.-Ing. Timo Hönig**
Betreuender Hochschullehrer:	**Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat**
Beginn der Arbeit:	**31. Dezember 2021**
Abgabe der Arbeit:	**24. Juni 2022**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Henriette Paulina Hofmeier)
Erlangen, 24. Juni 2022

ABSTRACT

Providing secure systems, for example, in computing centers, is an essential task of service providers. Vulnerabilities threatening secure execution are not only located in defective software but can also be found in the hardware itself. Hardware vulnerabilities, such as Spectre, Meltdown, and Microarchitectural Data Sampling, pose significant security concerns and can leave systems vulnerable to attacks extracting privileged information. As these vulnerabilities are often unfixable for already deployed hardware, software developers in general and operating system developers, in particular, go to great lengths to mitigate these attacks. These mitigations typically come with significant performance overheads, especially if speculative execution has to be restricted. Due to differences in the data they handle and security concerns, in general, processes may require varying degrees of protection. Thus, mitigations may only be required for short time spans or individual processes – if at all. The mainline Linux kernel does not offer run-time control for all mitigations. For some exploits, multiple mitigations may be available that differ in the underlying protection mechanism and in their overhead on performance and efficiency.

This thesis presents dynamic reconfiguration of hardware-vulnerability mitigations in the Linux kernel. By adapting the mitigation configurations to the current workload, the system's performance and energy efficiency can be optimized as the overhead of unnecessarily enabled mitigations is removed. Also, if multiple protection mechanisms are available for a specific vulnerability, a reconfiguration service determines the optimal configuration depending on workload characteristics, hardware, and system state. Dynamic control of mitigations that are only configurable at boot time in the mainline kernel is provided by a kernel module and kernel extensions. By utilizing code patching at run time, mitigations are omitted from the control flow if the respective mitigation is disabled. Combined, the service, kernel module, and kernel extensions provide the dynamic reconfiguration of hardware-vulnerability mitigations. The evaluation shows that using dynamic reconfigurations and adapting mitigation configurations to the system state has the potential to improve the system's efficiency significantly. The evaluation also shows that the design and implementation of dynamic reconfiguration of hardware-vulnerability mitigations, as presented in this thesis, can be integrated into the Linux kernel with only minimal run-time overhead. Thus, this thesis provides the means for future research into the optimization of hardware-vulnerability mitigation reconfigurations.

KURZFASSUNG

Die Bereitstellung sicherer Systeme, beispielsweise in Rechenzentren, ist eine wesentliche Aufgabe von Systembetreibern. Schwachstellen, die die sichere Ausführung gefährden, liegen nicht nur in fehlerhafter Software, sondern auch in der Hardware selbst. Hardware-Schwachstellen wie Spectre, Meltdown und Mikroarchitektonisches Daten-Abtasten (engl. Microarchitectural Data Sampling, MDS) stellen erhebliche Sicherheitslücken dar und können Systeme anfällig für Angriffe machen, die geschützte Informationen extrahieren. Daher ist die Schadensminderung solcher Schwachstellen unerlässlich, um sichere Systeme bereitzustellen. Diese Minderungen sind in der Regel mit signifikanten Leistungseinbußen verbunden, insbesondere wenn die spekulative Ausführung eingeschränkt wird. Für einige Schwachstellen stehen mehrere Gegenmaßnahmen zur Verfügung, die sich im zugrunde liegenden Schutzmechanismus und in den damit verbundenen Leistungs- und Effizienzeinbußen unterscheiden. Aufgrund von Unterschieden hinsichtlich der verarbeiteten Daten und der generellen Sicherheitsanforderungen, können Prozesse verschiedene Schutzgrade benötigen. Daher können Schutzmechanismen auch nur für kurze Zeit oder einzelne Prozesse erforderlich sein – wenn sie überhaupt benötigt werden. Der Linux Mainline-Kernel bietet nicht für alle Mechanismen Laufzeitsteuerung an. Dies kann zu vermeidbaren Leistungseinbußen führen, wenn Schutzmechanismen auch dann aktiv sind, wenn sie bei der Ausführung bestimmter Prozesse nicht erforderlich sind.

Diese Abschlussarbeit präsentiert die dynamische Rekonfiguration von Hardware-Schwachstellenminderungen im Linux-Kernel. Dies ermöglicht das Anpassen der Konfigurationen der Schutzmechanismen an die Bedürfnisse der laufenden Prozesse zur Laufzeit. Dadurch können Schutzmechanismen deaktiviert werden, sobald sie nicht mehr benötigt werden. Somit kann die Leistung und Energieeffizienz des Systems optimiert werden, da der Overhead unnötig aktivierter Minderungen entfernt wird. Wenn mehrere Schutzmechanismen für eine Schwachstelle verfügbar sind, bestimmt der Rekonfigurationsdienst außerdem die optimale Konfiguration in Abhängigkeit von Workload-Eigenschaften, Hardware und Systemzustand. Die dynamische Steuerung von Schwachstellenminderungen, die im Mainline-Kernel nur beim Hochfahren des Systems konfigurierbar sind, wird durch ein Kernelmodul und Kernelerweiterungen bereitgestellt. Durch die Verwendung von Laufzeitaktualisierungen des Kernel Codes werden Minderungen aus dem Kontrollfluss entfernt, wenn der jeweilige Schutmechanismus deaktiviert ist. In Kombination bieten der Dienst, das Kernelmodul und die Kernelerweiterungen die dynamische Rekonfiguration von Hardware-Schwachstellenminderungen. Die Evaluierung zeigt, dass der Einsatz dynamischer Rekonfigurationen von Schutzmechanismen gegen Hardware Schwachstellen und die Anpassung von deren Konfiguration an den Systemzustand das Potenzial hat, die Effizienz des Systems signifikant zu verbessern. Die Evaluierung zeigt auch, dass das Design und die Implementierung der dynamischen Rekonfiguration von Hardware-Schwachstellenminderungen, wie sie in dieser Arbeit vorgestellt werden, mit nur minimalem Laufzeit-Overhead in den Linux-Kernel integriert werden können. Somit kann auf Basis dieser Arbeit weitere Forschung zur Rekonfigurationen von Schutzmechanismen gegen Hardware-Schwachstellen durchgeführt werden.

CONTENTS

1 INTRODUCTION

For providers of computing systems, for example, cloud computing services, as well as their customers, execution costs are the defining factor in task-scheduling and application-design decisions. Costs can be understood in terms of performance and run time, as well as in terms of energy efficiency and energy consumption, among others. Both providers and the customers strive to minimize these costs. For providers, reducing costs allows for more throughput and higher revenue. For customers, higher performance and reduced execution time translate to lower fees. But, applications and their execution also have to satisfy security demands. Security measures typically induce additional overhead as protection mechanisms have to be executed, which contrasts with minimizing execution costs. Thus, these mechanisms should be designed to induce minimal overhead and only be deployed if currently required.

The operating system is the key component for providing, for example, process isolation and, thereby, protection from other processes deployed on the system. Thus, establishing trusted and secure system software is essential to providing a trusted execution environment. This trusted environment is threatened by attacks targeting hardware vulnerabilities, which may leave the operating system exploitable. With the disclosure of vulnerabilities such as Spectre [38] and Meltdown [45], attacks that can extract privileged information from within the operating system have become public. The root cause of these vulnerabilities lies in the hardware implementation of the processor. Thus, these vulnerabilities can only be completely removed in new processor revisions. In already deployed hardware, the vulnerabilities have to be mitigated through microcode updates or software constructs. These mitigations induce significant overhead both in performance and energy demand [32, 46, 1, 53]. In current Linux kernel versions, system users have to decide at boot time whether they require protection from such hardware vulnerabilities. So, if at least one application handles sensitive data or if it is unknown whether future applications may require protection, mitigations are typically enabled to ensure security. The corresponding overheads are accepted as a necessary evil. In terms of optimizing the system's and individual applications' performance and power consumption, a more narrowly targeted approach to mitigating hardware vulnerabilities is required. This requires run-time control of mitigations. Additionally, some vulnerabilities can be mitigated by using different combinations of mitigations. Tailoring this configuration of protection mechanisms individually at run time to the workload deployed on a system can also allow for an increase in the system's efficiency. In combination with dynamically en- or disabling mitigations as needed, these two methods present a way of optimizing the efficiency of computing systems.

This thesis introduces dynamic hardware-vulnerability mitigations in the Linux kernel and their use in optimizing a system's performance, energy demand, and protection configuration considering both the current system state and the workload in execution. By allowing for dynamic control of mitigations that previously could only be configured at boot time, further potential

for run-time optimizations of performance and energy efficiency are available. Based on system-state information and applications' protection demands, the hardware-vulnerability mitigation configuration is continuously adapted. If required, mitigations are enabled, and if applications' demands change or processes requiring protection finish execution, mitigations can be disabled again to eliminate unnecessary costs. In addition, the choice of mitigation is made based on the system characteristics as well as current load measurements. With these measures, both the underlying system and applications are protected while performance and energy efficiency are optimized.

The thesis is structured as follows. Chapter 2 gives an overview of background knowledge on hardware vulnerabilities and their mitigations in the Linux kernel. Research and findings related to the work of this thesis are introduced in Chapter 3. In Chapter 4, the concept and design of dynamic hardware-vulnerability mitigation reconfigurations are presented. This is followed by a description of the design's realization and implementation of the components required for dynamic reconfigurations. An analysis and evaluation of the implementation of hardware-vulnerability mitigation reconfigurations are presented in Chapter 6. Chapter 7 gives a short overview of future work to develop dynamic reconfigurations further. This thesis concludes with Chapter 8 and a resume of the design, implementation, and evaluation of dynamic hardware-vulnerability mitigation reconfigurations in the Linux kernel.

2 FUNDAMENTALS

This thesis introduces dynamic reconfiguration of different hardware-vulnerability mitigations in the Linux kernel and their use in optimizing a system's performance and energy demand. As background for the design and implementation of dynamic reconfigurations and their impact on efficiency, this chapter provides explanations of hardware vulnerabilities and their mitigations in the Linux kernel. Section 2.1 focuses on the Spectre, Meltdown, and Microarchitectural Data Sampling (MDS) hardware vulnerabilities, explaining how these vulnerabilities can be exploited to extract privileged information from a system. Section 2.2.1 describes mitigations for the respective vulnerabilities and possible implementations in the Linux kernel. The mitigations are grouped by hardware and software implementation for the Spectre vulnerabilities, Kernel Page Table Isolation (KPTI) as Meltdown mitigation, and Core Scheduling as mitigation for MDS-based attacks across logical CPUs.

2.1 Hardware Vulnerabilities

Mitigating vulnerabilities is essential to providing secure computing environments. There are two types of vulnerabilities that system designers and engineers have to provide mitigations for. On the one hand, software vulnerabilities, or bugs, must be fixed so that the security of systems deploying the respective programs is not compromised. This class of vulnerabilities is typically addressed by the software engineers, who then provide patches that fix the underlying problems. But vulnerabilities need not necessarily only affect software. The second version of the Internet Security Glossary in RFC 4949 [63] defines a vulnerability as a *»[...] flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy.«*. A computing system's design and its implementation, namely its hardware, can also be affected by vulnerabilities that threaten security. This section focuses on such hardware vulnerabilities and how they enable malicious processes to leak data from otherwise protected memory. While there exist multiple hardware vulnerabilities that affect current processors, the following sections focus on the prominent side-channel attacks that exploit speculative and out-of-order execution. First, Spectre attacks [38] and underlying vulnerabilities are explained, with a focus on variant 1 and variant 2. This is followed by a description of Meltdown [45]. Lastly, the Microarchitectural Data Sampling (MDS) vulnerabilities [8, 56, 57] are introduced.

2.1.1 Spectre Side Channels

With their 2018 publication, Kocher et al. [38] introduced a new class of microarchitectural side-channel attacks: the Spectre attacks. By exploiting speculative execution and branch prediction, Spectre attacks can leak secrets of user applications.

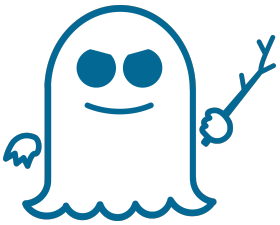


Figure 2.1 – Spectre hardware vulnerability logo, taken from [62].

Speculative execution is a key optimization of modern processors [30, p. 333 ff.]. By speculating on the results of instructions, the CPU can start the execution of following instructions that may depend on the former. This allows the processor to fully exploit its pipelining and instruction-level parallelism as it does not need to stall execution until the first instruction has been retired. Retiring, here, refers to the completion of an instruction's execution, on which the results are committed to memory or register and are, thereby, made visible to the system. All instructions that are executed speculatively are only retired once the original instruction has been completed, and the processor can determine whether its speculation was correct. If the predicted control flow was accurate, the instruction executed in the meantime could be retired and execution could continue. In case of a misprediction, the control flow has to be rerolled to the starting point, which also rewinds register states, and the correct instructions are fetched and executed. Even though speculatively executed instructions are not retired and are, thus, not exposed to the system state, they do result in changes to the microarchitectural state that are not reverted [38]. If, for example, load instructions are part of the speculated branch, their execution may change the contents of the caches and cause new data to be fetched. On rerolling wrongfully executed instructions, the pipeline is flushed and the register state is reverted, but the original cache state is not restored. Attacks, such as Spectre, leverage these changes in microarchitectural state to leak information from privileged memory areas or the address space of other processes.

Spectre attacks can be grouped by variants that differ in the chosen attack vector. The two variants presented in [38] are further explained based on examples provided in the paper.

Spectre Variant 1) The first variant of Spectre attacks exploits the speculative execution of conditional branches. An attacker can control which addresses are leaked by leveraging array out-of-bounds checks. By mistraining the processor's branch predictor, the CPU can be primed to speculatively execute the wrong branch. During the faulty execution, the memory contents for which the attacker does not possess the required permissions are accessed and loaded into the cache. Even though the processor will eventually determine that the wrong path has been taken and revert its state, the illegally accessed data remains cached and can be extracted via cache side-channel attacks.

Figure 2.2 lists the example from [38], which can be used to execute a Spectre variant 1 attack. In this example, a secret byte `k` located at a known address is targeted. The attacker controls two arrays `array1` and `array2`, `array1_size`, as well as `x`. `x` is chosen so that `x` = (address of `k`) - (base address of `array1`). Thus, `array1[x]` points to the secret byte. As another prerequisite, the attacker has to ensure that `k` is caches but `array1_size` and `array2` are not. When running the attack, first, the control flow reaches line 1 of Figure 2.2. `x` is actually out-of-bounds of `array1` and the

```
1  if (x < array1_size)
2      y = array2[array1[x] * 4096];
```

Figure 2.2 – Code example from [38] that can be utilized for Spectre variant 1 attacks.

comparison should evaluate to false. But, as `array1_size` is not cached, it has to be fetched from the main memory first. This takes several cycles, during which the processor continues execution speculatively. The branch predictor is used to determine the most likely branch. As this predictor has been mistrained prior to the attack, it predicts the branch to be taken. Thus, execution continues in line 2 until `arra1_size` has been fetched from memory, and the correct branch can be determined.

In line 2, the secret byte `k` is accessed via `array1[x]` and loaded in a register. As `k` is already cached, reading the byte is finished quickly. `k` is then multiplied by the system's page size, 4 KB in this example. This is followed by accessing the memory at `array2[k * 4096]`, which results in a cache miss. The corresponding value has to be loaded from memory to the cache. While this is underway, the processor should have determined that a misprediction occurred in line 1. Even though the state of registers prior to speculation is restored and execution continues with the correct path, changes to cache are not reverted. Also, previously triggered loads are still completed and `array2[k *4096]` is fetched. From this cache entry, `k` can be derived by executing cache side-channel attacks. For example, by iterating over all entries of `array2` and measuring how long each access takes. Accessing the previously cached entry will have a significantly shorter latency. Based on the index of the entry, `k` is then derived and, thereby, leaked to the attacker.

Spectre Variant 2) Variant 2 of Spectre also leverages speculative execution to leak information. But instead of targeting conditional branches, speculation on indirect branches is exploited. By mistraining the branch predictor, an attacker can control which target address of an indirect branch is used for speculative execution. If, for example, the target of an indirect jump is the result of a division, the processor will speculate on the result to avoid stalling execution until the division has been completed. Attackers can utilize this speculation to control at which address in a victim's context execution continues. Thus, Spectre variant 2 uses code segments in the victim's address space to leak data. These segments are also called gadgets. Figure 2.3 gives an example of code that can be used as a Spectre variant 2 gadget.

```
1 jmp [function]
2 ...
3 mov R2, [R1]
4 shl R2, 0xc
5 add R3, R2
6 mov var, [R3]
```

Figure 2.3 – Illustration of a simplified Spectre variant 2 gadget code that can be utilized to read sensitive data. Based on the explanation in [38].

By controlling the contents of registers `R1` and `R3`, the attacker can, similar to Spectre variant 1 attacks, cause a secret byte to be leaked. On reaching the indirect jump in Figure 2.3, the processor has to predict the jump target while `function` is executed to determine the target address. As the predictor has been mistrained prior to the attack, an attacker-controlled virtual address is predicted. In this example, the address of the gadget starting with line 3 is speculated to be the jump target and speculative execution continues with the corresponding instruction. First, the value pointed to by the attacker-controlled content of `R1` is loaded to `R2`. `R2` now contains the secret byte the attacker wants to leak. This byte is multiplied by the page size, 4 KB in this case, and summed up with the attacker-controlled register `R3`. `R3` contained the base address of the attacker's probe array and now points to an address dependent on the secret byte. With line 6, the memory addressed by `R3` is accessed and loaded into the cache. During this load and fetch from the main memory, the calculation of the indirect-jump target address may finish and the misprediction is detected. Thus, the state is reverted by restoring registers and flushing the pipeline. Changes to the cache persist across this reroll, leaving the illegally accessed entries vulnerable to extraction through cache side-channel attacks. For example, the entry already cached can be determined by iterating over all entries of the probe array and timing the access latency. From the index of the cached entry, the attacker can derive the secret byte.

The entire attack is carried out in the victim's address space and only register values are directly controlled by the attacker. Therefore, the branch predictor has to be mistrained by the attacker to target a certain address in another context. This is possible due to the predictor transferring its data from one context to another. Thus, the mistraining can be done in the attacker context with a virtual address that corresponds to the location of the gadget in the victim address space.

Summary By exploiting speculative execution and branch prediction, the Spectre attacks [38] expose critical weaknesses of current processors. The corresponding hardware vulnerabilities allow malicious user programs to extract potentially sensitive data from other processes. This is done by manipulating the branch predictor across different contexts, which enables attackers to control the predictions of speculative execution. Even though mispredictions are eventually detected by the processor and all speculatively executed system state is reverted, changes to microarchitectural state, such as cache contents, persist. Illegally accessed memory can thereby be leaked with the use of cache side-channel attacks.

2.1.2 Meltdown Side Channel

Along with the Spectre hardware vulnerabilities, Lipp et al. [45] published the Meltdown side-channel attack. With Meltdown, malicious users can overcome address-space isolation and gain access to all mapped virtual memory. The underlying hardware vulnerability enabling Meltdown-based attacks is located in the implementation of out-of-order execution and the corresponding instruction reordering.

Figure 2.4 – Meltdown hardware vulnerability logo, taken from [62]

Out-of-order execution is an important feature of modern, superscalar processors to fully utilize instruction-level parallelism [29, p. 167 ff.]. Reordering instructions allows the CPU to avoid stalls that would occur with in-order execution. If instruction `b` takes the result of instruction `a` as argument, `b` can only be executed once the result of `a` is known. Now, if instruction `a` requires loading data from the main memory, its completion may require several cycles. With in-order execution, the entire pipeline would have to stall. A processor implementing out-of-order execution can identify which instructions depend on one another and which do not. In a scenario as described above, instead of stalling, instructions following instruction `b` but independent from instruction `a` can be issued to avoid idling. While increasing the processor's performance, in some cases, the reordering can result in unintended behavior that opens up race conditions between privilege checks and memory accesses. Meltdown exploits exactly one such race condition to read kernel memory. A prerequisite for Meltdown attacks are the kernel mapping in user-space page tables, which was standard prior to the hardware vulnerability's disclosure. With Meltdown, Kernel Page Table Isolation (KPTI) was introduced in the Linux kernel to mitigate the vulnerability by removing most of the kernel mappings from user-space page tables. KPTI is further described in Section 2.2.3.

Figure 2.5 lists a simplified example of a possible Meltdown attack. The example is based on the explanation in [45]. To successfully extract kernel memory from user space, an attacker requires control of two registers, `R1` and `R3`, as well as an array located at `[R3]`. `R1` contains the target kernel address. In line 2, the secret byte located in the kernel is loaded into `R2`. As the kernel is mapped in user space, translating the address yields a valid physical address from which memory can be loaded. Part of loading memory is the permissions check to ensure a process legally accesses data. Reading memory and checking permissions are separate operations in the processor

execution, and both require multiple cycles. To avoid pipeline stalls, out-of-order execution ensures that the following instructions are already issued to mask the latency of previous instructions. Thus, while the permission check is still underway and the secret byte is already cached, instructions from line 3 on can already be issued. First, the secret value is multiplied by the page size in line 3. This is followed by calculating a valid address in the attacker's context using the secret value. By using the contents of R2 as the offset from the attacker-controlled array pointed to by R3, an entry of the array is determined and loaded from memory in line 5. As soon as the array entry is fetched from memory and added to the cache, the attack has succeeded. With the completion of the permission check, the exception corresponding to the illegal memory access is raised. Then, all instructions following the corresponding load instruction are rerolled, and the system state, such as register values, is reverted. As this reroll does not revert cache changes, the loaded array entry is still present. By executing a cache side-channel attack, the attacker can determine which entry has been cached and, from that, derive the secret byte.

```
1 xor R2, R2
2 mov R2, byte [R1]
3 shl R2, 0xc
4 add R3, R2
5 mov var, [R3]
```

Figure 2.5 – Illustration of a simplified Meltdown attack. The example is based on [45].

Summary By exploiting race conditions between memory accesses and permission checks, Meltdown [45] enables reading any kernel-mapped memory. This race condition is rooted in the out-of-order execution of superscalar processors. Out-of-order execution allows execution to continue while the permission checks of memory accesses are still underway. Until the checks have finished, sensitive data can be loaded and extracted from caches via side-channel attacks. The disclosure of Meltdown resulted in the introduction of Kernel Page Table Isolation (KPTI) in the Linux kernel, as removing kernel mappings in user space mitigates the hardware vulnerability.

2.1.3 Microarchitectural Data Sampling Side Channels

Modern processors contain various internal buffers required for optimizing performance and implementing out-of-order execution. The Microarchitectural Data Sampling (MDS) vulnerability has identified these buffers as possible microarchitectural structures that can leak information. Attackers leveraging MDS vulnerabilities exploit speculative execution to sample data from other processes. In contrast to Spectre and Meltdown, malicious programs cannot control from which target address secrets are leaked but instead forwards "in-flight" data, that is, data currently processed by the CPU for other processes, to side channels.

