OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

INFORMATIK UND
MATHEMATIK

Bachelor's Thesis

How Spectre Mitigations Affect Real-Time Capabilities of Virtualized Mixed-Criticality Systems on Modern AMD Processors

Submitted by: Andrej Utz
Matriculation number: 3012095
Course of Studies: Computer Engineering / Technische Informatik

Supervised by: Prof. Dr. rer. nat. Wolfgang Mauerer
Ralf Ramsauer, M.Sc.
OTH Digitalisation Lab

Regensburg, August 9, 2019

Contents

Chapter 1
1 Introduction

With the disclosure of the Spectre hardware vulnerabilities the guarantee for data boundaries inside the modern microprocessors, and therefore their computational 'Root of Trust', was lost. The caches of a Central Processing Unit (CPU) can be abused to leak sensitive information [27]. Depending on the Spectre variant, a local attacker can extract data from an application, kernel or hypervisor. Because Spectre exploits speculative execution, a vital feature for high performance of modern CPUs, a fix needs to happen in the hardware design phase and will probably take years. Software fixes are required to mitigate those exploit in order to protect data that is processsed inside billions of vulnerable CPUs in the field. These mitigations have already been implemented for most common CPU architectures and operating systems (OSs). However, they often come at a performance cost of up to 25%, depending on the type of workload [16]. Since their integration into OSs, many benchmark results were published which show that cost in real world scenarios. Benchmarks mostly focused concentrated on consumer spaces or data centers, more critical environments, like industrial systems with real-time requirements, were mostly neglected.

With the rise of Industry 4.0 and Internet of Things (IoT), a significant number of machines – industrial, scientific or in public transport – require increasingly more interconnection and sophisticated control. Additionally, those machines many of those components require real-time capabilities. Integrating Human-Machine Interfaces (HMIs) introduces soft real-time hardware components, thus creating a mixed-criticality system. The ubiquity of microchips with Symmetric Multiprocessing (SMP) makes consolidation of hardware components with various criticality levels into a single System-on-a-Chip (SoC) possible by using virtualized environments. Beside efficiency, single-target development and reduced production cost, a virtualized system allows hardware partitioning and coexistence between a general purpose OS and bare-metal applications. The use of speculative execution to meet performance requirements, their safety role in everyday life and the openness due to inter-networking considerably mark them as excellent targets for attacks using Spectre. As such, the mitigation of those attacks by system software may be inevitable, but in doing so, the real-time capabilities, and therefore safety, is endangered.

This thesis evaluates the impact of the mitigations on real-time systems. In the next chapters we elaborate on

- the advantages of virtualized mixed-criticality systems,
- the Jailhouse hypervisor design and the challenges enabling it on AMD systems,
- the Spectre mitigations and resulting problems for mixed-criticality system,
- the methodology to quantify the mentioned real-time degradation and
- An evaluation of our results.

Chapter 2
2 Background

2.1 Modern Processor Complexity

Modern CPUs use various techniques to improve the efficiency and throughput. In turn, they may degrade their real-time capabilities, which is elaborated on in this Section.

2.1.1 Symmetric Multiprocessing

Symmetric Multiprocessing (SMP) combines multiple processing units – *cores* – in a single CPU. Their individual composition and behavior still corresponds to that of a single CPU, but they share some components and the system bus. It allows the programs true parallel execution and can increase overall system performance manifold. However, this also can introduce race conditions and non-deterministic behavior due to resource competition.

2.1.2 Power States

When not under full load, it is preferable for a CPU to reduce it's power consumption. To do so it employs different power states, called *C-states* [13]. The CPU will always try to enter a higher C-state to save more energy. During high utilization, it stays in C0, the operational mode. When the load drops, it will switch to C1 or even C6, depending on its power-save logic. Switching back into the C0 state, however, is not simple and will create a considerable latency. This severely impedes determinism in an interrupt reaction scenario, but is preventable, as the CPU can be forced to stay in C0.

2.1.3 Multilevel Caching

The access to random access memory (RAM) has a high latency and fetching a result may take many cycles, stalling the execution. To mitigate this, a cache hierarchy of multiple levels with increasing sizes and access times are utilized. Commonly these consist of three levels. The first two levels, *L1* (smallest, fastest) and *L2* are located at each core. The last level,

L3 (largest, slowest), is shared. Caches store the data and its memory location in *cache lines*. When loading data from RAM, first the lines in L1 are checked. If a line is found (cache-hit), it fetched into the register and the access is fast. On a cache-miss the next level is queried. When all caches miss, the data is loaded from RAM and will be saved it through all cache levels, reducing the round-about time of further fetches. However, as the cache sizes are limited, this may evict data from cache, increasing its access latency. This complex interplay requires careful consideration about the worst-case. Keeping cache locality in mind when designing the implementation, is a possible mitigation.

2.1.4 Speculative and Out-of-Order Execution

During execution all CPU instructions are processed in a pipeline of multiple stages. After an instruction have been fetched, the CPU will decode it, to "understand" what to do. On CISC systems these will be broken down into more primitive micro-instructions. This helps to execute them in parallel, as most CPUs nowadays are superscalar. To avoid hazards resulting from logical dependencies between instructions, they are reordered before execution. This is known as *Out-of-Order* execution.

When a conditional instructions is decoded, the CPU has to decide what it will execute, *branching* the pipeline. If the operands lie still in memory, waiting for them stalls the pipeline. To prevent this, a *branch predictor* speculates the outcome and selects a branch on which the pipeline will continue its operation speculatively, while the real result is being fetched. During it, micro-instructions in the pipeline are called *transient instructions*, as their changes are not committed to memory yet. If the prediction was right, the branch predictor "memorizes" it and the executing continues. In the event of a misprediction, the transient operations are reverted, the pipeline is reset and the execution begins in the correct branch anew.

2.1.5 Simultaneous Multithreading

Simultaneous Multithreading (SMT), also known as hyper-threading, is a technique to subdivide a CPU core further into logical units – *hardware threads*. The scheduling between these threads for competing resources is delegated to the core and results in a more intricate resource management [17]. Pipeline stalls in one thread can be used to run the other, improving performance and efficiency. But