Figure 2.6 – Microarchitectural Data Sampling (MDS) logo, taken from [4]

With RIDL [56], Fallout [8], and ZombieLoad [57], three prominent attacks exploit the MDS hardware vulnerabilities. The three attacks target CPU-internal buffers, such as Line Fill Buffers (LFBs) or Store Buffers, to sample memory accesses of other processes. LFBs are used by the processor to track outstanding memory requests [56]. The buffers are also used for optimizations, such as merging multiple stores to the same address or speculatively loading data directly from not yet processed stores. Store Buffers (SBs) contain information of pending store operations. Queuing stores in the SBs serves as passing the write instruction to the memory subsystem. This allows the processor to continue execution and retire store operations even if the data has not been written to memory yet. All requests in the buffer

are processed in the correct order by the memory subsystem. Loading from memory, thus, requires monitoring the Store Buffer to check whether writes to the address are still pending.

Fallout exploits aggressive speculation during load operations. Canella et al. describe so-called Write Transient Forwarding (WTF) shortcuts that forward data from SBs to load operations of only partially matching addresses. WTF shortcuts exploit speculation in case the address translation of a load fails. If the full physical address of the load is not valid, but the address partially matches an entry in the SB, this is considered a "hit" nonetheless, and the data from the previous store is speculatively forwarded. Even though the load will never be retired, the data forwarded via the WTF shortcut can be extracted from the microarchitectural state, for example, through cache side channels. RIDL [56] attacks are based on the same principle as Fallout but target LFBs instead of SBs. By exploiting speculative load forwarding from LFBs, data accessed in another context but also tracked in the LFB of the same physical core is loaded to the microarchitectural state in the attacker's context. ZombieLoad [57] also samples data of other processes from the LFB, though they exploit an apparent implementation issue in Intel CPUs. Schwartz et al. observed that under specific conditions, stale data is forwarded from the LFB. In particular, loads that result in L1-cache misses may first receive outdated values from the LFB before the load is reissued to collect the correct data.

To mitigate MDS vulnerabilities, the affected internal buffers are cleared on context switches to avoid leaking information between processes [48]. This mitigation protects processes that share logical cores. If processors implement Symmetric Multithreading (SMT), some CPU-internal buffers may be shared between siblings. With such systems, merely flushing buffers on context switches is not enough, as cross-HT can be utilized by an attacker while the victim is active. In these cases, only disabling SMT can fully mitigate MDS. With version 5.14 of the Linux kernel, a new feature was introduced as an alternative to turning off SMT to mitigate cross-HT attacks. By utilizing Core Scheduling, processes can impose restrictions on which processes can be scheduled on a sibling core. Section 2.2.4 gives further details on Core Scheduling and its use to mitigate cross-HT attacks exploiting MDS vulnerabilities.

Summary With Microarchitectural Data Sampling (MDS), a class of hardware vulnerabilities became known, which allow attackers to sample data accessed by other processes. MDS-based attacks utilize CPU-internal buffers and speculative forwarding of data stored in these buffers to extract memory contents they could not access otherwise. These vulnerabilities also enable attacks between logical cores that share CPU buffers. This class of cross-Hyper-Threading (HT) attacks requires more mitigations than MDS-based attacks on systems without hardware multithreading. To successfully prevent exploitation of MDS vulnerabilities, current versions of the Linux kernel provide Core Scheduling as an alternative to disabling SMT.

2.2 Vulnerability Mitigations in the Linux Kernel

The hardware vulnerabilities introduced in Section 2.1 all pose significant risks to a system's security. To provide secure systems to all users, attacks exploiting these vulnerabilities have to be averted. Ideally, this is done by fixing the underlying hardware vulnerabilities in the hardware itself. As this may require new processor revisions or even new processor designs, patching vulnerabilities would equal exchanging the hardware. While microcode updates can patch some problems, vulnerabilities still persist. Instead, mitigations are required that reduce the attack surface to render attacks too inefficient to use. Such mitigations can be implemented in hardware or in software. This section introduced such mitigations for the vulnerabilities of Section 2.1. First, hardware mitigations for Spectre variants are explained. This is followed by an introduction of software mitigations that, in combination with the hardware mitigations, can mitigate Spectre variant 1 and variant 2 attacks. Kernel Page Table Isolation (KPTI), a mitigation for the Meltdown vulnerability, is explained in detail, followed by an explanation of Core Scheduling, which can be used to mitigate cross-HT attacks based on MDS vulnerabilities.

2.2.1 Spectre Hardware Mitigations

Spectre attacks exploit the speculative execution of modern superscalar processors. To mitigate vulnerabilities located in hardware, patches or updates also provided in hardware are typically more efficient solutions compared to software mitigation. For the Spectre vulnerabilities, several mitigation implementations exist that, in combination with software-based implementations, aim to protect from Spectre attacks. In the following, the different hardware mitigation implementations are explained.

Indirect Branch Restricted Speculation (IBRS) To restrict the speculation of indirect branches and protect branch prediction from Spectre attacks, the IBRS feature was introduced [13, 61]. IBRS functions like a barrier between different predictor modes[1].If IBRS is enabled, indirect branches and predicted targets cannot be controlled from a less privileged mode. To ensure that IBRS protection is provided, the corresponding bit in the speculation control MSR has to be set on every potential switch between predictor modes. The use of IBRS requires additional software mitigation. The Return Stack Buffer (RSB), which contains predicted target addresses, has to be overwritten so that all entries from the previous predictor mode are purged. IBRS only protects from attacks from a less privileged prediction mode. If protection from attackers within the same mode is required, additional (hardware) mitigations have to be enabled.

Enhanced / "Always-on" IBRS For more recent microarchitectures, an improved version of IBRS is available [13, 61]. This feature is named enhanced IBRS (eIBRS) for Intel processors and "always-on" IBRS for AMD hardware. As normal IBRS requires writing the speculation control Model-Specific Register (MSR) on every transition between predictor modes, it induced significant overhead [9]. With eIBRS, the mitigation only has to be enabled once. As with normal IBRS, rewriting the RSB is required to evict entries from less privileged predictor modes. Along with AMD-specific retpoline implementations, eIBRS was found to be insufficient as a replacement for the retpoline software mitigation in 2022 [6]. This is reflected in the default mitigation selection of the Linux kernel, starting from version 5.18.

[1] Predictor or prediction modes correspond to the current privilege level (CPL) [61, 17]. From most to least privileged, the levels are host-supervisor, host-user, guest-supervisor, and guest-user.

Indirect Branch Prediction Barrier (IBPB) IBPB provides a barrier for controlling predicted targets of indirect branches [12, 61]. Its use guarantees that no instruction executed before the barrier can control the target prediction of instructions after the barrier. Combining IBPB with IBRS provides protection of branch target prediction from attacks within the same predictor mode. The kernel can issue these barriers via setting the corresponding bit in the CPU's prediction-command Model-Specific Register (MSR). While the barriers can be used to restrict speculative execution on a logical core, they do not provide protection from Spectre attacks between sibling cores. For protection across siblings, Single Thread Indirect Branch Predictor (STIBP) is required.

Single Thread Indirect Branch Predictor (STIBP) To effectively mitigate Spectre variant 2 attacks between two logical CPUs of one physical core, STIBP can be utilized [19, 61]. If STIBP is activated on one logical core, branch predictors can no longer be used by the logical sibling cores. Thus, mistraining the predictor from a sibling CPU is not possible. Enabling STIBP on one logical core effectively enables it for all siblings.

2.2.2 Spectre Software Mitigations

Mitigating attacks leveraging speculative execution requires extensive efforts. While hardware mitigations were introduced with new processor revisions, as presented in Section 2.2.1, they cannot fully protect from all Spectre attacks. To provide comprehensive mitigations for the vulnerabilities explained in Section 2.1, additional software mechanisms are required.

Usercopy/Swapgs Barriers For the Spectre variant 1 vulnerability, the kernel provides mitigations in the form of usercopy and swapgs barriers. Both barriers are implemented via load-fence instructions, which ensure that all previous load-from-memory instructions are serialized and have been retired [14]. Thus, speculative execution of conditional branches following these barriers is restricted. In the kernel-entry and -exit paths, load-fence barriers are required to prevent speculatively skipping the conditional swapgs instruction.

Retpolines To mitigate attacks on the kernel, the indirect calls and jumps of the kernel have to be protected from being leveraged by speculative attacks. This can be achieved by the hardware mitigation enhanced IBRS or by the use of retpolines. The name retpoline is a combination of the words "return" and "trampoline" and effectively describes the mitigation. Instead of preventing speculative execution, retpolines control it by trapping the speculation in a pause loop until the actual branch target address is determined [18, 65]. To enable the use of retpolines, the RSB has to be filled with the `pause; lfence; jmp` constructs needed to trap the speculative execution in an infinite loop. For AMD systems, a more efficient hardware-specific retpoline implementation was developed [3], though it is no longer in use due to inherent race conditions [6]. Retpolines replace indirect branches in every program that requires protection from Spectre variant 2. To fully protect the kernel from Spectre variant 2 attacks, speculation when calling into firmware has to be restricted with the additional use of IBRS.

To mitigate Spectre variant 2 attacks, three attack vectors have to be considered. The kernel has to be protected from malicious user-space processes, user-space processes have to be protected from one another, and attacks from rogue guests against the kernel or other guests have to be mitigated. To mitigate attacks on the kernel, the indirect calls and jumps of the kernel have to be protected from being leveraged by speculative attacks. This can be achieved by the hardware mitigation

enhanced IBRS or by the use of so-called retpolines. The kernel mitigates attacks within user space by issuing Indirect Branch Prediction Barrier (IBPB) commands on context switches between user-space tasks. IBPB itself is a hardware feature used to restrict speculative control across a barrier. It is further described in Section 2.2.1. The kernel offers the use of IBPB barriers conditionally and unconditionally. With the latter, the barriers are issued every time a context switch occurs. With conditional IBPB, processes can request protection via system calls. Every time execution switches between tasks of which at least one is protected, the IBPB barrier is inserted. If the system provides SMT support, user-space processes also have to be protected from Spectre attacks across resources shared between sibling cores. On active SMT, the kernel activates the Single Thread Indirect Branch Predictor (STIBP) hardware mitigation, which is further described in Section 2.2.1. As with IBPB, STIBP can either be enabled for the entire system or be activated per task.

With the mitigations to protect kernel and user space from each other, the kernel also provides protection from malicious guests. The use of either enhanced IBRS or retpolines mitigates attacks exploiting speculation across indirect branches. Guests on logical cores of the same physical core are protected from one another via the STIBP hardware mitigation.

2.2.3 Kernel Page Table Isolation

For attackers, the kernel is the key to unlocking all of the system's secrets. Protecting the kernel from malicious processes is fundamental to providing a secure system to all users. A common attack vector is leaking kernel address-space information via microarchitectural side channels. One of the most severe vulnerabilities by which not only address mappings but the kernel's entire memory can be leaked is the Meltdown hardware vulnerability. The workings of Meltdown are described in Section 2.1.2. This section focuses on the kernel's mitigation: Kernel Page Table Isolation (KPTI).

The page-table isolation mechanism of KPTI was first presented by Gruss et al. [27], though it was still named KAISER (Kernel Address Isolation to have Side channels Efficiently Removed) at the time. Originally, KAISER was devised to mitigate attacks on Kernel Address Space Layout Randomization (KASLR). KASLR aims to protect the kernel by randomizing its memory location at boot time. This randomization, however, is vulnerable to attacks exploiting, for example, timing side-channels [36]. KAISER aimed to counter such attacks by removing the information leak in the form of the kernel mappings present in user-space page tables. A kernel patch set implementing the KAISER concept was proposed by Dave Hansen [28]. In light of the Meltdown vulnerability, this patch set was adapted and merged into the kernel as KPTI. Page-table isolation allows removing most of the kernel mappings from user-space page tables by managing two sets of page tables for each process [52]. The first set consists of tables for use in the kernel, while the second set contains page tables that are used in user mode. The kernel page tables are structured similarly to the previously used single-set page tables and still contain both kernel-space and user-space mappings. The page tables used in user mode, in contrast, no longer contain all kernel mappings. Instead, only the data necessary for entering and exiting the kernel and the interrupt descriptor table are mapped. On each kernel entry and exit, the page tables are exchanged. Thus, the kernel still has access to all memory mappings, while user-space processes can only access minimal kernel mappings. Thereby, side-channel attacks that leak data from mapped areas can no longer target kernel mappings.

While KPTI successfully mitigates Meltdown-based attacks, managing two sets of page tables per process and switching between the copies on each kernel entry/exit induces run-time overhead. Exchanging the page tables is implemented by changing the `CR3` register, which locates the page directory. Thus, the register must be written every time a process switches from user to kernel mode and back. Modifying this register requires around 100 cycles [52]. Thus, the often executed kernel entry/exit paths take significantly longer. Additionally, if the CPU does not provide Process Context

Identifiers (PCID) support, each switch between page tables requires flushing the entire Translation Lookaside Buffer (TLB) to ensure no kernel mappings are cached on user-space entry. The operation for flushing the TLB is in itself expensive in terms of cycles required for its execution. Even more costly is the following repopulation of the TLB via page-table look ups. If PCIDs are available, only partial TLB flushes are required when switching between kernel and user-space page tables, which reduces the overhead significantly. How costly exactly KPTI is in terms of performance and energy overhead depends on the processor and workload in execution [32, 26, 55].

2.2.4 Core Scheduling

To improve the efficiency of superscalar CPUs, the technique of Symmetric Multithreading (SMT) to fully utilize the multiple function units was introduced [64, 22]. With SMT, resources of one physical core are shared between two or more "sibling" threads that work in parallel. Thus, one physical core is presented to the system as multiple logical CPUs. While SMT implementations, such as Intel's Hyper-Threading (HT), allow for significant performance and efficiency improvements, they can also leave the system vulnerable to attacks that leak data from shared resources. The principle of one such hardware vulnerability, Microarchitectural Data Sampling (MDS), is described in Section 2.1.3. A natural mitigation for such cross–Hyper-Thread attacks is disabling SMT on vulnerable systems. While this effectively prevents the attacks, it can also result in significant performance penalties. As a more-lightweight alternative to disabling SMT, the Linux kernel introduced Core Scheduling [11].

Core Scheduling provides process isolation by defining trust relationships. Only tasks that have established trust in one another can be scheduled concurrently on sibling cores [10]. This restricts which tasks share a physical core and, thereby, resources like CPU buffers that may be used to leak sensitive data. With Core Scheduling, the kernel effectively mitigates cross-HT attacks between user-space processes that exploit, among others, variants of the MDS hardware vulnerabilities. In contrast to disabling SMT, the number of available logical CPUs is not reduced, though some may not be available for all processes. Thus, the overall impact on the system's performance and efficiency is softened. Trust relationships between processes are defined via so-called cookies that are assigned to processes. Tasks with the same cookie trust each other and can be scheduled on sibling cores. Initially, all processes are assigned a default cookie with a value of zero. If tasks require isolation, unique cookies are created and assigned to the respective tasks. These cookies are inherited on `fork/clone` and `exec`. Creating, changing, or removing cookies from user space requires root privileges. If Core Scheduling is in use, the scheduler may have to prioritize security over priority. Only processes that share a cookie with the process on the sibling core are available for scheduling on the other sibling, even if tasks with higher priority are queued. If no trusted task is available, the idle task is scheduled, which is trusted by all system processes. In addition to idle tasks, kernel threads are also trusted by all user-space tasks.

While Core Scheduling is an alternative to disabling SMT in many scenarios, it can only mitigate cross-HT attacks between user-space processes. Guest-to-guest attacks exploiting Page Table Entry (PTE) inversion and attacks from or targeting kernel threads are still possible. Also, the use of Core Scheduling induces additional scheduling overhead due to the synchronization of trust relationships across sibling cores. If the system is only lightly loaded, this overhead can cause Core Scheduling to not keep its promise of higher performance compared to disabling SMT. Thus, the system load should be considered before utilizing either Core Scheduling or deactivating SMT to provide process isolation. Even though Core Scheduling was primarily designed as a security measure, it may be used to enhance performance in general [47]. By grouping related tasks that profit from shared resources and, thus, ensuring they are scheduled on the same physical cores, their performance could be improved.

3 RELATED WORK

This thesis introduces dynamic reconfiguration of hardware-vulnerability mitigations in the Linux kernel and its use in optimizing a system's performance and energy efficiency. While a plethora of research for different aspects of this topic exists, to the best of the author's knowledge, this thesis presents the first work focusing specifically on dynamic control of Spectre and Meltdown mitigations as well as optimizing configurations of cross-HT attack mitigations. This chapter presents an overview of research fields related to the topic of this thesis. First, research into reconfiguring operating systems in terms of security and isolation primitives is introduced. As this thesis focuses on security and mitigation configurations, the field of research into optimizing a system's efficiency by (automatically) adapting general OS configurations as presented by [7, 23, 31, 33] is not included in this chapter. A second research topic outlined in the following is that of dynamic reconfiguration of security settings and protection mechanisms of a hardware system itself. This is followed by presented research into the automatic detection of hardware vulnerabilities and the detection of attacks exploiting such system weaknesses.

Operating-System Reconfiguration While this thesis proposes run-time reconfiguration of operating-system features, namely the hardware-vulnerability mitigations, related work has researched the reconfiguration of the operating system in its entirety. Changing the underlying security and isolation primitives can be utilized to tailor the operating system to its workload's requirements. One such approach is presented by Lefeuvre et al. [41, 42] in the form of FlexOS, a library operating system based on Unikraft [39]. FlexOS allows reconfiguring its isolation strategy at compile time, a configuration usually fixed at design time. Isolation in FlexOS is based on defining domains, so-called compartments. The user can determine which hardening methods, such as Control-Flow Integrity (CFI) or address sanitization, are deployed for the different domains. OS components are then allocated to the different compartments according to their isolation requirements. Though not explicitly targeting the mitigation of hardware vulnerabilities, LeFeuvre et al. argue that by offering a wide range of isolation primitives that rely on different hardware, they can mitigate vulnerabilities by changing the isolation mechanism. In contrast to the design presented in this thesis, FlexOS does not provide dynamic reconfiguration at run time and not on the level of different hardware-vulnerability mitigations. Instead, they focus on the general isolation of an operating system, its components, and different user-space applications.

Similar to FlexOS, Drescher presents adaptive memory protection as an approach to provide dynamic isolation in the context of invasive computing [50] in his doctoral thesis [21]. With Drescher's approach, the operating system supports dynamically protected and dynamically-unprotected applications. Protected applications are isolated from unallocated, operating-system, and other applications' memory. In contrast, if an application is trusted by the system, protection can be reduced. Also,

trusted applications are enabled to run in privileged mode. The two modes are implemented by defining protected and unprotected mappings and adapting page tables accordingly. In contrast to the dynamic mitigation configurations proposed in this thesis, adaptive memory protection targets the basic isolation of processes provided by the operating system. While this thesis introduces mitigation reconfiguration, this does not disable memory isolation to the same extent.

System Reconfiguration A different approach to dynamic mitigation control is presented by Dessouky et al. [20]. In their work, they utilize the reconfiguration of the underlying system components, namely the cache behavior, to adapt security configurations to applications' needs. With HybCache, they present a hybrid cache architecture that dynamically provides isolated execution domains with side-channel–resilient cache behavior. These domains can be fitted to individual processes, code segments, or Trusted Execution Environments (TEEs). As the reconfiguration of the cache behavior incurs a performance penalty, restricting this effect to only vulnerable processes is essential to limiting the overall overhead of protection from cache side-channel attacks. Non-isolated domains and processes, therefore, are still provided with conventional cache behavior. By providing processes with side-channel–resilient caches, hardware vulnerabilities, such as Spectre and Meltdown, are effectively mitigated in the majority of attack scenarios even though speculative execution in itself is not changed. Instead, the most commonly used covert channel to extract the illegally accessed information is removed. Thus, processes are only protected from attacks that utilize these side channels. Similar process-based cache-partitioning schemes to provide security guarantees exist [25, 66, 69, 37]. But Dessouky et al. argue that these approaches are either more limited in their applicability or do not scale as well as HybCache and incur a more significant performance overhead. Compared to the approach of hardware-vulnerability mitigation reconfiguration presented in this thesis, HybCache directly targets hardware configurations and eliminates the need for software mitigations, which are typically less efficient in terms of performance. Thus, hardware reconfigurations are a possible addition to the repertoire of mitigations for different vulnerabilities. As they do not protect from attacks utilizing side channels other than caches, they may not be applicable in all situations. Also, their performance and energy efficiency compared to that of current mitigations offered by the Linux kernel has to be analyzed to determine when, or whether at all, they are viable alternatives to current mitigation implementations.

Independent Vulnerability Detection The reconfiguration of hardware-vulnerability mitigations, as proposed in this thesis, relies on information provided by manufacturers. Based on CPU flags and datasheets, the kernel determines which vulnerabilities are present and which mitigations are available. Therefore, in order to provide a secure system, processor manufacturers have to be aware of vulnerabilities and communicate them to their customers. Here, Oleksenko et al. [51] propose a different approach. They developed Revizor, a testing framework to independently detect vulnerabilities in speculative execution. To do so, they first define speculation contracts to specify memory protection and isolation requirements. The validity of these contracts is then assessed with the help of Model-based Relational Testing. With their approach, they are able to detect violations of contracts concerning L1-cache data. This allows them to identify vulnerabilities that enable Spectre and Microarchitectural Data Sampling (MDS) attacks as well as previously unknown attack variants. With the limitation to the L1 cache, Revizor is not able to detect all hardware vulnerabilities in speculative execution. For example, it cannot identify the underlying vulnerabilities of Meltdown. Their approach differs from other black-box detection mechanisms, such as Medusa [49], that were developed to identify specific vulnerabilities instead of testing for weaknesses in speculative execution in general. The automatic detection of hardware vulnerabilities could be utilized in addition to the dynamic reconfiguration of mitigations. By employing this approach, for example,

in the Linux kernel, the manufacturer information could be verified. Also, if previously unknown variants of vulnerabilities are detected, and processes handling highly sensitive information are executed, the system could precautionarily enable mitigations.

Attack Detection and Dynamic Isolation The approach proposed in this thesis aims to dynamically reconfigure mitigations so all applications receive the protection they require, but mitigations are not enabled unnecessarily. This is done regardless of whether a malicious process is present. Schwarzl et al., in contrast, suggest adapting the security configuration in case attacks are detected. Their "Dynamic Process Isolation" preprint [58] describes their project on this research area that was conducted in cooperation with *Cloudflare*. They developed a mechanism to detect Spectre attacks in the *Cloudflare Workers* edge-computing service and isolate malicious threads from the remaining workers. By isolating attackers, they do not require to deploy system-wide mitigations during execution and, thereby, limit unnecessary overhead. The detection of Spectre attacks is based on continuously sampling mispredicted and retired branches through hardware performance counters. By comparing the event distribution to the profile of a template attack, they determine threads with suspicious behavior. Potentially malicious workers are then isolated with a mechanism similar to site isolation [54]. Even though false positives occur (0.61 % false-positive rate), they only result in unnecessary overhead for these few threads, as the remaining workers are left unaffected. Even though most threads can be executed without the overhead of mitigations, a small performance penalty of about 2 % overhead persists due to the continuous performance-counter sampling. In contrast to the approach of this thesis, Schwarzl et al. aim to provide security by isolating attackers. Thereby, they shift the performance penalty from the victim processes to the attackers. This approach could be incorporated as an additional mitigation that the dynamic-reconfiguration approach of this thesis can configure. Further research would have to determine whether this approach, which is specifically tailored to *Cloudflare Workers*, proves to be as efficient with different workloads.

4 DESIGN

This chapter presents the design of dynamic hardware-vulnerability mitigation reconfigurations in the Linux kernel and its utilization to improve the system's efficiency. Figure 4.1 gives an overview of the components required for utilizing dynamic reconfigurations and their integration into the system. The reconfigurations are controlled by a user-space–based service. The service functions as a coordinator who receives protection requirements from applications (ⓐ) and determines the optimal mitigation configuration. Run-time control of the different mitigations is provided by a custom kernel module (ⓑ) and the kernel itself (ⓒ). The mainline kernel does not provide support for the dynamic reconfiguration of the Spectre and Meltdown mitigations. Thus, the kernel module is required to offer this functionality to the service. To en- or disable the Spectre and Meltdown mitigations, the kernel module must interact with the kernel and trigger the required kernel-internal changes (ⓓ). Information on vulnerabilities and available hardware mitigations is provided by the hardware. This information is collected by the kernel at boot time (ⓔ) and passed to both the kernel module and the service during their initialization.

The following sections present the design of the components and their interactions in more detail. Section 4.1 focuses on the service and its role as the coordinating entity for the mitigation reconfigurations. The kernel module's design is presented in Section 4.2.

4.1 DRConf Service Design

Introducing and utilizing dynamic reconfiguration of hardware-vulnerability mitigations in the Linux kernel requires additional components: a user-space–based service and a kernel module. The service functions as coordinator and primary agent that monitors protection requirements and issues mitigation configurations. The kernel module implements the dynamic control of the Spectre and Meltdown mitigations. This section focuses on the design of the service and its integration into the Linux run-time environment. Figure 4.2 visualizes the service's interaction and information exchange with the kernel and the kernel module.

The service determines the system's configuration of hardware-vulnerability mitigations. The configuration is tailored to the current protection requirements of applications as well as system state information, such as CPU utilization. Reconfigurations encompass the (de-) activation of mitigations or a change in mitigation implementation. The service issues reconfigurations dynamically at run time. If multiple configuration options are available, the service determines the best-suited setting considering three main objectives: Providing all processes in execution with their required protection, optimizing the system's performance, and optimizing the system's power consumption. Typically, these three objectives compete with each other. Adding additional security features introduces an overhead in performance and power consumption. Guaranteeing secure execution of applications is

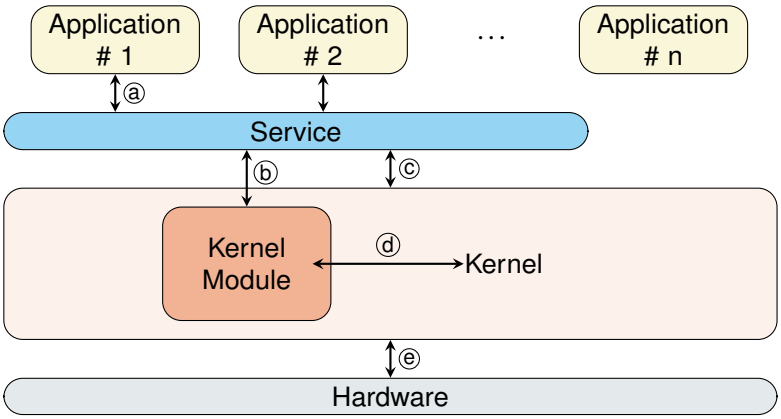


Figure 4.1 – Illustration of integration of dynamic hardware-vulnerability mitigations into the Linux kernel. The service receives protection demands from applications ⓐ and controls mitigations via the kernel module ⓑ and the kernel ⓒ. The kernel module dynamically controls the Spectre and Meltdown mitigations via the kernel ⓓ. The hardware provides the kernel with information on vulnerabilities and available hardware mitigations ⓔ. Interactions not related to the dynamic reconfigurations are omitted for clarity reasons.

non-negotiable and takes the highest priority. Thus, if the system provides the required protection mechanism, it is activated regardless of the overhead incurred. If multiple mitigations can satisfy security requirements, the service chooses the option matching the provider prioritization. The different selection criteria and the selection process itself are further detailed below.

4.1.1 Dynamic Security Configurations

The service dynamically adapts the system's security configurations in terms of hardware-vulnerability mitigations. The current design of the service allows for dynamic reconfiguration of mitigations that have been introduced in the Linux kernel. As of kernel version 5.14, these include mitigations for the Spectre, Meltdown, and Microarchitectural Data Sampling (MDS) vulnerabilities. The focus of the dynamic reconfigurations as proposed in this thesis lies in mitigations of attacks by malicious user-space processes and cross–Hyper-Threading (cross-HT) attacks. As part of future work on the service, mitigations for attacks of or against guest operating systems or guests in virtual machines can be incorporated. Additionally, the configuration options of the service could be extended to include further mechanisms, such as Trusted Execution Environments (TEEs).

As part of this thesis, support for the reconfiguration of the Spectre, Meltdown, and MDS-based cross-HT attacks mitigations was realized. The service can control these mitigations on a hardware-vulnerability level. Thus, it cannot reconfigure the individual protection mechanism that combined form one of the mitigations or influence which concrete mitigation implementation is chosen. To deny Spectre attacks, the Linux kernel offers specific mitigations for variants 1 and 2 of the attacks. Spectre exploits vulnerabilities in the hardware implementation of speculative execution to bypass access controls and extract privileged data. Further detail on Spectre and corresponding mitigations in the Linux kernel are given in Section 2.1.1 and Section 2.2. In addition to mitigations for Spectre, the service can reconfigure the Meltdown mitigation Kernel Page Table Isolation (KPTI). With Meltdown, attackers can leak data from any memory region by exploiting race conditions during the access of kernel mappings in user space. KPTI mitigates such attacks by removing kernel mappings from user-space page tables. Only data required for entering kernel mode persist in the page tables.

On kernel entry, the page tables are switched with a set of page tables that include mappings of both user space and kernel addresses. Further details on both Meltdown and KPTI are given in Section 2.1.2 and Section 2.2.3. In the mainline Linux kernel, the mitigations of Spectre and Meltdown can only be controlled at boot time. Thus, the work on this thesis includes the design and implementation of the run-time reconfigurability of these mitigations. This design and that of the kernel extensions required for this feature are outlined as part of Section 4.2.

In contrast to the Spectre and Meltdown mitigations, the Linux kernel already provides run time reconfiguration of mitigations for cross-HT attacks based on variants of the MDS vulnerability. To fully mitigate such attacks, processes have to be protected from attackers running on sibling cores that exploit shared resources. This can be achieved by disabling Symmetric Multithreading (SMT) entirely so that no processes share a physical core. Additionally, Core Scheduling was introduced in version 5.14 of the Linux kernel. This new feature serves as an alternative to disabling SMT. The two mitigations differ in performance overhead and efficiency depending on the workload characteristics and system state. Thus, if applications request protection from cross-HT attacks, the service has to determine which mitigation is best suited to the current system state and prioritization of the provider and initiate reconfigurations accordingly.

Application-Specific Security Configurations

The security requirements of the applications in execution on the system form the basis for the service's reconfiguration decisions. Applications can request the isolation of all or only a subset of their processes. This request then triggers the service to deploy one of the corresponding mechanisms, such as disabling SMT or Core Scheduling. Which of the possible configurations is chosen depends on workload characteristics as well as the system's overall load and underlying hardware. Thus, the configuration decisions of the service are no "one-size-fits-all" but have to be adapted for each system and each change in the workload. Applications can also call on the service to communicate the absence of sensitive data and the prioritization of performance. If all applications and the underlying system agree on this prioritization, the service can issue the dynamic deactivation of performance-intensive mitigations, such as KPTI. The application-specific information, thus, forms the basis of all the service's reconfiguration decisions and provides the framework for optimizations of the mitigation reconfigurations. The basis for the service's reconfiguration decisions is laid by security requirements of the applications deployed on the system.

Performance- and Energy-Aware Configurations

Providing all processes with the requested protection can usually be achieved with more than one configuration. These configurations may differ in regard to which combination of mitigations is enabled or which mitigation implementations are chosen. Mitigation can be, for example, implemented in software or hardware. Each configuration typically results in different system performance and power consumption compared to the other options. For example, protection from cross–Hyper-Thread attacks can be provided either by disabling Symmetric Multithreading (SMT) to isolate vulnerable processes and their data or by utilizing Core Scheduling. As Hyper-Threads are disabled on all cores without SMT, this effectively mitigates cross-HT attacks. But therein lies the possibility of significantly reducing the system's overall performance as only half the logical cores are available for scheduling and parallel execution. With Core Scheduling, SMT can stay enabled. Isolation is instead achieved by defining which processes share a trust relationship. The scheduler then ensures that only those processes trusting each other are scheduled on siblings of the same physical core. This generally limits the isolation mechanism's negative impact on the overall system performance,

while Core Scheduling's power consumption is typically higher. Though, this observation might not be the case if the system is only lightly loaded, as reported in the kernel documentation [10]. Also, the number of processes for which Core Scheduling is used and the number of process groups that require isolation from each other may influence which of the two mechanisms is the most efficient. Thus, the service must determine which mitigation is best suited to the provider's prioritization while taking the current system state into account.

Optimizing the system's hardware-vulnerability mitigation configurations in regard to performance and energy efficiency is not only achieved by selecting the most efficient mitigation combination if new protection demands come up. Also, mitigations should be disabled once they are no longer required, for example, due to the requesting application finishing execution of critical sections. This, of course, requires the cooperation of applications deployed on the system as they need to update their information at the service's interface should their demands change.

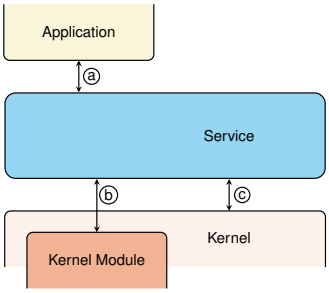


Figure 4.2 – The service's integration into the run-time environment. Applications provide security demands and the service responds on success ⓐ. The service controls mitigations via the kernel module, which reports the mitigations' internal configuration ⓑ. The service provides process isolation via the kernel, which reports the system's vulnerabilities ⓒ. Interactions not related to dynamic reconfigurations are omitted.

To further improve the system's performance and decrease its energy demand through efficient mitigation configurations, the service could be adapted to provide scheduling hints. For example, grouping the execution of processes with similar, reduced requirements could allow the service to disable mitigations. Otherwise, a single process' protection demand would prevent the service from choosing a more efficient configuration. The scheduler might then be required to delay dispatching certain processes in order to allow the service to disable mitigations temporarily. Also, the service could be extended to utilize Core Scheduling not only as a protection mechanism but also as a means to increase performance. Processes that operate on shared data could, for example, profit from Core Scheduling, as they would most likely run on siblings of one physical core and share its resources. Also, Core Scheduling could be utilized to ensure that processes requiring the same hardware resources are executed on separate physical cores. Further potential for the continued development of dynamic reconfiguration of hardware-vulnerability mitigations and the corresponding service is discussed in Chapter 7.

4.1.2 Run-time Environment Integration

The dynamic reconfiguration of hardware-vulnerability mitigations requires interaction between applications, the service, kernel module, and the kernel itself. By positioning the service in the user space and granting it root privileges, these interactions can be successfully realized. Figure 4.2 illustrates the service's integration in the Linux run-time environment. The service is initialized at boot-time and continues its execution in the background for the system's entire run-time or until the user disables it. This allows for the service's continuous availability and enables it to handle frequent changes in the security requirements of user-space applications. The arrows in Figure 4.2 indicate the flow of information between components and passed reconfiguration directives. ⓐ shows the interaction between an application and the service. The application passes changes in its protection requirements while the service responds with an acknowledgment or error code in case protection cannot be guaranteed. Based on information collected from both the kernel and

the implemented kernel module, the service determines the optimal mitigation configuration. The corresponding changes to kernel data structures and calls to internal functions are realized by the service passing information to the module (ⓑ) and the kernel (ⓒ). The reconfiguration of most mitigations, such as the SMT control and initiating code-patching sequences to kernel-internal code paths, requires administrative access to the system. Thus, the service is granted root privileges.

4.1.3 Service Interface

For the service to keep mitigation configuration up-to-date with current protection requirements, the applications must communicate changes in their security demands. Therefore, the service provides a method that applications can invoke anytime during their execution. With the first call, the application is registered with the service and added to its administrative structures. The service, in turn, acknowledges and confirms or, in case of errors or unavailability of mechanisms, rejects changes in protection requirements. This allows applications to handle the unavailability of the required protection and, if needed, terminate their execution.

Summary This section presented the design of the **DRConf** service. In order to utilize dynamic reconfiguration of hardware-vulnerability mitigation configurations, a coordinating entity is required. This thesis proposes the use of a user-space service to act as such a central coordinator. Based on information on hardware and system specifics, the service is tasked with determining mitigation configurations. By continuously monitoring the system load and the applications' security demands, the service chooses which protection mechanisms to enable. The configurations are also optimized in terms of performance and power consumption. This optimization is based on data that has to be provided by the system provider. The information should also include prioritization for cases in which performance and low energy demand prove to be conflicting goals.

4.2 DRConf Kernel Module Design

This thesis proposes to utilize dynamic reconfiguration of hardware-vulnerability mitigations to adapt the system configuration at run time to the needs of different applications. Thereby, the system's overall performance and power consumption can be improved as protection mechanisms, which naturally induce overhead, are only active if required. Additionally, the choice of mechanism or mitigation implementation can be optimized in terms of performance and energy demand. The design of the user-space–based service required for utilizing the dynamic reconfigurations is introduced in Section 4.1. This section focuses on the design of the kernel module that provides the service with the dynamic control of Spectre and Meltdown mitigations, which can only be controlled at boot time in the mainline Linux kernel.

Figure 4.3 illustrated the kernel module's integration into the system. As an extension of the Linux kernel, the module is shown as part of the kernel. The module collects information on the underlying hardware at boot-time from the kernel initialization routines (ⓑ). If it receives reconfiguration directives from the service, the module issues the corresponding changes in kernel settings. The collected information and the current mitigation settings are provided to the service via `sysfs` entries, which also serve as an interface for reconfigurations (ⓐ). In the following, details on the design of the kernel module itself and its integration into the kernel are presented. Additionally, an overview of the kernel code extensions required for the dynamic control of the Spectre and Meltdown mitigations is given.

4.2.1 Dynamic Mitigations Configuration

The kernel module extends kernel functionality by making mitigations dynamically reconfigurable that can only be controlled at boot-time in the mainline kernel. As such, the kernel module provides the service with dynamic control of the Spectre and Meltdown mitigations.

At boot-time, the kernel module collects information from the kernel on which of these vulnerabilities are present in the underlying hardware and which mitigation implementations are best suited. For example, retpolines as part of the mitigation for Spectre variant two are available as a generic software implementation and hardware-specific implementation. With some processors, retpolines do not have to be used at all, as mitigations have been implemented directly in hardware, for example, enhanced Indirect Branch Restricted Speculation (eIBRS). In general, hardware-specific solutions induce less overhead but only guarantee protection on the respective hardware. Thus, if available, they should be part of the mitigation configuration to minimize the overhead in terms of performance and power consumption. In addition to selecting the mitigation implementations, the kernel module prompts the actual reconfigurations of said mitigations.

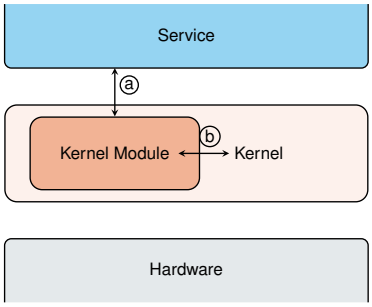


Figure 4.3 – The kernel modules's integration into the run-time environment. The module receives reconfiguration instructions from the service and reports the mitigations' internal configuration ⓐ. The module collects information on Spectre and Meltdown mitigations from the kernel and controls their configuration ⓑ. Interactions not related to dynamic reconfigurations are omitted.

With the selection of mitigation implementations best suited to the system's underlying hardware and the active reconfiguration of these mitigation implementations, the two main functionalities of the kernel module have been presented. In the following, extensions to the kernel code basis are outlined, which are required to allow the kernel module control of mitigations that were previously only available for configuration at boot-time.

4.2.2 Kernel Extension

Introducing dynamic control of mitigations for which the mainline kernel only offers reconfiguration at boot time requires changes and extensions to the kernel. With this thesis, support for the dynamic control of the Spectre and Meltdown mitigations was incorporated into the 5.14 Linux kernel.

Spectre mitigations in the kernel are composed of single instructions, such as memory barriers, or routines, such as setting processor registers to enable hardware mechanism. These functions are integrated into different kernel section. Most can be found in the kernel-entry and -exit code. Thus, introducing reconfigurability requires changes to these code regions to either omit the mitigations or execute them, depending on the current configuration. This thesis proposes the use of code patching as the reconfiguration mechanism. At first, conditional jumps were considered, but their use proved problematic. Conditional jumps and, therefore, conditional branches can again be exploited by attacks. Mistraining the branch predictor could lead to speculatively omitting mitigations as they are "hidden" in if-else constructs. This could be avoided by using memory barriers. This would, in turn, introduce additional overhead in some highly performance-sensitive code sections, such as the switch between kernel and user space. With jump labels, the kernel offers an SMP-compatible code-patching mechanism. The use of jump labels is described in detail in Section 5.3.

Changes to the kernel entry and exit are also required for the Meltdown mitigation KPTI. Here, the switch of page tables is omitted if KPTI is disabled. This is again realized with code patching. Additionally, dynamic control of KPTI requires additional functionality. The design of this thesis proposes to implement reconfigurable KPTI by suppressing the switch of page tables that usually occurs when changing from kernel to user space and back. Instead, the kernel page tables are also used by user-space processes. This allows skipping costly switch routines. But, as a security measure of the mainline kernel, kernel page tables cannot be simply used in user space. Thus, kernel page tables have to be "unpoisoned" before KPTI can be disabled. Unpoisoning encompasses making the user-space mappings executable. This concept of deactivating KPTI at run time was proposed in the original KAISER patch-set [28] and was implemented for the 5.14 Linux kernel as part of this thesis.

The use of code patching to omit or execute the Spectre and Meltdown mitigations in kernel-entry and -exit paths is a significant part of the kernel extension required for the dynamic control of said mitigations. Additionally, the functionality of making kernel page tables usable in user space is introduced. With these two extensions, reconfiguration of hardware-vulnerability mitigations as proposed by this thesis can be realized.

4.2.3 Module Interface

While the kernel module is responsible for controlling the Spectre and Meltdown mitigation implementations in the kernel, the decisions on when to reconfigure the respective mitigations are made by the service. As the interface for the service, the kernel module implements `sysfs` entries. The entries are only provided per hardware vulnerability and corresponding attack. Thus, the module currently exposes one entry for Spectre and one for Meltdown to the service. This level of abstraction is chosen as the kernel module already optimizes the mitigation composition by preferring hardware implementations if available. Additional entries could be added to represent the different strategies for the use of IBPB and STIBP, which can also only be activated conditionally per process. With the help of the `sysfs` entries, the module also provides details on the mitigations. For example, for the Spectre mitigations, the module passes the information on which implementations comprise the mitigations. By utilizing the `sysfs`, the module provides an interface that is in line with the control of mitigations provided by the kernel itself. Thus, it allows for consistent reconfiguration of all hardware-vulnerability mitigation by the service.

Summary This section presented the design of the **DRConf** kernel module. While the service coordinates and issues mitigation reconfigurations, the kernel module introduces dynamic control of Spectre and Meltdown mitigations. The mainline Linux kernel only provides control of these mitigations at boot time. Thus, the kernel module is required. Control of said protection mechanisms is provided to the service via the `sysfs`. Based on processor characteristics, the module chooses the optimal combination of implementations that form the Spectre mitigation at boot time. If available, hardware mechanisms are preferred as they typically induce less overhead than software constructs. Making Spectre and Meltdown mitigations reconfigurable at run time is not possible without the implementation of kernel extensions and kernel patches. Code sections where mitigations are located have to be patched with constructs that allow omitting mitigations if required. Therefore, jump-label code-patching directives are added to the kernel source code. When KPTI is disabled, the kernel page tables are also used in user space, which requires modification of user-space–mappings permissions. The functionality of adapting the kernel page tables is also provided through kernel extensions.

4.3 Summary

This chapter outlined the design of the service and kernel module required for introducing dynamically reconfigurable hardware-vulnerability mitigations in the Linux kernel. The service is the central actor and is responsible for determining mitigations configurations according to security requirements. Also, the service realizes the optimization of the configurations in terms of the system's performance and energy demand. The service's reconfiguration decisions are based on information on the underlying hardware and its vulnerabilities alongside information provided by the kernel module designed as part of this thesis. The kernel module allows for run-time control of mitigations that are only controllable at boot time in the mainline kernel. Changing the reconfigurability of the respective mitigations also required kernel extensions, in the form of code-patching in kernel-entry and -exit paths, as well as additional functionality in the page-table management.

5 IMPLEMENTATION

This chapter presents the implementation of **D**ynamic hardware-vulnerability mitigation **ReC**onfiguration (**DRCONF**) for the Linux kernel with its service, kernel module, and required kernel extensions. With this implementation, dynamic reconfigurations of hardware-vulnerability mitigations in the Linux kernel are made available and can be utilized to adapt the system's configuration to application demands while optimizing the system's performance and energy demand. The implementation is based on the design detailed in Chapter 4. The chapter is structured as follows: Section 5.1 provides an overview of the framework implemented to function as the service according to its design introduced in Section 4.1. The description of the framework's implementation is followed by the description of the realization of the kernel module in Section 5.2. This section focuses on how the kernel module provides the service with run-time control of mitigations, which can only be reconfigured at boot time in the mainline kernel. Section 5.3 details the kernel extensions required for dynamically controlling mitigation execution. All implementations are based on the Linux mainline kernel version 5.14[2], which was newly released in August of 2021 at the start of this thesis. Some mitigations of hardware vulnerabilities, such as the type of retpoline used on AMD systems and the Core Scheduling interface, have changed since then. These changes can be incorporated as part of future development of the service and kernel module but are not considered in the scope of this thesis.

5.1 DRCONF Service Implementation

Central to utilizing dynamic hardware-vulnerability reconfigurations in the Linux kernel is the user-space service. The service collects information on the system characteristics and underlying hardware, as well as information on the workload deployed on the system. Based on the collected information, the service is responsible for managing the system's hardware-vulnerability mitigations to ensure that all processes are protected accordingly. Usually, protection can be provided via different configurations. In this case, the service is tasked with optimizing reconfigurations in terms of performance and energy demand, whereas the system operator determines which of the two is prioritized. Section 4.1 provides more detail on the service's design. As part of this thesis, a basic framework serving as the service was implemented. This framework demonstrates the service's functionality and is used for evaluating dynamic hardware-vulnerability reconfigurations in terms of the effect on performance and energy demand. The following section presents the implementation of the framework with a focus on the incorporated scripts and binaries for reconfiguring mitigations and providing protection.

[2] https://github.com/torvalds/linux/tree/v5.14

5.1.1 Framework Infrastructure

To showcase the service's functionality, a framework was developed that deploys applications on the target system and sets hardware-vulnerability configurations according to provided protection demands. This framework encompasses several Bash and Python scripts as well as programs developed in C. Processes are started with the help of a corresponding script. The caller also passes information on whether mitigations for Spectre and Meltdown are required and if process isolation is needed. Should no protection from Spectre or Meltdown attacks be requested, the script reconfigures the system so that the corresponding mitigations are disabled before the application is started. In case of process isolation, the framework can either protect the processes by way of disabling Symmetric Multithreading (SMT) or by utilizing Core Scheduling. The script can disable SMT and change the Spectre- and Meltdown-mitigations configuration before applications are started. As Core Scheduling requires the process ids (PIDs) for assigning them scheduling cookies, the application must be in execution so that the parent PID is available. With the framework developed for this thesis, Core Scheduling is activated immediately after deploying processes. All of the reconfigurations mentioned above can only be performed with root privileges. This is due to the fact that all affect the system's state and, thus, all applications deployed on the system, albeit to different extents. Therefore, all scripts and binaries of the framework are run with the `sudo` command.

5.1.2 Dynamic Security Configurations

To ensure all applications are run with the required protection, the framework representing the service must dynamically reconfigure the hardware-vulnerability mitigations. As described above, the framework is passed the demands of the application and adapts the system's configuration accordingly. Dynamic control of the Spectre and Meltdown mitigation as well as dynamic control of process isolation for mitigating cross–Hyper-Thread attacks is provided by the framework.

The Spectre and Meltdown mitigations are controlled via the respective `sysfs` entries. Disabling SMT as a mechanism to provide process isolation can also be controlled in the `sysfs`. The second possibility of providing process isolation and thereby mitigating cross-HT attacks is Core Scheduling. Core Scheduling, in contrast to the previously mentioned mitigation configurations, is not controlled via `sysfs`. Instead, Core Scheduling is activated for a group of processes with help of the `prctl` syscall. The syscall has to be provided with the process id of the process for which a scheduling cookie is to be set. To utilize this feature, the framework provides a custom `coresched` binary and script for collecting all process ids of an application. The process ids are gathered either via a combination of the `getpid` command and the `proc`-filesystem entry for the parent id of the deployed

```c
void core_schedule_pids(pid_t *pids, size_t num_pids) {

  prctl(PR_SCHED_CORE, PR_SCHED_CORE_CREATE, getpid(), 0, 0);

  for (int i = 0; i < num_pids; i++)
    prctl(PR_SCHED_CORE, PR_SCHED_CORE_SHARE_TO, pid[i], 0, 0);

}
```

Figure 5.1 – System calls required for Core Scheduling a group of `num_pids` processes using their process ids (`pid[i]`). The implementation is based on version 5.14 of the Linux mainline kernel. Error handling is omitted for reasons of clarity.

application. The ids are then passed to the binary. Only the processes already running have to be explicitly assigned a scheduling cookie. Any newly created child automatically inherits its parent's cookie. To enable Core Scheduling for the selected processes, the `PR_SCHED_CORE` interface of the `prctl` system call is used. Figure 5.1 lists the routine for sharing a scheduling cookie between all process ids stored in the `pids` array. First, a cookie is created for the current process that is executing the `coresched` binary. This is done by sending the `PR_SCHED_CORE_CREATE` command with the current process id obtained by `getpid` (Figure 5.1, line 3). The created cookie is then assigned to all ids of `pids` with the help of the `PR_SCHED_CORE_SHARE_TO` command (Figure 5.1, lines 5-6). After all of the cookies have been set, the `coresched` program terminates its execution. This leaves only the processes of `pids` with a shared scheduling cookie. With that, the processes are isolated from other applications, and only processes of this group are allowed to be scheduled on sibling cores. The detour of first creating a cookie for the `coresched` process is necessary due to the limited commands provided by the `PR_SCHED_CORE` interface. As of kernel version 5.14, only the cookie of the current process can be shared with another process. Assigning the cookie of process a directly to process b is not supported.

Summary Dynamically controlling hardware-vulnerability mitigations is realized by way of a framework that provides the service functionality outlined in Section 4.1. The framework is comprised of bash scripts for starting applications and adapting the mitigation configuration according to the protection requirements passed by the caller. Reconfigurations are done via changing the respective `sysfs` entries for Spectre, Meltdown, and SMT control and by invoking custom binaries to utilize Core Scheduling.

5.2 DRConf Kernel Module Implementation

Reconfiguration of hardware-vulnerability mitigations results in changes to kernel data structures and kernel functionality. Interfaces for changes with regard to providing process isolation, such as disabling SMT and Core Scheduling, are provided by the mainline kernel out-of-the-box. But the mainline kernel does not provide support for dynamically reconfiguring all mitigations. As such, the Spectre and Meltdown mitigations can only be controlled at boot time. To enable the framework to issue reconfigurations that encompass these mitigations, a kernel module providing their dynamic control was implemented as part of this thesis. The underlying design for the module's implementation is outlined in Section 4.2. The kernel module manages the different mitigation implementations and controls which combination of mitigations is best suited to the underlying system. On a reconfiguration command of the framework, the module en- or disables the corresponding mitigation implementations. As an interface for the framework to control the mitigation configuration, the module provides `sysfs` entries. The implementation of the kernel module is described in more detail in the following.

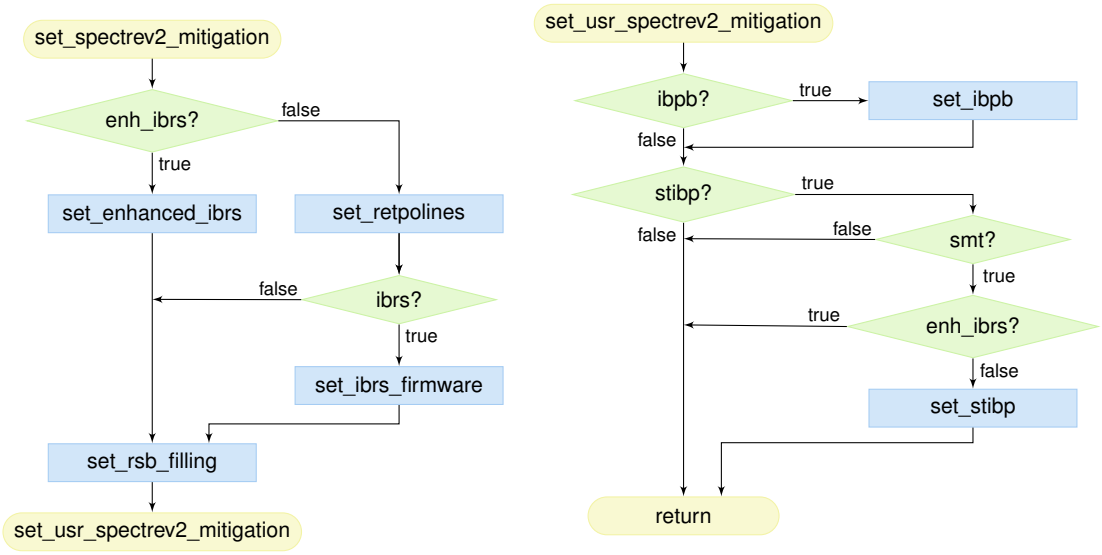
5.2.1 Dynamic Mitigation Reconfiguration

The kernel module provides the framework with support for dynamically reconfiguring the Spectre and Meltdown mitigations. In the mainline kernel, both mitigations can only be controlled at boot time and cannot be dis- or enabled afterward. To introduce this functionality, the module first determines the mitigation implementations best suited to the underlying processor. This is done analogous to the kernel's routine for identifying mitigations compositions at boot time. Based on this information, the module can select the relevant mitigation implementations and control whether

they are active or inactive. The different mitigation mechanisms are controlled differently, depending on how their reconfigurability was realized. Some mechanisms can be dis- or enabled with the help of code patching so that the control flow omits the mitigation execution when the configuration does not include protection from attacks of the corresponding vulnerability. The kernel module controls this with the help of static keys [59]. The implementation of control-flow manipulation is presented in more detail in Section 5.3. In the case of KPTI as Meltdown mitigation, additional steps to adapt the kernel page-table to dynamic control must be taken. The different steps for dynamic reconfiguration of both Spectre and Meltdown mitigations are described in detail in the following.

Controlling Spectre Mitigations The Spectre hardware vulnerability can be mitigated via different combinations of mitigation implementations. Which mechanisms are suitable to mitigate Spectre on the given system is first determined at boot time during kernel checks (see `check_bugs` routine [43]). The underlying control flow of the kernel routine also forms the basis for the dynamic reconfiguration. With the Spectre vulnerability, two variants can be distinguished. The Spectre hardware vulnerability in general and the different mechanisms used for mitigation in the Linux kernel are described in detail in Section 2.1.1.

For Spectre variant 1, the kernel provides protection with the help of load fence instructions (`lfence`), which are added to kernel entry and exit paths. `lfences` are added to kernel entry paths if one of two conditions is met. Either if the system allows its applications to write the `FS` and the `GS` base registers or if Supervisor Mode Access Prevention (SMAP) does not provide protection from speculation. The load fence instructions are also required on kernel exit if the system is susceptible to the SWAPGS bug and KPTI is not in use.



(a) Flow chart for the selection of kernel mitigations for Spectre variant 2.

(b) Flow chart for the selection of user program mitigations for Spectre variant 2.

Figure 5.2 – Flow charts for selecting the Spectre variant 2 mitigation implementations based on the availability of hardware features.

Attacks utilizing the Spectre–variant-2 vulnerability can be categorized in terms of whether the kernel, other user processes, or a virtualized guest is the target. The first category of attacks can be mitigated by unconditionally filling the Return Stack Buffer (RSB) during context switches in combination with either the use of eIBRS or retpolines. These mechanisms are highly hardware-specific. Enhanced IBRS is a feature available on some Intel processors and is controlled by writing the corresponding Model-Specific Registers (MSRs). Also, a retpoline implementation optimized for AMD systems exists. Other processors are required to use the generic retpoline implementation. If the system cannot be protected with the use of enhanced IBRS, but IBRS is available in general, speculation around firmware calls is restricted via this feature. Both retpolines and firmware-call protection is controlled via patching in or out instructions to redirect the control flow. The flow chart for deciding on the combination of mechanisms for preventing attacks on the kernel from user space using Spectre variant 2 attacks is illustrated in Figure 5.2a.

Figure 5.2b shows the flow chart of how the combination of mechanisms for protecting user processes from one another are derived. The Linux kernel offers IBPB as protection for processes located in user space, in general, and STIBP as protection for Hyper-Threads of the same physical cores. If available, IBPB is activated by default and can be controlled by way of code patching. The use of STIBP is controlled via writing the corresponding MSR and depends on whether SMT and enhanced IBRS are available. If the processor does not support hyperthreading, the attack scenario requiring STIBP is not possible. If enhanced IBRS is active, it also protects Hyper-Threads from one another, making STIBP superfluous. The remaining category of Spectre variant 2 attacks, a virtualized guest being attacked by one another guest, does not require additional protection mechanisms. The mechanisms selected as part of the mitigations for the first two attack scenarios are suitable to mitigate guest-to-guest attacks as well. All mitigation implementations are determined on the basis of the flow charts of Figure 5.2. The combination of the different implementations forms the system's overall Spectre mitigation, which can then be controlled by the framework. The kernel module en- or disables the different implementations as described above.

Controlling the Meltdown Mitigation The Meltdown hardware vulnerability is mitigated on susceptible systems with the use of Kernel Page Table Isolation (KPTI). The vulnerability and how KPTI mitigates such attacks is described in detail in Section 2.1.2. In the mainline Linux kernel, KPTI can only be en- or disabled at boot time, just as is the case with the Spectre mitigations. Allow for its dynamic reconfigurations requires additional functionality provided to the framework by the kernel module. The actual mitigation of switching between user-space and kernel page-tables has to be en- and disabled. To make the kernel page tables available in user space, the tables have to be modified on KPTI reconfiguration. KPTI is controlled at run time by redirecting the control flow to either execute switching the page tables or omit the corresponding instructions. The module is required to set flags and static keys according to the configuration set by the framework through the corresponding `sysfs` entry. If the system is vulnerable to Meltdown-based attacks, the kernel manages two sets of page tables instead of one. One is used in user space and only contains the kernel data required for kernel entry and exit. The second set encompasses the complete mappings of both kernel and user space and is only available for use in the kernel. On changing from user space to the kernel, the tables are switched by writing the address of the page tables to the Page Directory Base Register (PDBR), which is also called CR3 register. When disabling KPTI at run time, this manipulation of the CR3 register is skipped. This leaves the kernel page tables, which include mappings of both kernel and user-space, to be used in user mode as well. This approach of making KPTI dynamically controllable is based on the original `KAISER` patch [28] that aimed to introduce the page-table isolation to the Linux kernel. The kernel module initiates the changes to

the control flow to either include or omit CR3 manipulation by setting a corresponding flag and static key. Before the kernel page tables can actually be used in user mode, the user space mappings need to be "unpoisoned" first. Unpoisoning, here, refers to marking the mappings as executable. As a security measure, to detect the unintended use of the kernel page tables by user processes, the user-space mappings are set to be non-executable. This has to be adapted when dynamically disabling KPTI and reverted once the isolation is enabled again. The unpoisoning and poisoning of the mappings is realized by iterating over all entries of a process' Page Global Directory (PGD) and setting or removing the NX bit of all entries that map user space. This approach was also proposed in the `KAISER` patch but was not included in the mainline kernel.

Dynamically reconfiguring KPTI and manipulating the PGD entries at run time requires protection from race conditions and inconsistent system states. As proposed by Dave Hansen in his KAISER implementation [28], this implementation makes use of the kernel's `stop_machine` mechanism. With the `stop_machine` function, all threads of all CPUs are called into the kernel and run the function passed to the `stop_machine` call [44]. Figure 5.4 lists the functions implemented for changing the KPTI configuration. The listings are shortened and simplified for clarity reasons. An exemplary control flow of a system changing its KPTI configuration is illustrated in Figure 5.3 and references key functions of Figure 5.4. In Figure 5.3 the dynamic reconfiguration of the Meltdown mitigation is triggered by a process running on `CPU 0` as it writes 0 to the respective `sysfs` file (ⓐ). `Sysfs` then calls the corresponding function in the kernel module (`set_meltdown_mitigation`). This causes `CPU 0` to switch into the kernel (Figure 5.4a, line 3). First, a mutex protecting the critical path of changing the KPTI configuration from concurrency issues is locked (ⓑ). As a next step, `stop_machine` called with a reference to the `pti_stop_machine` function passed as argument. Calling `stop_machine` schedules a thread on every CPU, which then disables interrupts and runs the specified function. Thus, `CPU 1-3` also switch from user space to the kernel and proceed to run `pti_stop_machine` (ⓒ). This function ensures that only one thread issues the (un-)poisoning of the kernel page-table while all other CPUs idle. In the given example illustrated in Figure 5.3, the thread running on `CPU 0` runs `poison_pgds` (Figure 5.4b, line 7) and `CPU 1`, `CPU 2`, and `CPU 3` idle (ⓓ and Figure 5.4b, line 12). Once `CPU 0` signals completion of the reconfiguration (Figure 5.4b, line 9), threads on `CPU 1-3` proceed in the control flow and flush their TLB (ⓔ and Figure 5.4b,

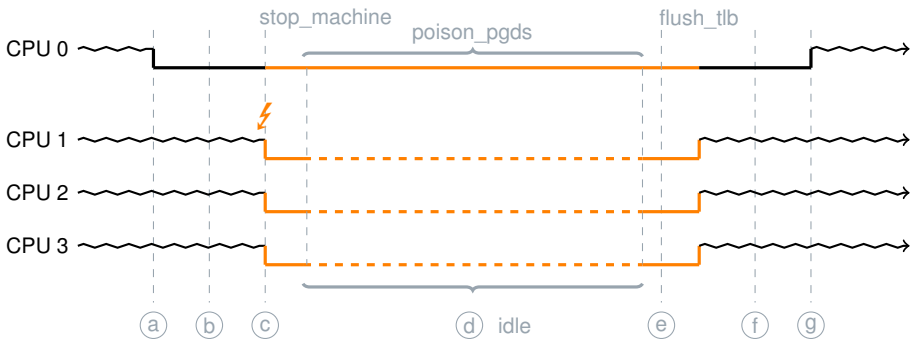


Figure 5.3 – Control-flow example of reconfiguring KPTI. ⇝ indicates execution in user space, → execution in the kernel. ⓐ the process on CPU 0 initiates KPTI reconfiguration and switches into the kernel. ⓑ the mutex to protect against concurrency issues is locked. ⓒ all CPUs are brought into the kernel via the `stop_machine` call of CPU 0. ⓓ CPU 1-3 idle while CPU 0 adapts the kernel page-table. ⓔ all CPUs flush their TLB and CPU 1-3 resume execution. ⓕ the mutex is unlocked. ⓖ the reconfiguration is completed and CPU 0 returns.

line 13). These threads then resume their execution prior to being interrupted and return to user space. CPU 0 also runs flush_tlb instruction. Instead of directly returning to user space, the thread sets the flag and static key according to the new configuration (Figure 5.4a, line 9). Then, the mutex protecting the KPTI reconfiguration (ⓕ and Figure 5.4a, line 11) is unlocked and CPU 0 returns to user space (ⓖ). With the return, the KPTI reconfiguration, consisting of changing the kernel-entry and -exit control-flow to omit or execute the page-table switching and adapting the user-space mappings accordingly, is completed.

5.2.2 Mitigation Selection Interface

The kernel module enables the framework to reconfigure the Spectre and Meltdown mitigations dynamically. To do so, the module implements functionality for en- and disabling the execution of mitigating instructions. In the case of the Spectre vulnerability, the module also determines the combination of mitigation implementations that are best suited for the system's underlying processor and makes this combination available for reconfiguration as the Spectre mitigation. The sysfs is used as the interface for the framework. By reading and writing the respective entries, the framework passes the commands for configuring mitigations to the kernel module. On module initialization, a sysfs directory is created, represented by a kobject internally. If the processor is susceptible to attacks exploiting Spectre or Meltdown vulnerabilities, corresponding kobject attributes and sysfs entries are created. Figure 5.5 lists an excerpt of the directory tree of the kernel sysfs subdirectory. Each entry can be written and read with the help of the functions defined in the respective kobject attributes. When reading,

```
/sys/kernel
└── drconf
    ├── meltdown_mitigation
    └── spectre_mitigation
```

Figure 5.5 – Tree illustration of the sysfs subdirectory for the **DRConf** kernel module and the entries for reconfiguring the Spectre and Meltdown mitigations.

```c
static int first_stop_machine;

void set_meltdown_mitigation(bool enable) {
  mutex_lock(&pti_ctrl_mutex); // b

  first_stop_machine = STOP_MACHINE_INIT;
  stop_machine(pti_stop_machine, enable,
    ↪ cpu_online_mask);  // c

  pti_active = enable;

  mutex_unlock(&pti_ctrl_mutex); // f
}
```

(a) Implementation of KPTI reconfiguration. A mutex is utilized to prevent concurrency issues. The changes to the user-space mappings are initiated via stop_machine, which calls all CPUs into the kernel. All threads then run the pti_stop_machine function. After the kernel page tables are readied for use in user space, the flag and static key are set, and the mutex is unlocked.

```c
void pti_stop_machine(bool enable) {
  int first, signal;
  first = cmpxchg(&signal, STOP_MACHINE_INIT,
    ↪ STOP_MACHINE_START);

  if (first == STOP_MACHINE_INIT) {
    poison_pgds(enable);
    signal = STOP_MACHINE_END;
  } else {
    while (signal != STOP_MACHINE_END)
      cpu_relax(); // d
  }
  flush_tlb(); // e
}
```

(b) This function is run by all CPUs. Atomically setting the first variable ensures that only one thread calls poison_pgds while all others idle. Thus, the page tables are not used while they are being manipulated. Before returning, the TLBs are flushed so that the changes to the user-space mappings are visible. The use of atomic functions is omitted for clarity reasons.

Figure 5.4 – Implementation of the set_meltdown_mitigation and pti_stop_machine. The functions are simplified for clarity reasons. The commented letters correspond to Figure 5.3.

for example, the `spectre_mitigation` file, `sysfs` calls the `read` function listed in the respective attribute's definition. This function then prints 0 or 1, depending on whether Spectre mitigations are currently active or not. Additionally, information on the chosen combination of mitigation implementations is provided. The `store` functions defined per attribute are called if the `sysfs` files are written. Valid input is 0 or 1 to disable or enable the mitigation. This triggers the module's implementation of setting flags and static keys accordingly as well as initiating the required changes to the execution bit of the kernel page tables' user mappings.

Summary In the Linux mainline kernel, not all mitigations can be controlled at run time. As such, the Spectre and Meltdown mitigations are only available for configuration at boot time. To allow for their dynamic reconfiguration, a kernel module managing the mitigation control and providing an interface for the framework was implemented. Changes to the control flow to omit code sections in case of disabled mitigations are initiated by the module setting respective flags and static keys. In the case of KPTI, additional functionality for making the kernel page-table available in user space is provided. For the Spectre mitigations, the kernel module determines the optimal combination of mitigation implementation based on the system's processor.

5.3 Kernel Extension

Utilizing dynamic reconfigurability of the hardware-vulnerability mitigations in the Linux kernel requires the combination of user-space service and kernel module. The service provides the functionality of determining the system's security requirements and changing configurations accordingly. The kernel module introduces support for dynamically controlling the kernel's Spectre and Meltdown mitigations. While the module can issue some changes to the control flow through kernel-entry and -exit paths where most mitigations are located, adding a module cannot fully introduce the dynamic reconfiguration of these mitigations. As outlined in Section 4.2, additional changes to the kernel code basis are necessary. Functionality to allow redirecting the control flow to either omit or run mitigations has to be added directly in central kernel functions such as the switch between kernel and user space. These kernel extensions encompass changing the mechanism used for patching mitigations into the loaded kernel and adapting the kernel page-table management so that the kernel copy is usable in user space. The following sections detail these extensions and their use for realizing dynamically reconfigurable Spectre and Meltdown mitigations.

5.3.1 Dynamic Mitigation Execution in Kernel–User-Space Paths

Mitigations for the Spectre and Meltdown hardware vulnerabilities are located in context switches as well as the kernel-entry and -exit paths. These sections are typically executed frequently and are, thus, highly performance-critical. Therefore, only as much overhead as necessary should be included in these paths. One way of adding mitigations in these code paths with minimal overhead is by using code patching of the loaded kernel image. This is utilized in the mainline Linux kernel. There, both Spectre and Meltdown mitigations are controlled via kernel command-line and set at boot time. Based on the parameters provided, the kernel patches its code to either include the mitigations or omit the corresponding instructions. This code patching is realized with the alternative functionality provided by the kernel.

Alternatives Alternatives provide one way on code modifications in a loaded kernel image but can only be run before Symmetric Multiprocessing is enabled. Thus, the early evaluation and setting of the mitigations. Alternatives are used with via C-function or assembly macros:

```
alternative ( old_instructions, new_instructions, feature )
```

With alternatives, the developer defines old and new instructions that are patched in instead of the alternative function depending on whether a processor feature is present. Processor features are defined via x86 feature flags [68]. These flags detail the CPU's traits and can be either predefined by the processor itself or be newly defined by the kernel during the boot process. Alternatives are generally used to optimize the kernel at boot time for the system it is deployed on. During kernel compilation, the `old_instructions` are compiled in while the alternative instructions are located in a dedicated ELF section. If a `feature` is present the `new_instructions` is patched into the loaded kernel image, replacing `old_instructions`. This happens early in the boot process when `apply_alternatives` is called. Alternatives are also available for assembly code via the ALTERNATIVE macro.

In the case of the control of hardware-vulnerability mitigation in the mainline kernel, new features are defined in the `check_bugs` routine. This function analyzes the system's vulnerabilities and selects the mitigation implementations best suited to the underlying hardware. For the Spectre mitigations, alternative instructions are defined in the kernel-entry and -exit paths to patch in load fence instruction after `swapgs` directives as well as patching in instructions for filling the RSB. If retpolines are selected as Spectre variant 2 mitigation, the corresponding code constructs are also patched into the kernel image via alternatives. Additionally, the IBRS-based speculation restriction around firmware calls is patched in. Alternatives are also used to patch the code for switching the page tables on kernel entry and exit to realize KPTI. The existing alternative instructions and corresponding code-patching mechanism are not suitable for implementing dynamic reconfiguration of hardware-vulnerability mitigations in the Linux kernel. This is mainly due to the fact that they are designed for use without active Symmetric Multiprocessing (SMP) and lack the otherwise required synchronization mechanisms.

With static keys, the kernel offers a different code patching procedure that is compatible with SMP and can be used during the regular operation. But static keys are not without challenges either when used for implementing dynamic control of Spectre and Meltdown mitigations. As the code-patching mechanism of static keys is not fully atomic, problems can arise when code is patched that might be executed during interrupts, especially during Non-Maskable Interrupts (NMIs) and when hitting breakpoints. These critical code sections are typically found in the kernel-entry and -exit paths where, coincidentally, most of the mitigations are located. Thus, a combination of static keys and conditional jumps is used to evaluate the dynamic reconfigurability of hardware-vulnerability mitigations in this thesis.

Static Keys While alternative instructions are not feasible for use during run time, a different approach to patching the live kernel can be utilized: static keys [59]. Static keys in the Linux kernel are typically used for implementing tracepoints with minimal overhead. But they are also already chosen for implementing the conditional activation of IBPB and STIBP. In contrast to alternatives, static keys are used for switching between jump and no-operation instructions in the kernel image to omit code sequences if required. This is used for defining likely and unlikely branches in the kernel code. If a likely branch is defined and the corresponding static key is enabled, the code of the branch is executed normally. Once the static key is disabled, a jump instruction is

patched in in place of the branch definition, which causes the control flow to skip the instruction of the likely branch. The use of static keys, thus, adds jump labels and switches between a nop and an unconditional `jmp` to the jump label if the state of the respective static key is changed. The underlying code-patching mechanism is also called jump-label patching. This feature can be utilized to omit or execute mitigations depending on the configuration of hardware-vulnerability mitigations.

As a first step to realizing dynamic control of the Spectre and Meltdown mitigations, the alternative instructions for patching the mitigations at boot time are replaced by constructs utilizing static keys. Most mitigation routines are located in kernel–user-space paths, which are implemented in assembly. The mainline kernel only provides static branch implementations for C code. Therefore, as part of this thesis, STATIC_JUMP assembly macros were reintroduced to the kernel code. Static jumps have been part of earlier kernel versions but were removed with kernel version 5.14. Based on the previous definitions, static-jump implementations were added to the 5.14 kernel. Two variants of static jumps are available: one patches in the jump if the key evaluates to true (STATIC_JUMP_IF_TRUE) while the other patches the jump into the control flow if the key is disabled (STATIC_JUMP_IF_FALSE). Figure 5.6 lists the macro implementing the latter variant. Line 4 shows the jump to the target label located after the code that is only executed if the corresponding key is disabled. The following lines (5 - 10) define the jump table entry with relative references to the jump instruction that is to be patched out (lines 7 - 9). On enabling the static key, the `jmp` is replaced with a special no-operation instruction. This is reverted once the static key is disabled. Figure 5.7 gives an example of the use of static jumps to patch in and out one of the load fence instructions mitigating Spectre variant 1. The kernel source code is listed in Figure 5.7a. The static jump after the `swapgs` instruction causes the control flow to directly jump to the target `.Ldone` if `swapgs_key` is disabled. The corresponding assembly directives in the loaded kernel image are given in Figure 5.7b. Once the key is enabled, the `jmp` in line 4 of Figure 5.7b is replaced with a no-operation instruction, which can be implemented differently, depending on the system architecture. In the case of an x86, 64 bit architecture, the no-operation is realized by `xchg %ax, %ax` as shown in line 4 of Figure 5.7c.

```
1  .macro STATIC_JUMP_IF_FALSE target, key
2    .Lstatic_jump_\@:
3
4    jmp             \target
5    .pushsection __jump_table, "aw"
6    _ASM_ALIGN
7    .long        .Lstatic_jump_\@ - ,
8    .long        \target - .
9    _ASM_PTR     \key + 1 - .
10   .popsection
11
12 .endm
```

Figure 5.6 – Implementation of the static jump macro in assembly. (Re-)Introducing this feature allows the use of the static-key–based code patching in assembly files that implement the switch between kernel and user space.

With static keys and static jumps, the overhead of dynamic reconfiguration of the Spectre and Meltdown mitigation is kept to a minimum. To disable the mitigations, a single unconditional jump per mitigation location is patched into the loaded kernel image. For enabled mitigations, no additional instructions are required. In contrast, conditional jump constructs (load, test, jump) execute at least three additional instructions, irrespective of whether mitigations are enabled or not. Thus, the code-patching mechanism is preferable in regard to overhead in the performance-critical paths. But this comes at the cost of higher overhead for changing the configuration. The jump-label code-patching involves several steps to modify multi-byte instructions: By first patching in an `int3` trap in place of the first byte of the instructions to be replaced, the code patching is made compatible with active SMP. It ensures that the control flow does not reach outdated or only partially updated code. Then, the remaining bytes of the instruction are updated according to the jump-table entry of the corresponding static key. Only when these updates are complete is the breakpoint replaced by the first byte of the new instruction. After each step that modifies code, the CPUs are

```
1 swapgs
2
3 STATIC_JUMP_IF_FALSE
  ↪ .Ldone, swapgs_key
4
5 lfence
6
7.Ldone:
```

(a) The use of the static-jump macro in the kernel source code.

```
1 swapgs
2
3 # swapgs_key == false
4 jmp    0xff81e01199 # line 8
5
6 lfence
7
8
```

(b) Assembly instructions of the loaded kernel image with `swapgs_key` disabled.

```
1 swapgs
2
3 # swapgs_key == true
4 xchg   %ax,%ax
5
6 lfence
7
8
```

(c) Assembly instructions of the loaded kernel image with `swapgs_key` enabled.

Figure 5.7 – Exemplary code for the use of static jumps for code patching. In this example, the load fence instruction, a mitigation for the Spectre variant 1 vulnerability, is executed depending on the state of `swapgs_key`. Figure 5.7b and Figure 5.7c show the generated kernel image assembly directives for disabled and enabled keys, respectively.

synchronized. This is required so that the instruction cache and stream of prefetched instructions are not out-of-date. With the CPU synchronization after the `int3` breakpoint is removed again, the code patching is for the respective static key is complete. The corresponding code is located in `arch/x86/kernel/alternative.c:text_poke_bp_batch`. While the use of `int3` ensures no `stop_machine`, halting the execution on all CPUs is required, it still induces significant overhead with a stack trace of multiple functions and the look-up in the jump table. If changes to the mitigation configuration occur frequently, said overhead might become problematic in terms of reducing the system's overall performance. With the occasional reconfiguration of the Spectre and Meltdown mitigation, this thesis argues that the overhead is acceptable, as it allows for the performance gain of disabling mitigations at run time if they are not required.

There are two scenarios where problems with the jump-label code-patching persist in the current implementation. One is the code patching of the `int3` handler and corresponding kernel-entry paths. As Spectre and Meltdown mitigations are located in kernel-entry and -exit paths, patching in an `int3` here can result in an infinite loop and cause the kernel to panic. Therefore, mitigations in paths taken by the kernel to handle breakpoints, such as the `error_entry` function, require different code-patching mechanisms. Also, the use of static keys in combination with the `stop_machine` mechanisms proves problematic. `stop_machine` is used as part of dynamically controlling KPTI and allows the manipulation of all kernel page-table user mappings with active SMP. Here, synchronization is required to ensure the kernel page tables are not used in user space before all have been unpoisoned. The regular CPU synchronizations required for the jump-label–based code patching conflict with the enforced stalls of all but one CPU during the page table (un-) poisoning. Thus, for the first implementation of the kernel with the loaded kernel module, allowing for the dynamic mitigation reconfiguration, some mitigation implementations are in part controlled by flags and conditional jumps. This allows for a first evaluation of the approach's feasibility.

Conditional Jumps For paths where the use of code patching for reconfiguring Spectre and Meltdown mitigations is not possible with the jump-label code-patching mechanism, conditional jumps are used instead. For example, in the previously mentioned `error_entry` path, a load fence instruction is used to mitigate Spectre variant 1 on kernel entry. As this path is also taken by breakpoint handling, code-patching with the jump-label mechanism is not possible here. Instead, the original alternative instruction is replaced with a conditional jump.

The flags tested for the conditional jumps are defined in the kernel and are, thus, located in the kernel address space. With active KPTI, they can, therefore, only be accessed if the kernel page-table is in use. When the control flow switches from user space to kernel, the `swapgs` and subsequent `lfence` are executed before the page tables are interchanged. With the user-space page-table still in use, the flag cannot be read, crashing the system. This is the case, for example, in the branches of the `error_entry` and `asm_exc_nmi` functions coming from user space. To still allow for the use of conditional jumps, flags are defined as `PER_CPU` variables, and each CPU manages its individual flag. This definition locates the flag in the `.data.percpu` memory section. Variables in this section can be added to user-space page tables' kernel mappings and are then available in user-space page tables as well. Another problem of conditional jump proves to be more existential. Introducing checks and conditional control flow in the kernel-entry and -exit paths offers more attack surfaces for speculation-based side-channel attacks. Thus, by adding conditional jumps, the kernel can be left more vulnerable to attacks exploiting Spectre and Meltdown vulnerabilities. These constructs are therefore only used to allow for the first evaluation of the approach. For introducing dynamically reconfigurable hardware-vulnerability mitigations in productive systems, these jumps should be replaced by code-patching mechanisms that are adapted for use in the critical code paths.

5.3.2 Dynamic Kernel Page Table Isolation Management

The kernel module introduced in Section 5.2 provides support for dynamically reconfiguring KPTI. The functions implemented as part of the module enable the use of the kernel page-table in user space. This allows for the deactivation of KPTI at run time, as the kernel page-table contains both the kernel and user-space mappings. As detailed in Section 5.2, the page tables have to be unpoisoned before the first access from user space to avoid crashing the system. While this unpoisoning marks all current user mappings executable, entries that are newly added to the page table still pass through the kernel's original KPTI functions that do not include dynamic control. These functions automatically mark all new user-space entries as non-executable by setting the `NX` bit, and the system would crash on first access if the kernel page-table is used. Thus, the existing functions for adding entries have to be extended to guarantee the use of kernel page-tables in user space if KPTI is temporarily disabled. In addition, the function `pti_clone_user_shared` is extended for the first implementation. This function is called during KPTI initialization and populates the user-space page-tables with the kernel data needed for entering the kernel from user space. Here, the flags used for conditional jumps are added alongside the CPU_ENTRY_AREA if the kernel module is loaded.

Summary Along with the framework and kernel module, introducing dynamically reconfigurable hardware-vulnerability mitigations requires kernel patches. As the Spectre and Meltdown mitigations can only be controlled at boot time in the mainline kernel, additional functionality is required to enable the conditional execution of said mitigations. This is realized by utilizing the jump-label code-patching mechanism via static keys and static jumps. Static jumps patch jump instructions to defined labels into the code if the corresponding key is set accordingly. With this, the control flow can be manipulated to omit mitigation implementations. In some code paths, this code-patching mechanism can cause kernel panics and needs to be adapted before being deployed in productive systems. In these paths, the dynamic reconfiguration is realized via conditional jumps for realizing the evaluation of the approach.

5.4 Summary

This chapter presented the implementation developed as part of this thesis. The implementation is based on the design outlined in Chapter 4 and encompasses a service framework, the corresponding kernel module, as well as kernel extensions. For the service, a framework was realized that starts applications with the defined mitigation configurations as described in Section 5.1. The framework provides support for isolating threads in the form of dynamically adapting SMT and utilizing Core Scheduling. The first is controlled via the `sysfs` entry provided by the kernel. The latter is realized by way of a custom script and binary that determine all threads and their process ids for an application and set a shared scheduling cookie for those threads. The kernel module implementations are detailed in Section 5.2. The main objective of the kernel module is to enable run-time control of the Meltdown and Spectre mitigations. For both, the module implements `sysfs` entries so that the framework can reconfigure these mitigations in the same way as SMT. Based on information on the system's processor and its vulnerabilities, the module selects the mitigation implementation best suited to the hardware. This combination of mechanisms providing protection is then en- or disabled according to the framework's input. Additionally, the kernel module provides functionality to make the kernel page tables available in user space, which allows for the reconfiguration of KPTI at run time. Section 5.3 introduces the implementation of the kernel code extensions. To allow for the dynamic reconfiguration of Spectre and Meltdown mitigations, code patching mechanisms are utilized. The mainline kernel's alternatives used for patching in mitigations at boot time are replaced by the jump-label code-patching mechanism. With these static jumps, the execution of mitigations can be controlled by en- or disabling static keys. On deactivating a key, the corresponding mitigation is omitted from the control flow. This is achieved by patching in jumps to the end of the mitigation code. As some code paths require additional code-patching functionality, conditional jumps are also used to allow for the first evaluation of the approach proposed in this thesis.

6 EVALUATION

This chapter presents the evaluation of dynamically reconfigurable hardware-vulnerability mitigations in the Linux kernel. The evaluation focuses on determining the price of hardware-vulnerability mitigations in terms of performance and power consumption and how their dynamic reconfiguration can reduce this impact. Therefore, the first measurements determine the mitigations' price on the chosen evaluation platforms. The results are presented in Section 6.3. This is followed by the evaluation of the overhead induced by the dynamic reconfiguration and the corresponding kernel code extensions in Section 6.4. Section 6.5 presents the evaluation of the two process-isolation mechanisms that may be used to mitigate MDS: disabling SMT and Core Scheduling. Section 6.6 showcases the evaluation of dynamic reconfiguration of hardware vulnerabilities in use. The setup for all four evaluations and details on the evaluation platforms are presented in Section 6.1.

6.1 Evaluation Setup

This section introduces the setup used to evaluate dynamic hardware-vulnerability reconfigurations in the Linux kernel. This includes details on the evaluation platforms, the benchmarks, and the tools used for measurements.

Figure 6.1 illustrates the general setup used for all evaluations. The experiments are initiated by the coordinator, a system located in the same local network as the system under test. The coordinator also collects the measurement data from the system under test and configures the mitigations corresponding to the respective evaluation scenario. Details of the different processors serving as systems under test are presented in Section 6.1.1. The benchmark used for this evaluation is the `CPU` test from the sysbench benchmark tool. The benchmark offers detailed statistics on its performance that are used as a basis for the evaluation. Additionally, the perf tool is utilized to collect data from the processor's performance counters. Also deployed on these systems is an NGINX web server used for generating different system background loads. Both the use of the sysbench benchmark and the NGINX web server are further elaborated in Section 6.2. The system under test is powered via a power-monitoring IC, the MCP39F511N. The power monitor also directly provides the coordinator with the power data. To reduce the overhead induced by the coordinator, power measurements are only provided for entire benchmark evaluations and not per execution repetition as with the performance data. To stress the web server and, thus, increase background load, two client systems continuously send requests to the evaluation platform. This is done with the help of the wrk HTTP benchmarking tool. All tools, namely, *perf*, the MCP power monitor, and wrk, are further described in Section 6.1.2.

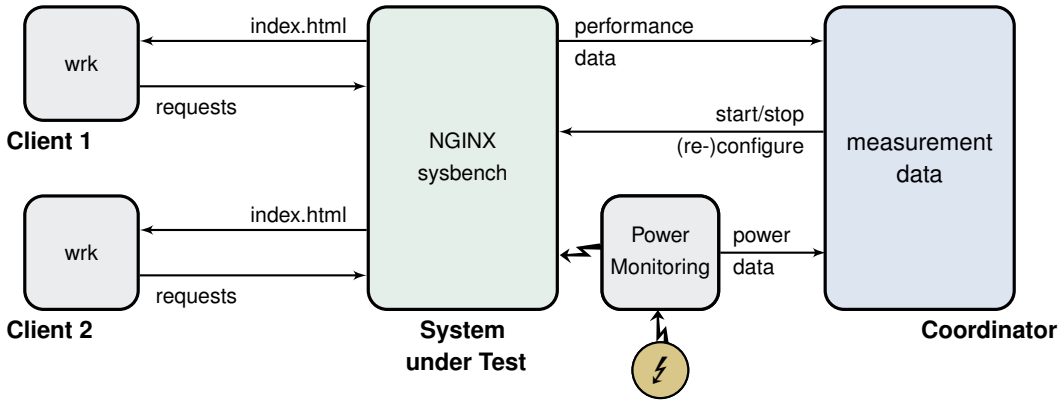


Figure 6.1 – Illustration of the evaluation setup. The coordinator initiates the experiments and collects measurement data. The system under test is powered through a power-monitoring device that directly passes the power data to the coordinator. Two clients continuously send requests to the NGINX web-server deployed on the system under test to generate varying system loads. The evaluations platform's performance is determined via the sybench cpu benchmark.

6.1.1 Platforms

To evaluate the impact of dynamic mitigation reconfigurations with different hardware vulnerabilities and mitigation implementations, three processors are tested. Two of the evaluation platforms are systems with Intel processors, one i5-8400 and one i7-10700K processer. The third system has the AMD Ryzen 7 processor. Table 6.1 lists the respective CPU details of the different evaluation platforms. More detail on the processor characteristics and the present hardware vulnerabilities are given in the following.

Intel Core i5-8400 (2017) The first evaluation platform is a desktop system with an Intel Core i5-8400 processor [15]. The processor has six cores and does not provide Hyper-Threading. Thus, six CPU threads can be used. The system's base frequency is 2.8 GHz. With active DVFS, the frequency ranges between 800 MHz and 4 GHz. Hardware vulnerabilities that affect the processor include Meltdown, Spectre, and MDS. As Meltdown mitigation, KPTI is used. For MDS, the system provides protection via clearing CPU buffers. The Spectre mitigations for this system include the use of load-fence instructions and generic retpolines. The complete configuration of the Spectre mitigation is listed in Table 6.2.

	Intel Core i5-8400	Intel Core i7-10700K	AMD Ryzen 7 5700G
# CPU cores	6	8	8
# CPU threads	6	16	16
Base frequency	2.8 GHz	3.8 GHz	3.8 GHz
Frequency limits	800 MHz / 4 GHz	800 MHz / 5.1 GHz	1.4 GHz / 3.8 GHz

Table 6.1 – Information on the evaluation-platform processors based on [2, 15, 16]

Intel Core i7-10700K (2020) The second evaluation system encompasses the Intel Core i7-10700K processor [16]. This processor has eight physical cores with Hyper-Threading support. Thus, 16 CPU threads are available. The processor runs at a base frequency of 3.8 GHz. With active Dynamic Voltage and Frequency Scaling (DVFS) the frequency can scale from 800 MHz up to 5.1 GHz. The system is not susceptible to the MDS and Meltdown vulnerabilities but is affected by Spectre. To mitigate Spectre, the kernel utilizes a combination of load-fence instructions and hardware mitigations[3]. The full list of protection mechanisms combined for mitigating Spectre attacks is given in Table 6.2.

AMD Ryzen 7 5700G (2021) The third evaluation platform is a desktop system with the AMD Ryzen 7 5700G processor [2]. This processor has eight physical cores and provides 16 CPU threads via SMT. The base frequency is 3.8 GHz, which is also the maximal selectable frequency. The minimum frequency is 1.4 GHz. The system is affected by the Spectre vulnerabilities but is not affected by Meltdown and MDS. The Spectre mitigations include the use of load-fence instructions and retpolines specifically designed for AMD processors[3]. The complete Spectre mitigation configuration is listed in Table 6.2.

			Intel Core i5-8400	Intel Core i7-10700K	AMD Ryzen 7 5700G
Spectre variant 1	kernelswapgs		✓	✓	✓
	userswapgs		✓	✓	✓
Spectre variant 2	RSB filling		✓	✓	✓
	IBRS	enhanced	–	✓	–
		firmware	✓	–	✓
	retpolines	generic	✓	–	–
		AMD	–	–	✓
	IBPB	conditional	✓	✓	✓
		always	–	–	–
	STIBP	conditional	–	–	–
		always	–	–	✓

Table 6.2 – Overview of each evaluation platform's default mitigations for Spectre variant 1 and variant 2. ased on the `lspcu` information.

[3]The hardware-specific mitigations enhanced IBRS and AMD retpolines were found to be vulnerable to speculative attacks and are thus no longer in use from Linux kernel version 5.18 on [6].

6.1.2 Tools

Different tools are used to collect system data that form the basis of the dynamic hardware-vulnerability mitigation reconfigurations evaluation. The Linux *perf* command is utilized for collecting processor data, while the MCP39F511N power-monitoring IC samples the systems power consumption. Additionally, the *mpstat* software is used to determine the system load. The tools and their use in the evaluation are detailed below.

Perf To collect data on performance and kernel events, the Linux *perf* command is used. *Perf* is included in the Linux kernel and utilizes CPU performance counters, tracepoints, and different probes for dynamic tracing to provide users with all information for detailed performance analyses. For the evaluation of dynamically reconfigurable hardware-vulnerability mitigations and their effect on performance and power consumption, the CPU performance counters are read via *perf*. Of interest are events, such as context switches, retired instructions, retired cycles, and the number of system calls. This is used to, for example, calculate the average processor frequency during evaluation runs with active DVFS. This information supplements the statistics provided by the benchmarking tools and serves to allow for a more comprehensive understanding of the system performance.

MCP39F511N Power-Monitoring IC An important factor in analyzing dynamic reconfiguration of hardware-vulnerability mitigations is the effect on the system's power consumption. Therefore, all evaluations analyze the performance as well as the power consumption. For the power measurements, the MCP39F511N power-monitoring IC [34] is used. The MCP39F511N provides real-time measurements of input power. The power adapter of the system under test is connected to the monitor and the power supply is redirected through the IC before it reaches the evaluation system. The measured power data is passed via a serial USB connection. The MCP39F511N continuously provides timestamps (in seconds) and the power consumed in between timestamps (in watts). From this, the energy demand (in joules) for the entire measurement can be derived by integrating over time and power consumption. While the MCP provides high-resolution data, it can only sample the wall power of the entire system and not of individual components, for example, the processor.

mpstat The sysbech `CPU` benchmark is also run with varying background loads when analyzing the impact of cross–Hyper-Thread attack mitigations. The system load is created via clients sending requests to the web server deployed on the system under test. The same number of requests per second can result in different system utilization depending on the processor and its characteristics. To determine the actual CPU utilization and performance, the *mpstat* [24] command is used. *mpstat* is a Linux tool used for collecting CPU statistics. For this evaluation, the command is used to determine the system load generated by the web server prior to the benchmark execution. During the benchmark runs, *mpstat* continuously provides CPU utilization reports. These reports are passed to the coordinator system after benchmark completion.

6.2 Benchmarks

The effect of dynamic reconfiguration of hardware-vulnerability mitigations is analyzed via the performance of the sysbench `CPU` benchmark. The effect of dynamic reconfiguration of hardware-vulnerability mitigations is analyzed via the performance of the sysbench `CPU` benchmark. Sysbench is chosen as it can be seamlessly integrated into the measurement infrastructure. Also, the workload is clearly defined and stable if the benchmark is run with DVFS enabled. Hence, the performance

metric can be compared across runs with different frequency settings. The CPU test of sysbench is used to simulate compute-intensive workloads that might be deployed in data centers. To determine the influence of the overall system load and resource contention, an NGINX web server is utilized to increase the system's CPU utilization. This section outlines the two tools and their use in the evaluation of dynamically reconfiguring mitigations.

Sysbench Benchmark Tool For the evaluation of dynamic hardware-vulnerability mitigation reconfiguration, sysbench [60] is used. Sysbench is a scriptable, LuaJIT-based tool that provides multiple benchmarks. Of those benchmarks, the CPU performance test is used to analyze the impact of different mitigation configurations on the performance and the overall system power consumption. The sysbench CPU benchmark is invoked as follows:

```
sysbench --time=15 --threads=18 --cpu-max-prime=1000 cpu run
```

The processor performance is tested by calculating prime numbers up to a certain value. This value is defined by `--cpu-max-prime=P`. An event is defined as the calculation of all prime numbers up to `P`. The benchmark's run-time is defined using the `--time=S` parameter. Sysbench executes events for `S` seconds and completes the last event. Thus, the exact run time of sysbench may vary and `--time=S` merely defines the limit after which no new events are started. The number of simultaneously running worker threads is determined by `--threads=T`. sysbench provides a summary of its execution. This summary includes the total execution time, total number of events, and events per second as a metric for CPU speed.

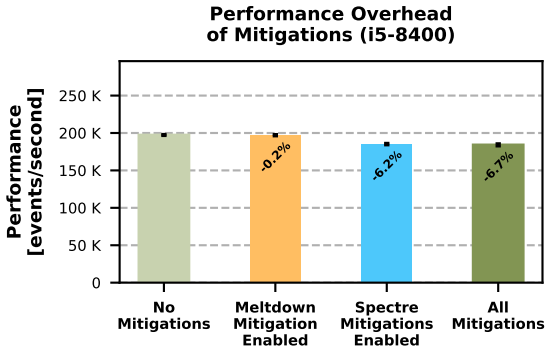
Wrk Benchmarking Tool To create load on the evaluation platforms, wrk [67] is used to stress the NGINX web server [35] on the target system. On each evaluation platform, a default installation of NGINX is installed. The web server is configured to accept 768 connections and communicate via HTTP. The wrk benchmarking tool is deployed on two client systems and used to create load on the systems under test. Wrk opens and keeps a varying number of connections (`-c`) to the NGINX web server using up to 16 threads (`-t`) over a given time span (`-d`). It is invoked as follows:

```
wrk -t18 -c256 -d30s --latency http://target-system-ip/index.html
```

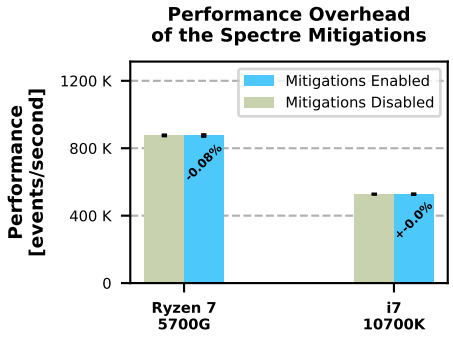
On completing its benchmarking run, wrk prints statistics on latency, requests, and data transfer. The statistics are given for the entire run of all workers and per thread. For the evaluation of the dynamic reconfiguration of hardware-vulnerability mitigations, NGINX and wrk are used to simulate background load on the target systems to analyze the performance of the sysbench benchmark under varying system loads. Therefore, wrk continuously requests a small HTML file (137 B) over the time span of the sysbench CPU benchmark. The number of open connections is increased between runs to simulate different degrees of background load. The NGINX access and error logs are deleted between benchmark runs to create similar conditions for each run.

6.3 Mitigation Costs

Dynamic reconfiguration of hardware-vulnerability mitigations aims to individually tailor the system's configuration to applications' demands and optimize configurations in terms of performance and power consumption. To analyze the benefits of dynamic mitigation control, first, the costs of the mitigations are determined. As the service and kernel module currently allow for run-time control of the Spectre and Meltdown mitigations as well as process isolation to mitigate MDS-based attacks, the overhead is determined for these mitigations. The performance is measured in benchmark events per



(a) Performance of the sysbench CPU benchmark on the i5-8400 system with different mitigation configurations. The annotated percentages show the decrease in performance of the respective configuration compared to the performance of the first measurement with neither Spectre nor Meltdown mitigations.



(b) Performance of the sysbench CPU benchmark on the i7-10700K and AMD Ryzen 7 systems with Spectre mitigation en- and disabled. The annotations show the performance difference of the execution with active mitigations compared to the execution without Spectre mitigations.

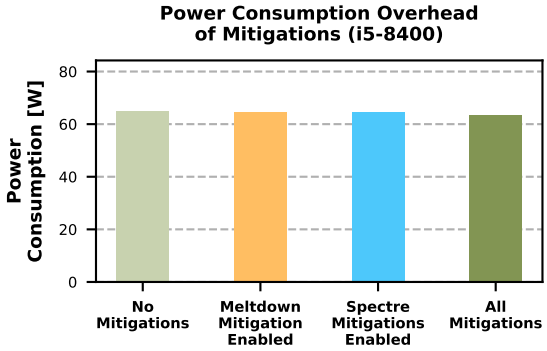
Figure 6.2 – Evaluation results of the Spectre and Meltdown mitigation costs in terms of performance.

second. A Sysbench event encompasses the calculation of all prime numbers up to 1000. Over ten seconds, sysbench executes these events continuously with as many worker threads as CPU threads are available. This routine is repeated 15 times. The system's power consumption is determined using the power monitor and reported as joule per second for the entire experiment. A warm-up phase precedes the experiment to minimize the impact of caching effects and initialization routines. The three evaluation platforms are booted with the 5.14 mainline Linux kernel, and the frequency is set to the maximum frequency with DVFS disabled. In the following, the results of the evaluation are grouped by active mitigations. First, the differences in performance and power consumption with active and inactive Meltdown and Spectre mitigations are presented. This is followed by analyzing the impact of disabling SMT on the benchmark's performance and the system's power consumption.

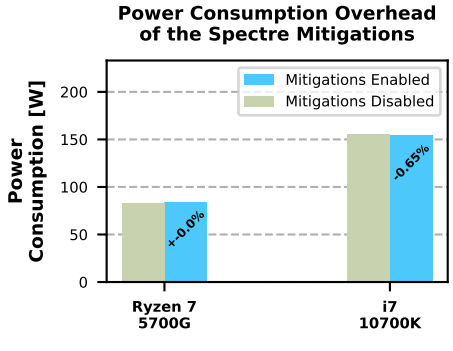
Results for the Spectre and Meltdown Mitigations First, the impact of the Spectre and Meltdown mitigations as implemented on the i5-8400 system is analyzed. On this system, four configurations are tested: Spectre and Meltdown mitigations disabled ("No Mitigations"), only Spectre mitigations enabled, only KPTI as Meltdown mitigation enabled, and mitigations for both hardware vulnerabilities enabled ("All Mitigations"). Figure 6.2a shows the results for these configurations. Compared to the benchmark execution without active mitigations, enabling the Meltdown mitigation results in only minor performance penalties as the events per second drop by 0.2 %. Active Spectre mitigations, in contrast, result in a more noticeable performance decrease by -6.2 %. Analyzing the system's power consumption during the experiment shows no significant differences between the mitigation configurations. The values range from 63.45 W to 64.78 W for "All Mitigations" and "No Mitigations", respectively. Figure 6.3a plots the results of the power consumption evaluation. With stable power consumption over all configurations but differences in the performance, the evaluation shows that the energy efficiency of the different configurations varies. With 3046 events per joule, disabling both Spectre and Meltdown mitigations is the most efficient configuration. In contrast, enabling KPTI and Spectre mitigations lowers the system's energy efficiency by almost 5 %.

On both the AMD Ryzen 7 and i7-10700K system, only the impact of the Spectre mitigations is evaluated as the processors are not vulnerable to Meltdown-based attacks. The benchmark performance with and without Spectre mitigations is illustrated in Figure 6.2b. On the Intel system, the performance is not affected by the Spectre mitigations. For the AMD system, a minimal drop in performance can be observed, though this is well within the standard deviation of the performance measurement results. Therefore, for both systems, the Spectre mitigations do not impose significant overhead in terms of performance. The power consumption of both systems, plotted in Figure 6.3b, mirrors the findings of the performance evaluation. Executing the CPU benchmark with and without Spectre mitigations results in no significant variation in the system's power consumption.

Discussion The evaluation of the costs of the Spectre and Meltdown mitigations shows different results for the three platforms. While the Spectre mitigations significantly impact the i5-8400 system's performance, the other two systems show, if any, only insignificant overhead. This finding is not surprising insofar that the AMD Ryzen 7 and i7-10700K are more recent processors that implement mitigations tailored to the respective hardware. With the AMD-specific retpoline implementation and enhanced IBRS as hardware mitigation replacing retpolines on the Intel CPU, two more efficient mitigation implementations limit the costs drastically. It also has to be noted that the choice of a CPU-heavy benchmark, which does not stress other system components to the same amount, might not reflect the costs of the mitigations in productive systems with more diverse workloads. As Herzog et al. [32] show in their paper, different workload characteristics showcase different degrees to which mitigations impact the processor energy demand. Their evaluation also yields that benchmarks stressing primarily the CPU on systems with active Spectre and Meltdown mitigations result in little to no overhead. At the same time, workloads with high system interaction suffer from more significant overheads that are induced primarily by KPTI. For this evaluation, sysbench was chosen due to its constant workload across different frequencies, which allows comparing results of runs with active DVFS. The CPU-intensive test was used as it most closely represents calculation-heavy workloads that might be deployed in data centers. In future evaluations, more diverse workloads could be used to gain a more comprehensive understanding of the effect dynamic reconfiguration of hardware-vulnerability mitigations can have on productive systems.



(a) The i7-10700K system power-consumption during the sysbench CPU benchmark execution with different mitigation configurations.



(b) The AMD Ryzen 7 and i7-10700K system power-consumption during the sysbench CPU benchmark execution with Spectre mitigations en- and disabled.

Figure 6.3 – Results of the evaluation of the costs of the Spectre and Meltdown mitigations in terms of the system power-consumption.

The evaluation of the i5-8400 processor illustrates the potential of the approach, especially when deployed on systems for which no hardware-specific mitigation implementations exist. The results for the more recent processors show little to no overhead induced by the Spectre mitigations. Evaluations using a more recent kernel version might show a more significant overhead as hardware-specific solutions, such as AMD retpolines and enhanced IBRS, are no longer in use. This is due to the findings of Barberis et al. [6]. They show that these hardware-specific mitigations are still vulnerable to Spectre-based attacks. Thus, further evaluations may present different results for the AMD Ryzen 7 and i7-10700K systems that show a higher impact of the Spectre mitigations.

Results for Disabling SMT The cost of disabling SMT to mitigate MDS-based attacks is analyzed only on the AMD Ryzen 7 and i7-10700K system, as the i5-8400 processor does not support Hyper-Threading. Even though these systems chosen for the evaluation of this thesis are not vulnerable to MDS-based cross-HT attacks, the applied mitigations are identical, and the results can be transferred to vulnerable systems. For the evaluation, the number of parallel sysbench worker-threads is increased to 18. A value greater than the maximum number of CPU threads is chosen to provoke preemption and rescheduling of the threads. Clients continuously send requests via a varying number of open connections to the NGINX web server deployed on the system under test. This creates background load and additional resource contention between sysbench and NGINX worker threads. Figure 6.4 and Figure 6.5 show the results for the AMD and Intel system, respectively. In general, disabling hardware multithreading causes the benchmark performance to decrease. Also, the higher the system load, the more pronounced the drop in performance. For the AMD system, disabling SMT results in performance values that are 23 % below that of enabled SMT if no additional background load is presented. With the maximum number of parallel connections, thus, the highest background load, the performance falls by 44 % when multithreading is disabled. The Intel system mirrors this behavior, though the performance loss due to disabling Hyper-Threading is even more pronounced. Without background load generated by the web server, the events per second drop by 39 %. On 768 parallel connections to the web server, the performance drops by 47 %.

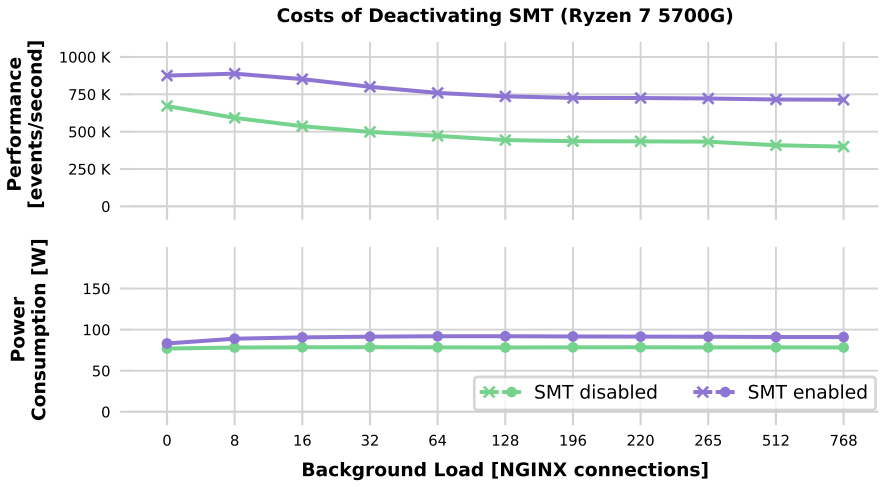


Figure 6.4 – Performance and system power-consumption the sysbench CPU benchmark with SMT en- and disabled on the AMD Ryzen 7 system. The evaluation was run with varying background load generated by parallel connections to the NGINX web sever.

As the two systems differ in maximum frequency, the same amount of parallel connections to the web server results in different CPU utilization. While 768 connections cause an average CPU utilization of 41 % on the AMD system (SMT active), the same number results in 55 % load on the Intel system (SMT active). Disabling hardware multithreading also seems to affect the AMD system more heavily. Disabling SMT while 768 connections are open, the CPU utilization climbs from 41 % to 70 %. On the Intel system, disabling Hyper-Threading with the maximum number of NGINX connections results in 62 % CPU utilization compared to 55 % with active SMT. This difference in system load between the two processors may explain the more pronounced decrease in benchmark performance of the AMD Ryzen 7 when the number of web-server connections is increased.

Analyzing the systems' behavior in regard to power, both show a decrease in the overall power consumption with SMT disabled. The AMD Ryzen 7 shows a relatively constant power consumption during the benchmark execution without hardware multithreading. On active multithreading, the power consumption increases from the run without web-server background-load to the run with the first connections open. Thereafter, the power consumption remains stable. The measured data are plotted in Figure 6.4. The plot also shows an almost constant difference in power consumption between active and disabled SMT executions starting at 8 parallel connections. The constant overhead of disabling SMT shows a reduction in power consumption of approximately 14 %.

The results for the i7-10700K system plotted in the lower figure of Figure 6.5 show that execution without SMT also requires less power. The difference in power consumption between the two configurations shows slightly more variation compared to the AMD system. The results also differ in the first data points with zero NGINX connections. On the Intel system, the difference in power consumption is most pronounced in this point, while it shows the smallest difference on the AMD system. This correlates with differences in performance behavior between the two systems.The Intel system's performance reflects a more significant impact of disabling SMT. With the AMD system, deactivating SMT does not incur as much performance degradation if the background load is low. Considering energy efficiency, the use of SMT proves more efficient on both systems. Hardware multithreading on the AMD Ryzen 7 yields energy efficiency values ranging between 10525 events

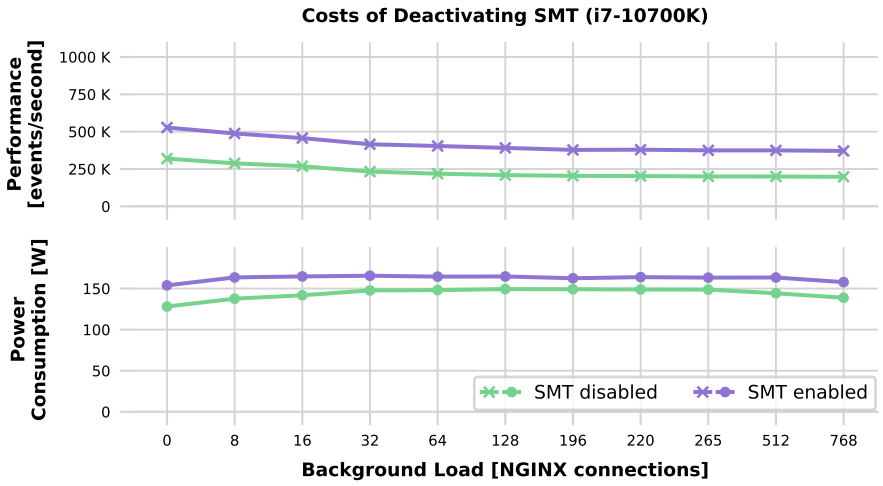


Figure 6.5 – Performance and system power-consumption the the sysbench CPU benchmark with SMT en- and disabled on the i7-10700K system. The evaluation was run with varying background loads generated by parallel connections to the NGINX web server.

per joule (zero NGINX connections) and 7848 events per joule (768 NGINX connections). Disabling SMT causes the efficiency to drop by 17 % and 35 %, respectively.

Discussion The benchmark runs with different SMT configurations on the AMD Ryzen 7 and i7-10700K systems show the impact hardware multithreading has on both performance and power consumption. As with the evaluation of the Spectre and Meltdown mitigations costs, the results have to be interpreted in light of the CPU-heavy workload used as the benchmark. Thus, for diverse workloads in productive systems, more evaluations reflecting these workload characteristics have to be conducted to give a more comprehensive picture of the impact of SMT. For the sysbench `CPU` benchmark, disabling hardware multithreading always results in performance drops and reduced energy efficiency. The decrease in energy efficiency becomes more prominent the higher the background load generated by the web server. This showcases the importance of considering the entire system state when deciding on the most efficient mitigation configuration. Also, this result raises the question of whether it may be more efficient to adapt the scheduling of different applications to the system state when optimizing for energy efficiency. On a system with high CPU utilization, it may prove to be more efficient to stall scheduling a process requiring process isolation until the system load has decreased. This, of course, requires knowledge or predictions of the future system load. It would also increase the process's latency, which may not always be tolerable. Nonetheless, including scheduling strategies in dynamic mitigation reconfiguration decisions is an interesting subject for further evaluations and future work on this thesis.

Summary This section presented the cost evaluation results for different hardware-vulnerability mitigation configurations. The costs were determined in terms of performance, power consumption, and energy efficiency. For the Spectre mitigations, the more recently developed processors show little overhead when activating these mitigations as hardware-specific implementations are still in use. On the older i5-8400 system, the Spectre mitigations reduced both the performance and the energy efficiency by almost 6 %. KPTI showed no significant overhead, though this can be attributed to the choice of a CPU-heavy benchmark. Evaluations with different workloads may yield different results [32]. For disabling SMT, the evaluation shows significant performance and energy-efficiency losses for the sysbench `CPU` benchmark. The decrease in efficiency becomes more prominent with increasing background load. These results show that the dynamic reconfiguration of hardware-vulnerability mitigations has the potential to improve a system's efficiency, especially if no hardware mitigation implementations are available. For mitigating MDS-based attacks, additional mitigation implementations such as Core Scheduling, which reduces performance and energy efficiency penalties, should be considered.

6.4 Reconfigurability Overhead

To enable dynamic control of the Spectre and Meltdown mitigations, kernel extensions are required. The mainline kernel implementation of patching mitigations into the image at boot time is replaced by jump-label code-patching directives. With this mechanism, disabling mitigations inserts a jump instruction into the live kernel image so that the respective mitigation implementations are omitted. On enabling the mitigation, this jump is replaced with a no-operation instruction. In some paths, code patching is not possible with the current patching mechanism. Thus, the original code-patching instructions there are replaced by conditional jumps to allow for the evaluation of the dynamic reconfiguration. The overhead of these changes is determined by running the sysbench `CPU` benchmark on both mainline and modified kernel. Relative to the mainline kernel's geometric mean of

performance and power consumption, the overhead of the changes to the kernel is determined. The overhead is calculated per mitigation configuration. In the plots, the 95 % confidence interval of the overhead is indicated by error bars, while the baseline's confidence interval is represented by gray bars. For all benchmark runs, DVFS was disabled on the systems under test.

Results The overhead in performance and power consumption induced by the kernel changes is illustrated in Figure 6.6 and Figure 6.7. For the i5-8400 system, the evaluation shows that introducing dynamic control of the Spectre and Meltdown mitigations incurs low to zero performance overhead. Noteworthy are the overhead value determined for the configurations with either KPTI or Spectre mitigations enabled. Enabling either one of the mitigations shows a more pronounced overhead compared to dis- or enabling both. With KPTI enabled, the performance overhead reaches 0.83 %. Enabling the Spectre mitigations, in contrast, causes an increase in performance compared to the mainline kernel by 0.92 %. The 95 % confidence intervals for the two configurations indicate a greater variation in the measured performance, though none of the values within the confidence interval exceed 2 % overhead. Considering the power-consumption overhead, the results show similar low overheads that range between +1 % and -1 %. An exception is the overhead determined for the configurations with both Spectre and Meltdown mitigations enabled. Here, a more prominent negative overhead of -1.6 % is observed.

Results for the two more recent processors show that the changes to the kernel and the additional instructions do not result in high overhead. Only the Spectre mitigation configurations are evaluated on these systems as they are not susceptible to Meltdown-based attacks. The values for the AMD Ryzen 7 are plotted in Figure 6.7a. The performance overhead on this system for both enabled and disabled Spectre mitigations is close to zero with -0.03 % and -0.17 %, respectively. The power consumption overhead does not exceed 1 % (0.08 % and 0.23 %) when comparing execution with enabled and disabled Spectre mitigations. Results for the i7-10700K show similar results, though the individual overheads are slightly higher. Running the benchmark with active Spectre mitigations on the modified kernel results in a performance overhead of 0.34 % compared to the mainline kernel with the same mitigation configuration. For disabled Spectre mitigations, the overhead is slightly

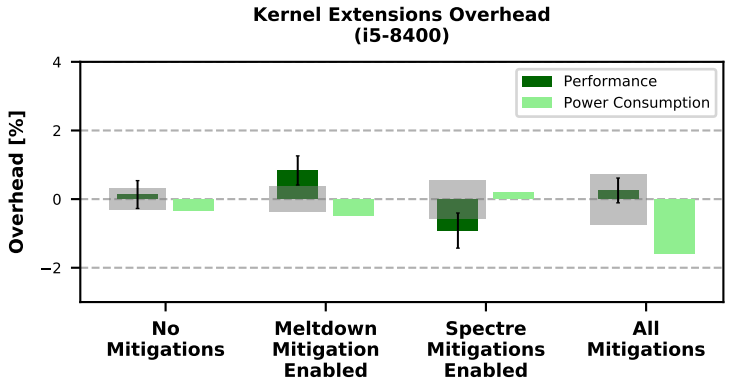
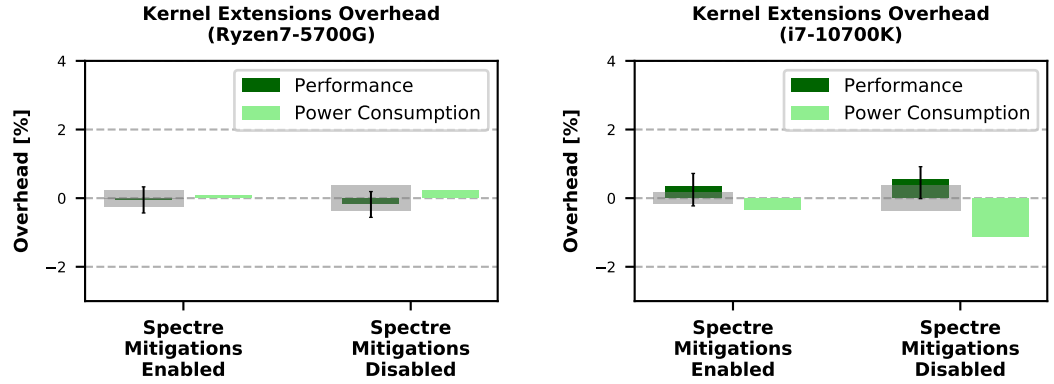


Figure 6.6 – Overhead of the kernel extensions for dynamic control of the Spectre and Meltdown mitigations on the i5-8400 system. The performance and power consumption of the modified kernel are normalized to the corresponding mainline kernel results. The gray bars indicate the baseline's 95 % confidence interval. The error bars indicate the measurement's 95 % confidence interval.

higher at 0.54 %. The power consumption, in contrast, shows an improvement when the modified kernel is booted as opposed to execution with the mainline kernel. The most significant reduction in power consumption is observed when disabling the Spectre mitigations. This appears more effective for the custom kernel as a negative overhead of -1.15 % is determined.

Discussion The evaluation of the overhead induced by the kernel extensions shows that, in general, the changes cause only a small overhead. For the i5-8400 system, the results show a small overhead for the configurations with all mitigations enabled and with no mitigations active. As the values lie within the baseline's 95 % confidence interval, the overhead can be considered insignificant. In contrast, the results for active KPTI show overheads that exceed the baseline's confidence interval. The overhead, in general, can be attributed to implementation details. The code-patching mechanism utilized for dynamic mitigation control poses problems when used in combination with the CPU synchronizations required for depoisoning the kernel page table. Thus, to allow dynamic reconfiguration of KPTI, the code sections modifying the `CR3` register on kernel entry and exit are mainly changed to include conditional jumps. The modified kernel, therefore, includes more instructions. As part of future work on the dynamic reconfiguration of mitigations, the code-patching mechanism could be adapted, further reducing the overhead. The slight increase in performance with enabled Spectre mitigations could be an effect of more efficient code generated by the compiler. Overall, the evaluation shows that the implementation of dynamically reconfigurable Spectre and Meltdown mitigations only incurs minor overheads on the i5-8400system.

Results for the AMD Ryzen 7 show only insignificant performance and power-consumption overheads. For the i7-10700K, the overheads differ only slightly from the baseline. The kernel configuration with Spectre mitigations enabled results in a higher performance overhead compared to Spectre mitigations disabled. This can also be attributed to implementation details. The code patching inserts additional jump instructions on disabling the mitigations. Thus, the kernel requires more instructions and cycles per kernel entry/exit than in active mitigation configurations. The small overhead detected for the active Spectre mitigations can be explained by the code paths where code patching could not be utilized and conditional jumps are used instead.



(a) Results for the AMD Ryzen 7 system.

(b) Results for the i7-10700K system.

Figure 6.7 – Overhead of the kernel extensions for dynamic control of the Spectre mitigations. The performance and power consumption of the modified kernel are normalized to the corresponding mainline kernel results. The gray bars indicate the baseline's 95 % confidence interval.

Summary This section analyzed and discussed the extensions to the mainline-kernel code basis and their impact on performance and power consumption. The changes are required to enable the dynamic control of the Spectre and Meltdown mitigation configuration. For the i5-8400 system, the kernel extensions induce only minor overhead. If mitigations are either entirely en- or disabled, the overhead does not surpass the baseline's confidence interval. With the Meltdown mitigation enabled, the performance overhead is most pronounced but does not exceed 1 % overhead. For the AMD Ryzen 7, no significant performance overhead could be identified. The i7-10700K shows a small performance overhead for both active and inactive Spectre mitigations, though it does not exceed 0.6 %. The power-consumption analysis also shows only minor overhead induced by the kernel changes. Overall, this evaluation shows that extending the kernel to allow for the dynamic reconfiguration of the Spectre and Meltdown mitigations incurs only a small overhead with the given evaluation setup and benchmark.

6.5 MDS Mitigations Properties

For process isolation, the Linux kernel provides two mechanisms: disabling SMT on the system and isolating a group of processes via Core Scheduling. While changes to the SMT settings can have significant effects on the system's performance, as shown in Section 6.3, Core Scheduling has the potential to limit these costs. To further analyze this potential, the performance of the sysbench CPU benchmark is evaluated with both process isolation mechanisms. Each benchmark run is repeated with a different system load, which is generated by the NGINX web server. To simulate productive conditions, DVFS was enabled. This evaluation is only run on the AMD Ryzen 7 and i7-10700K system, as the i5-8400 processor does not offer SMT support.

Results The performance of the benchmark and the system power-consumption under different background loads is plotted for both evaluation platforms in Figure 6.8 and Figure 6.10. As a

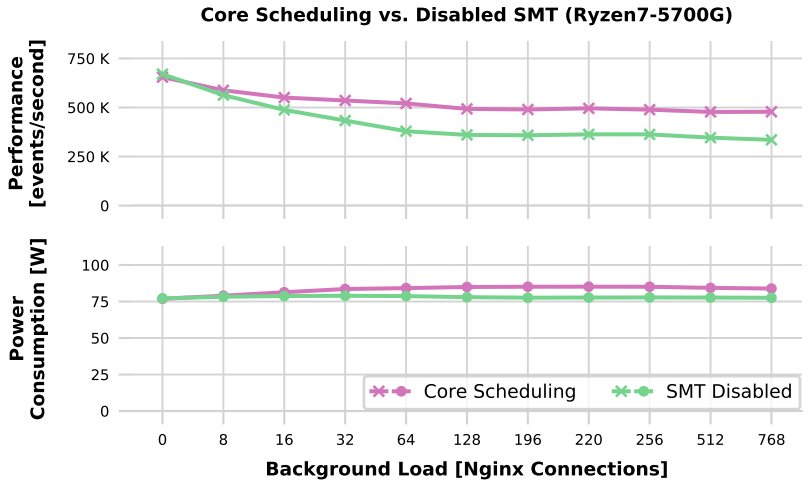


Figure 6.8 – Evaluation of the performance of the sysbench CPU benchmark and system power consumption for SMT disabled and Core Scheduling on the AMD Ryzen 7 system.

metric for the background load serves the number of parallel connections to the web server over which two clients continuously send requests. For both systems, these connections result in different CPU utilization. The background load induced by the NGNIX web server ranges between 12 % (8 parallel connections) and 41 % (768 parallel connections) utilization for the AMD system. On the Intel system, the same number of connections generates utilization between 8 % and 81 %.

The results for the AMD Ryzen 7 show that for most measurements, Core Scheduling outperforms the benchmark execution without hardware multithreading. The first two benchmark executions with low background load (zero and eight parallel connections) are the exception from this observation. Without additional system load (zero NGINX connections), both Core Scheduling and disabling SMT show close to the same result. Here, the benchmark execution without SMT is the more energy-efficient configuration as the performance is slightly higher while the power consumption is below that of Core Scheduling. This increase in energy efficiency is limited to approximately 2 %. All results in terms of energy efficiency are plotted in Figure 6.9. With increasing system load, Core Scheduling becomes the more efficient process isolation mechanism for the chosen CPU-heavy workload. At 8 NGINX connections, disabled SMT still ranges close to Core Scheduling in power consumption and performance. Further increasing the parallel connections causes more substantial performance degradation on the system with hardware multithreading disabled. This results in Core Scheduling outperforming disabled SMT by 30 % at 768 NGINX connections. In terms of power consumption, Core Scheduling is the more expensive variant. This, though, does not outweigh the increase in performance. Thus, Core Scheduling is determined to be the more energy-efficient option on the AMD Ryzen 7.

The evaluation on the i7-10700K yields similar results, with Core Scheduling being the overall more energy-efficient form of providing process isolation. The results are plotted as events per joule in Figure 6.11. At zero NGINX connections, Core Scheduling and disabled SMT show close to the same performance and only minimal differences in power consumption. In contrast to results with higher background load, the execution without hardware multithreading shows slightly better results (1 % and 2 %). At 8 NGINX connections, the performance of disabled SMT displays a first significant drop while Core Scheduling's performance remains more stable. This results in Core Scheduling showing a 9 % increase in energy efficiency compared to disabled SMT. With 32 connections, Core Scheduling completes 2191.96 events per joule, while the run without hardware multithreading calculates 1629.67 events per joule. Thus, the latter's energy efficiency is 26 % below Core Scheduling.

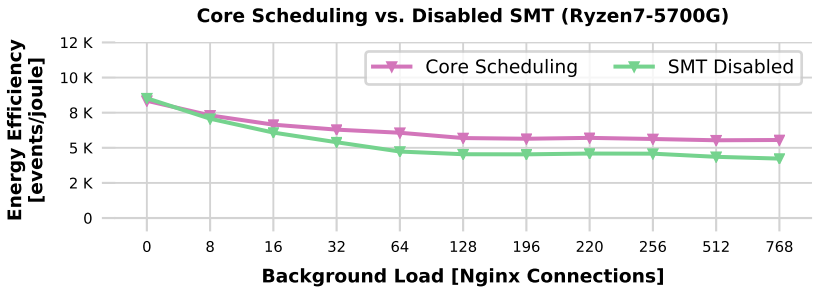


Figure 6.9 – Evaluation of the energy efficiency of SMT disabled and Core Scheduling during the sysbench CPU benchmark execution on the AMD Ryzen 7 system.

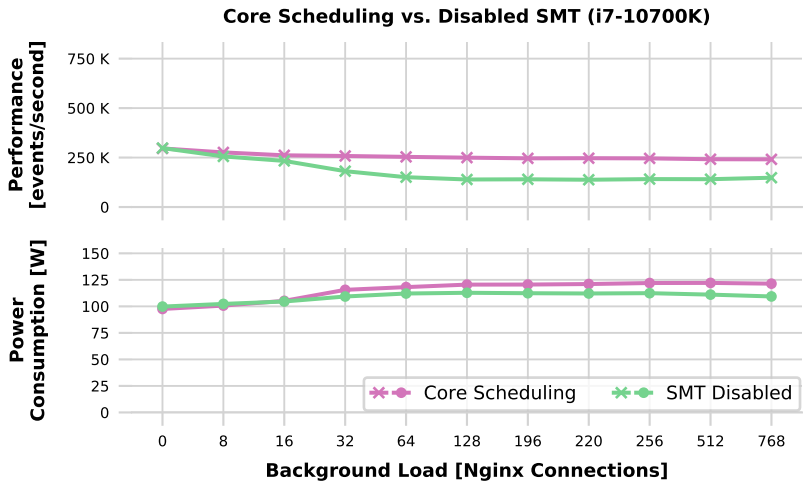


Figure 6.10 – Evaluation of the performance of the sysbench CPU benchmark and system power consumption for SMT disabled and Core Scheduling on the i7-10700K system.

Discussion The evaluation of the two process-isolation mechanisms shows that Core Scheduling outperforms disabled SMT on both evaluation systems. Disabling SMT reduces the number of available CPU threads to 8, which is below the number of sysbench and NGINX worker-threads. Thus, high contention between the threads and frequent preemption are expected, which significantly affects the performance. Even though execution without hardware multithreading reduces the system's overall power consumption, this cannot outweigh the corresponding performance degradation. Thus, the energy efficiency of disabling SMT is below that of Core Scheduling for both systems. Only with a low background load do the two process isolation mechanism show close to the same efficiency.

Without any NGINX connections, Core Scheduling's performance does not significantly exceed that of disabled SMT. This behavior is also described in the Linux kernel's official documentation. In the corresponding entry [10], the developers point out that Core Scheduling might perform worse

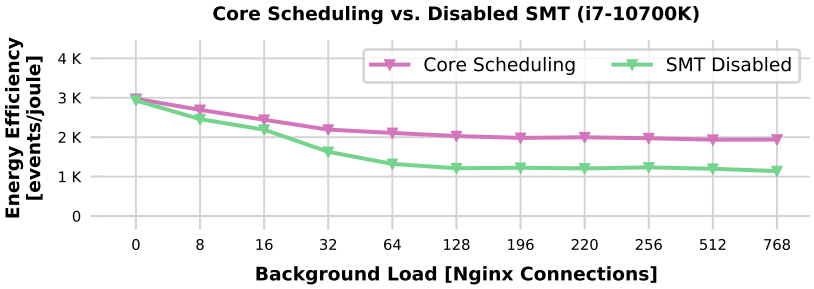


Figure 6.11 – Evaluation of the energy efficiency of SMT disabled and Core Scheduling during the sysbench CPU benchmark execution on the i7-10700K system.

than disabling SMT on lightly loaded systems. This is attributed to the scheduling synchronization overhead that becomes more significant if the overall load is relatively low. Thus, monitoring the overall system load is an important factor in determining the most efficient mitigation configuration. Further evaluations focusing on process isolation in systems with low system load could help identify detailed parameters for defining thresholds for the use of Core Scheduling. Also, the effect of Core Scheduling on other processes in the system should be examined in further evaluations. Utilizing Core Scheduling to isolate processes from untrusted processes effectively influences the entire system's scheduling decisions. This interference and reduced availability of CPU threads might affect the overall system performance depending on the number of threads being isolated and the number of process groups that have to be separated from each other.

Summary In this section, two mechanisms for providing process isolation were evaluated with regard to their performance and energy efficiency. While disabling SMT is an efficient mitigation for MDS-based attacks and protects all processes on the system, it incurs significant performance degradation and reduces the system's overall energy efficiency. Core Scheduling is used to isolate individual process groups by ensuring that untrusted processes are not scheduled on the same physical core. This more narrow approach aims to reduce the impact of process isolation. The evaluation in this section shows that with Core Scheduling, the sysbench CPU benchmark performs significantly better compared to disabling SMT. Considering the energy efficiency of the two approaches to process isolation, Core Scheduling also proves to be more efficient, especially with a high system load.

6.6 DRConf in Practice

Dynamic reconfiguration of hardware-vulnerability mitigations aims to not only provide processes with required protection but also to optimize the mitigation configuration in terms of performance and power consumption. The previous evaluations analyzed the different mitigations and their impact on the system's efficiency. In this section, an evaluation of dynamic reconfigurations in practice is presented. For this, two evaluation scenarios are designed. Both encompass the sequential execution of two applications, simulated by use of the sysbench CPU benchmark, with diverging security demands. For the i5-8400 system, the second application does not require mitigations for the Spectre and Meltdown vulnerabilities. Therefore, the kernel providing dynamic reconfigurations can disable the mitigations before the second application is scheduled. The performance and power consumption of this experiment are compared to the execution on the mainline kernel, which does not allow for disabling the mitigations at run time. On the AMD Ryzen 7 and i7-10700K

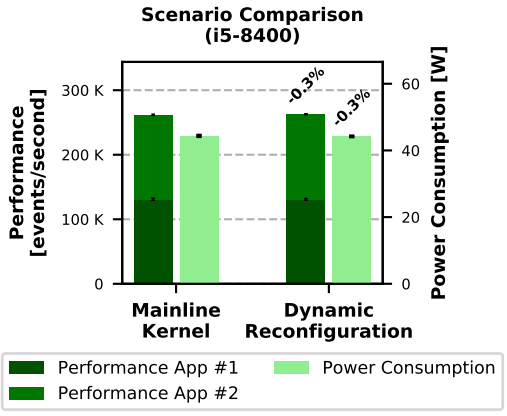
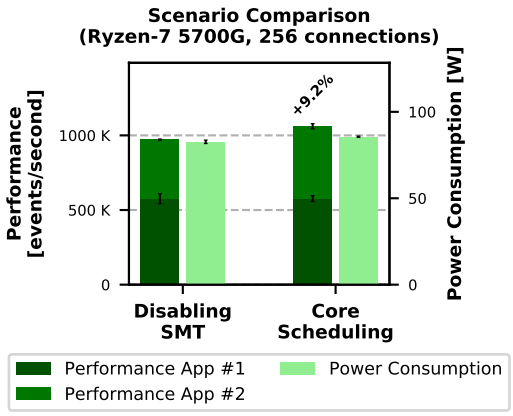


Figure 6.12 – Evaluation results of dynamic Spectre and Meltdown mitigation reconfiguration on the i5-8400 system. The mainline kernel does not offer dynamic control of these mitigations. Thus, all mitigations are enabled for both applications. With dynamic reconfigurations, the mitigations are before the second application starts its execution.

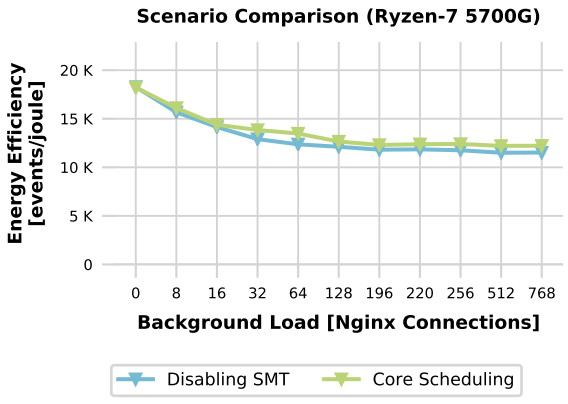
systems the evaluation analyzes the use of Core Scheduling and disabling SMT to provide a second application with protection from MDS-based attacks. The NGINX web-server on these systems is again utilized to generate system load. Each scenario is repeated five times.

Results on the i5-8400 platform The results of the first evaluation of dynamic reconfigurations in use are shown in Figure 6.12. Dynamically reconfiguring the Spectre and Meltdown mitigations in between the execution of the two applications yields a minor performance penalty of -0.3 %. In terms of power consumption, the evaluation shows that reconfiguring the mitigations results in a minor improvement of also -0.3 %. Thus, the resulting energy efficiency of both runs does not differ significantly.

Discussion The evaluation of the dynamic reconfigurations of Spectre and Meltdown mitigations on the i5-8400 showed no increase in performance or energy efficiency of the scenario execution. Even though the evaluation of the mitigation costs showed some potential for improving the system's efficiency when disabling the mechanisms if not required, this did not translate into this evaluation. The potential gain in performance was 6.7 % for the execution of only the second application as determined in Section 6.3. Thus, the overall performance gain would be expected to be reduced. In addition, the performance of the first application is slightly increased as the mitigation configurations are the same as on the mainline kernel, but the minor overhead induced by the extensions comes into factor. Also, the use of the sysbench CPU benchmark does not adequately simulate more diverse workloads of productive systems. For a more comprehensive understanding of how dynamic reconfigurations of Spectre and Meltdown mitigations may or may not improve a system's efficiency, evaluations with different workloads are needed.



(a) Power Consumption and performance of disabling SMT and utilizing Core Scheduling for the second application with 256 connections to the NGINX web server.

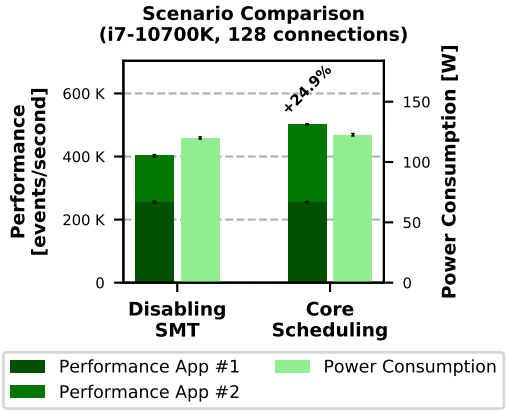


(b) Energy efficiency of disabling SMT and utilizing Core Scheduling for the second application over varying background load.

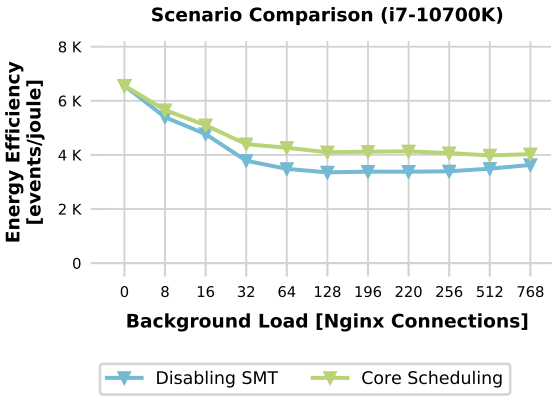
Figure 6.13 – Results of evaluating the different process-isolation mechanisms and their dynamic reconfiguration on the AMD Ryzen 7 system. a) shows the results with the highest performance gain at 256 NGINX connection. b) shows the energy efficiency with varying background load.

Results on the AMD Ryzen 7 and i7-10700K platforms Providing applications with process isolation can be achieved by either disabling SMT or utilizing Core Scheduling. How this change affects the overall performance, power consumption, and energy efficiency of the execution on the AMD Ryzen 7 and i7-10700K system is plotted in Figure 6.13 and Figure 6.14, respectively. For the AMD system, Figure 6.13b shows the energy efficiency of the chosen strategies (disabling SMT before application 2 is dispatched or isolating the processes with Core Scheduling). The plot illustrates that as soon as background load is induced by the web server, Core Scheduling is the more efficient strategy. While the efficiency of both mechanisms decreases with increasing system load, the execution with SMT disabled cannot match the other strategy's efficiency. The greatest difference is at 64 parallel connections. Here, the use of Core Scheduling proves to be more efficient by almost 9 %. In contrast, when optimizing for performance only, Core Scheduling shows the highest increase in performance compared to disabled SMT at 256 NGINX connections, with 9.2 %.

For the i7-10700K system, Core Scheduling is again the more efficient choice. As with the AMD system, utilizing Core Scheduling results in the highest increase in energy efficiency at 64 NGINX connections. With this background load, the energy efficiency is almost 25 % above that of disabling SMT for the second application. Overall, the results for the Intel system show that the efficiency of Core Scheduling remains relatively stable after an initial drop when increasing the number of NGINX connections from zero to 64. The efficiency of disabling SMT, in contrast, shows a more steep degradation with less than 64 parallel web-server connections. Thereafter, the efficiency remains relatively stable. At 768 NGINX connections, the efficiency of disabling hardware multithreading for the second application is only reduced by approximately 11 % compared to the use of Core Scheduling. If optimizing for performance only, Core Scheduling is the better choice in all cases but the first data point at zero NGINX connections. At 128 NGINX connections, Core Scheduling proves to be the most efficient in terms of performance compared to disabling



(a) Power Consumption and performance of disabling SMT and utilizing Core Scheduling for the second application with 128 connections to the NGINX web server.



(b) Energy efficiency of disabling SMT and utilizing Core Scheduling for the second application over varying background load.

Figure 6.14 – Results of evaluating the different process-isolation mechanisms and their dynamic reconfiguration on the i7-10700K system. a) shows the results with the highest performance gain at 128 NGINX connection. b) shows the energy efficiency with varying background load.

SMT. The corresponding performance and power consumption are plotted in Figure 6.14a. The performance of disabling hardware multithreading also slightly improves with higher background load from the minimum at 128 parallel connections. But the performance in comparison to Core Scheduling does not recover as much as does the energy efficiency. At the maximum background load tested in this evaluation, Core Scheduling is still outperforms deactivating SMT by almost 21 %.

Discussion The evaluations of the different process isolation mechanisms in use support the findings of Section 6.5. On both systems, Core Scheduling is the most efficient solution both in terms of performance and energy efficiency. Only without additional background load does disabling SMT perform close to or better than Core Scheduling. This, again, shows that additional evaluations taking into focus systems with low CPU utilization are of interest. With their help, thresholds for switching process-isolation strategies for individual systems could be identified. This could be of use as this evaluation also shows that the two processors show different behavior regarding the performance and energy efficiency disabling SMT.

Summary The evaluation of dynamic mitigation reconfigurations and different mitigation configurations in use was presented in this section. For utilizing dynamic control of the Spectre and Meltdown mitigations, the results of this evaluation show no significant improvement in the performance and energy efficiency. One factor for this result is the choice of benchmark, as CPU-heavy workloads are not as susceptible to performance degradation as other workloads. For a more comprehensive understanding of the impact of dynamic reconfigurations, further evaluations can be of use. Analyzing the efficiency of the different process isolation mechanisms in use shows that with additional background load Core Scheduling is the more efficient mitigation. The evaluation also indicates the impact of the underlying hardware on the most efficient strategy.

6.7 Summary

This chapter presented the evaluations of dynamic reconfigurations of hardware-vulnerability mitigations. The evaluations analyzed different aspects of mitigation configurations in the Linux kernel. The first experiments focused on determining the impact of the different mitigations on the respective system's performance and power consumption. In general, if hardware mitigations are available, the impact of the mitigations is limited to a minimum. Otherwise, the protection mechanisms induce a noticeable overhead. Section 6.4 analyzed the overhead caused by the kernel code extensions to allow for dynamic control of the Spectre and Meltdown mitigations. The results show that the extensions impact the performance, though only on two of the evaluation platforms significant performance impacts were found. The overhead found on these two systems does not exceed 1 %. The third evaluation analyzed how disabling SMT and Core Scheduling, as MDS mitigations, differ in their performance and energy efficiency. With the exception of systems with low system load, Core Scheduling is the more efficient mechanism for providing process isolation. The evaluation only considered the performance of the benchmark isolated with one of the two mitigations. Here, further evaluations could analyze Core Scheduling's impact on the overall performance of all processes deployed on the system. The evaluation also shows varying performance and efficiency results depending on the underlying hardware, indicating that reconfiguration strategies have to be adapted to the individual systems. Section 6.6 presented the evaluation of dynamic reconfigurations and different mitigation configurations in use. Changing the mitigation setting in between the execution of two applications showed different results depending on the reconfigured mitigations and underlying system. Dynamic control of the Spectre and Meltdown mitigations did not result in increased

performance or efficiency. This can be attributed partly to the choice of benchmark. Thus, further evaluations to gain a comprehensive understanding of the impact of dynamic reconfiguration of these mitigations are suggested. Analyzing the use of Core Scheduling at run time compared to disabling SMT supported the findings of Section 6.5. Core Scheduling was also identified as the more efficient mitigation except for the scenarios with low background load.

7 FUTURE WORK

This thesis presents the design and implementation of dynamic hardware-vulnerability mitigation reconfigurations in the Linux kernel. Chapter 6 also provided an analysis and evaluation of the reconfigurations and their impact on performance, power consumption, and energy efficiency. This chapter gives an overview of how to develop this thesis' approach further and discusses which additional features could be incorporated. First, internal design and implementation refinements are proposed. This is followed by outlining how Trusted Execution Environments (TEEs) and machine learning components could offer additional functionalities and improve dynamic reconfigurations.

The current implementation of hardware-vulnerability mitigation reconfigurations at run time already allows for testing its functionality, as shown in Chapter 6. However, some implementation details limit the applicability and possible performance gain. The dynamic control of the Spectre and Meltdown mitigations is implemented with the help of run-time code patching based on the jump-label mechanism. However, this mechanism cannot be used to reconfigure the mitigations in all code paths. Specifically, some kernel-entry and -exit functions that are executed during specific interrupts prove problematic. Also, the use of jump-label code patching in combination with CPU synchronizations required for reconfiguring Kernel Page Table Isolation (KPTI) is currently limited. This is described in more detail in Section 5.3. Therefore, further development should take into focus this code-patching mechanism and how it could be adapted for use in all cases. Removing the provisional conditional jump constructs and replacing them with static jumps increases the potential performance gain as the number of instructions executed is significantly reduced. Apart from adapting the code-patching mechanism, the dynamic mitigation reconfigurations could be extended to offer applications customized Trusted Execution Environments (TEEs). With customizable TEEs, such as Keystone [40] and CURE [5], enclaves can be configured in light of typical workload security requirements and in light of the underlying hardware. For example, Keystone enclaves could be configured to offer users secure memory areas or cache partitioning to mitigate side-channel attacks on the TEE if the underlying hardware provides these features.

As analyzed in Section 6.5, the most efficient choice of protection against cross-HT attacks depends on several parameters. Workload and hardware characteristics in combination with the current system load are defining factors in determining which isolation mechanism is the most efficient. Thus, defining indicators and thresholds for when to utilize which mitigation is essential to realizing the full potential of dynamic mitigation reconfigurations. For this, testing and experiments prior to deploying the system are required. This task could also be realized by deploying machine-learning components that determine the optimal parameters during an initial learning phase. In addition, machine-learning components could be deployed to continuously monitor the system's efficiency and optimize thresholds over time. Thereby, varying workload characteristics or changing system behavior due to aging effects could be accounted for.

8 CONCLUSION

Providing secure systems is a key task of providers of different computing centers, such as cloud services. One aspect is to provide secure system software and protect applications from defective or malicious software. Another aspect of security is providing protection from vulnerabilities that are located in the hardware itself. Attacks like Spectre, Meltdown, and Microarchitectural Data Sampling (MDS) exploit the hardware implementation of performance-critical features like out-of-order and speculative execution. By way of vulnerabilities in these features, attackers can gain access to privileged information. Thus, mitigating these vulnerabilities is essential to providing a secure system. These mitigations often induce significant overhead in both performance and power consumption. Therefore, fine-grained control that allows enabling protection mechanisms only when needed could allow minimizing the overhead.

This thesis has introduced dynamic reconfigurability of hardware-vulnerability mitigations for the Linux kernel. By utilizing run-time control of mitigations, the approach of this thesis allows tailoring the system's configuration to the protection demands of the processes in execution. With a user-space service that manages the security requirements of applications and monitors the system state, the mitigation configuration can be continuously adapted to avoid the overhead of unnecessarily enabled mitigations. Additionally, the service can configure the most efficient combination of the mechanisms available based on predetermined system characteristics. If multiple mitigation implementations, for example, for mitigating cross–Hyper-Thread attacks, exist, the service determines which is the most efficient choice in light of system specifics, system load, and workload. Corresponding strategies are determined by the provider prior to deploying the system productively. To allow for dynamic reconfiguration of all mitigations, a kernel module is presented that implements run-time control of the Spectre and Meltdown mitigations. In addition to the kernel module, kernel patches are developed to add reconfiguration functionality and to integrate code-patching directives into the kernel source code. With the help of code patching, the mitigation implementations are removed from the control flow if the system does not require protection from the respective vulnerability. Combined, the service, the kernel module, and the kernel extensions provide systems with dynamic reconfigurability of hardware-vulnerability mitigations and allow for optimizing the system's efficiency.

The evaluation of the dynamic reconfigurability showed that by introducing this functionality to the Linux kernel, only minor overhead is induced. Its effect on the system's performance and efficiency is limited by the overhead induced by the mitigations themselves. Hardware specifics, system state, and workload characteristics have been determined as essential factors in deciding whether the use of dynamic mitigation reconfigurations yields significant efficiency improvements. These factors are also critical to determining the optimal configuration. In the case of process isolation, Core Scheduling is the most efficient mechanism if the system load surpasses a system-

specific threshold. Otherwise, it might be more profitable to disable Symmetric Multithreading (SMT) to provide isolation. Further evaluations with more diverse workloads could be used to gain a more comprehensive understanding of the impact of the different mitigations and their implementations on a system's efficiency. Also, as part of further development on dynamic reconfigurability, support for additional security features, such as customizable Trusted Execution Environments (TEEs), could be integrated. The service could also be trained to provide scheduling hints if delaying processes with higher protection demands could improve the overall performance.

In summary, this thesis has introduced dynamic reconfigurability of hardware-vulnerability mitigations and has presented an implementation for the Linux kernel that is integrated with minor overhead. The evaluation showed that with mitigations inducing significant overheads, the dynamic reconfiguration and mitigation selection based on the system state can be utilized to improve the system's efficiency. The efficiency gains that can be achieved depend on both hardware and the costs the mitigations induce in general. Overall, the minimal overhead design and implementation presented by this thesis provide the groundwork for further research into the efficiency optimization of reconfigurable hardware-vulnerability mitigations.

LIST OF ACRONYMS

CPU	Central Processing Unit
DVFS	Dynamic Voltage and Frequency Scaling
eIBRS	enhanced Indirect Branch Restricted Speculation
HT	Hyper-Threading
IBPB	Indirect Branch Prediction Barrier
IBRS	Indirect Branch Restricted Speculation
KPTI	Kernel Page Table Isolation
KASLR	Kernel Address Space Layout Randomization
LFB	Line Fill Buffer
MDS	Microarchitectural Data Sampling
MSR	Model-Specific Register
NMI	Non-Maskable Interrupt
PDBR	Page Directory Base Register
PGD	Page Global Directory
PTE	Page Table Entry
RSB	Return Stack Buffer
SB	Store Buffer
SMAP	Supervisor Mode Access Prevention
SMP	Symmetric Multiprocessing
SMT	Symmetric Multithreading
STIBP	Single Thread Indirect Branch Predictor
TEE	Trusted Execution Environment
TLB	Translation Lookaside Buffer
WTF	Write Transient Forwarding

LIST OF FIGURES

REFERENCES

[1] Omar Alhubaiti and El-Sayed M. El-Alfy. "Impact of Spectre/Meltdown Kernel Patches on Crypto-Algorithms on Windows Platforms." In: *Proceedings of the International Conference on Innovation and Intelligence for Informatics, Computing, and Technologies (3ICT '19)*. 2019, pp. 1–6.

[2] AMD. *AMD Ryzen 7 5700G Processor Specifications*. [Online] last accessed 2022-05-31. URL: https://www.amd.com/en/products/apu/amd-ryzen-7-5700g.

[3] AMD. *Software Techniques for Managing Speculation on AMD Processors, Revision 3.5.22*. [Online] last accessed 2022-05-22; URL: https://www.amd.com/system/files/documents/software-techniques-for-managing-speculation.pdf.

[4] Vrije Universiteit Amsterdam. *MDS: Microarchitectural Data Sampling*. [Online] last accessed 2022-06-08. URL: https://mdsattacks.com/.

[5] Raad Bahmani et al. "CURE: A Security Architecture with CUstomizable and Resilient Enclaves." In: *Proceedings of the 30th USENIX Security Symposium (USENIX Security '21)*. 2021, pp. 1073–1090.

[6] Enrico Barberis et al. "Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks." In: *Proceedings of the 31st USENIX Security Symposium (USENIX Security '22)*. Prepublication. 2022.

[7] L. Benini et al. "Monitoring system activity for OS-directed dynamic power management." In: *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED '98)*. 1998, pp. 185–190.

[8] Claudio Canella et al. "Fallout: Leaking Data on Meltdown-Resistant CPUs." In: *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. 2019, pp. 769–784.

[9] Jonathan Corbet. *Addressing Meltdown and Spectre in the kernel [LWN.net]*. [Online] last accessed 2022-06-08. URL: https://lwn.net/Articles/743265/.

[10] *Core Scheduling – The Linux Kernel documentation (version 5.14)*. [Online] last accessed 2021-06-10. URL: https://www.kernel.org/doc/html/v5.14/admin-guide/hw-vuln/core-scheduling.html.

[11] *Core scheduling lands in 5.14 [LWN.net]*. [Online] last accessed 2022-06-06. URL: https://lwn.net/Articles/861251/.

[12] Intel Corporation. *Indirect Branch Predictor Barrier*. [Online] last accessed 2022-05-22. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-predictor-barrier.html.

[13] Intel Corporation. *Indirect Branch Restricted Speculation*. [Online] last accessed 2022-05-22. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html.

[14] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Vol. 2A*. [Online] last accessed 2022-06-07. 2022. URL: https://cdrdv2.intel.com/v1/dl/getContent/671200.

[15] Intel Corporation. *Intel Core i5-8400 Processor Specifications*. [Online] last accessed 2022-05-31. URL: https://ark.intel.com/content/www/us/en/ark/products/126687/intel-core-i58400-processor-9m-cache-up-to-4-00-ghz.html.

[16] Intel Corporation. *Intel Core i7-10700K Processor Specifications*. [Online] last accessed 2022-05-31. URL: https://ark.intel.com/content/www/us/en/ark/products/199335/intel-core-i710700k-processor-16m-cache-up-to-5-10-ghz.html.

[17] Intel Corporation. *Intel Security Features and Technologies Related to Transient Execution Attacks*. [Online] last accessed 2022-06-08. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/related-intel-security-features-technologies.html.

[18] Intel Corporation. *Retpoline: A Branch Target Injection Mitigation*. [Online] last accessed 2022-06-07. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/retpoline-branch-target-injection-mitigation.html.

[19] Intel Corporation. *Single Thread Indirect Branch Predictors*. [Online] lsat accessed 2022-06-07. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/single-thread-indirect-branch-predictors.html.

[20] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. "HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments." In: *Proceedings of the 29th USENIX Security Symposium (USENIX Security '20)*. 2020, pp. 451–468.

[21] Gabor Drescher. "Adaptive Address-Space Management for Resource-Aware Applications." Doctoral Thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 2021. URL: https://opus4.kobv.de/opus4-fau/frontdoor/index/index/docId/16511.

[22] S.J. Eggers et al. "Simultaneous multithreading: a platform for next-generation processors." In: *IEEE Micro* 17.5 (1997), pp. 12–19.

[23] Krisztián Flautner and Trevor Mudge. "Vertigo: Automatic Performance-Setting for Linux." In: *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*. 2002.

[24] Sebastien Godard. *Sysstat Utilities Home Page: mpstat manual page*. [Online] last accessed 2022-06-02. URL: http://sebastien.godard.pagesperso-orange.fr/man_mpstat.html.

[25] Michael Godfrey and Mohammad Zulkernine. "Preventing Cache-Based Side-Channel Attacks in a Cloud Environment." In: *IEEE Transactions on Cloud Computing (TCC)* 2.4 (2014), pp. 395–408.

[26] Brendan Gregg. *KPTI/KAISER Meltdown Initial Performance Regressions*. [Online] last accessed 2022-06-06. 2019. URL: https://www.brendangregg.com/blog/2018-02-09/kpti-kaiser-meltdown-performance.html.

[27] Daniel Gruss et al. "KASLR is Dead: Long Live KASLR." In: *Proceedings of the 9th International Symposium on Engineering Secure Software and Systems (ESSoS '17)*. Ed. by Eric Bodden, Mathias Payer, and Elias Athanasopoulos. 2017, pp. 161–176.

[28] Dave Hansen. *LKML: [PATCH 00/23] KAISER: unmap most of the kernel from userspace page tables*. [Online] last accessed 2022-05-22. URL: https://lkml.org/lkml/2017/10/31/884.

[29] John L Hennessy and David A Patterson. *Computer Architecture – A Quantitative Approach*. Ed. by Nate McFadden Todd Green. 5th ed. Waltham, MA: Morgan Kaufmann, 2012. Chap. 3.

[30] John L Hennessy and David A Patterson. *Computer Organization and Design – The Hardware / Software Interface*. Ed. by Nate McFadden Todd Green. 5th ed. Oxford / Waltham, MA: Morgan Kaufmann, 2014. Chap. 4.

[31] Benedict Herzog et al. "Automated Selection of Energy-Efficient Operating System Configurations." In: *Proceedings of the 12th ACM International Conference on Future Energy Systems (e-Energy '21)*. 2021, pp. 309–315.

[32] Benedict Herzog et al. "The Price of Meltdown and Spectre: Energy Overhead of Mitigations at Operating System Level." In: *Proceedings of the 14th European Workshop on Systems Security (EuroSec '21)*. 2021, pp. 8–14.

[33] Henry Hoffmann et al. "Dynamic Knobs for Responsive Power-Aware Computing." In: *ACM SIGARCH Computer Architecture News* 39.1 (2011), pp. 199–212.

[34] Microchip Technology Inc. *MCP39F511N Datasheet*. [Online] last accessed 2022-05-31. URL: https://ww1.microchip.com/downloads/aemDocuments/documents/OTH/ProductDocuments/DataSheets/20005473B.pdf.

[35] NGINX Inc. *NGINX: Advanced Load Balancer, Web Server, & Reverse Proxy*. [Online] last accessed 2022-05-31. URL: https://www.nginx.com.

[36] Yeongjin Jang, Sangho Lee, and Taesoo Kim. "Breaking Kernel Address Space Layout Randomization with Intel TSX." In: *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. 2016, pp. 380–392.

[37] Vladimir Kiriansky et al. "DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors." In: *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '18)*. 2018, pp. 974–987.

[38] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution." In: *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP '19)*. 2019, pp. 1–19.

[39] Simon Kuenzer et al. "Unikraft: Fast, Specialized Unikernels the Easy Way." In: *Proceedings of the 16th European Conference on Computer Systems (EuroSys '21)*. 2021, pp. 376–394.

[40] Dayeol Lee et al. "Keystone: An Open Framework for Architecting Trusted Execution Environments." In: *Proceedings of the 15th European Conference on Computer Systems (EuroSys '20)*. 2020.

[41] Hugo Lefeuvre et al. "FlexOS: Making OS Isolation Flexible." In: *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS '21)*. 2021, pp. 79–87.

[42] Hugo Lefeuvre et al. "FlexOS: Towards Flexible OS Isolation." In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. 2022, pp. 467–482.

[43] *linux/bugs.c – Linux Kernel Source Tree*. [Online] last accessed 2022-05-22; Github. URL: https://github.com/torvalds/linux/blob/v5.14/arch/x86/kernel/cpu/bugs.c.

[44] *linux/stop_machine.h – Linux Kernel Source Tree*. [Online] last accessed 2022-05-22; Github. URL: https://github.com/torvalds/linux/blob/v5.14/include/linux/stop_machine.h.

[45] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space." In: *Proceedings of the 27th USENIX Security Symposium (USENIX Security '18)*. 2018, pp. 973–990.

[46] Michail Loukeris. "Efficient Computing in a Safe Environment." In: *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)*. 2019, pp. 1208–1210.

[47] *Many uses for Core scheduling [LWN.net]*. [Online] last accessed 2022-03-16. URL: https://lwn.net/Articles/799454/.

[48] *MDS - Microarchitectural Data Sampling – The Linux Kernel documentation (version 5.14)*. [Online] last accessed 2021-06-08. URL: https://www.kernel.org/doc/html/v5.14/admin-guide/hw-vuln/mds.html.

[49] Daniel Moghimi et al. "Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis." In: *Proceedings of the 29th USENIX Security Symposium (USENIX Security '20)*. 2020, pp. 1427–1444.

[50] Benjamin Oechslein et al. "OctoPOS: A parallel operating system for invasive computing." In: *Proceedings of the International Workshop on Systems for Future Multi-Core Architectures (SFMA '11)*. 2011, pp. 9–14.

[51] Oleksii Oleksenko et al. "Revizor: Testing Black-Box CPUs against Speculation Contracts." In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. 2022, pp. 226–239.

[52] *Page Table Isolation (PTI) – The Linux Kernel documentation (version 5.14)*. [Online] last accessed 2022-06-07. URL: https://www.kernel.org/doc/html/v5.14/x86/pti.html.

[53] Andrew Prout et al. "Measuring the Impact of Spectre and Meltdown." In: *Proceedings of the 22nd IEEE High Performance extreme Computing Conference (HPEC '18)*. 2018.

[54] Charles Reis, Alexander Moshchuk, and Nasko Oskov. "Site Isolation: Process Separation for Web Sites within the Browser." In: *Proceedings of the 28th USENIX Security Symposium (USENIX Security '19)*. 2019, pp. 1661–1678.

[55] Xiang (Jenny) Ren et al. "An Analysis of Performance Evolution of Linux's Core Operations." In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. 2019, pp. 554–569.

[56] Stephan van Schaik et al. "RIDL: Rogue In-Flight Data Load." In: *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP '19)*. 2019, pp. 88–105.

[57] Michael Schwarz et al. "ZombieLoad: Cross-Privilege-Boundary Data Sampling." In: *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. 2019, pp. 753–768.

[58] Martin Schwarzl et al. *Dynamic Process Isolation*. 2021. arXiv: 2110.04751 [cs.CR].

[59] *Static Keys – The Linux Kernel documentation (version 5.14)*. [Online] last accessed 2022-05-23. URL: https://www.kernel.org/doc/html/v5.14/staging/static-keys.html.

[60] *Sysbench Project*. [Online] last accessed 2022-05-31; Github. URL: https://github.com/akopytov/sysbench.

[61] AMD64 Technology. *Indirect Branch Control Extension - Revision 4.10.18*. [Online] last accessed 2022-06-08. URL: https://developer.amd.com/wp-content/resources/Architecture_Guidelines_Update_Indirect_Branch_Control.pdf.

[62] Graz University of Technology. *Meltdown and Spectre Website*. [Online] last accessed 2022-05-28. URL: https://meltdownattack.com/.

[63] The IETF Trust. *Internet Security Glossary, Version 2*. [Online] lsat accessed 2022-05-28. 2007. URL: https://datatracker.ietf.org/doc/html/rfc4949.

[64] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. "Simultaneous Multithreading: Maximizing on-Chip Parallelism." In: *ACM SIGARCH Computer Architecture News* 23.2 (1995), pp. 392–403.

[65] Paul Turner. *Retpoline: a software construct for preventing branch-target-injection*. [Online] last accessed 2022-06-07. URL: https://support.google.com/faqs/answer/7625886.

[66] Yao Wang et al. "SecDCP: Secure Dynamic Cache Partitioning for Efficient Timing Channel Protection." In: *Proceedings of the 53rd Annual Design Automation Conference (DAC '16)*. 2016.

[67] *wrk - a HTTP benchmarking tool*. [Online] last accessed 2022-05-31; Github. URL: https://github.com/wg/wrk.

[68] *x86 Feature Flags – The Linux Kernel documentation (version 5.14)*. [Online] last accessed 2022-05-23. URL: https://www.kernel.org/doc/html/v5.14/x86/cpuinfo.html.

[69] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. "A Software Approach to Defeating Side Channels in Last-Level Caches." In: *Proceedings of the 23th ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. 2016, pp. 871–882.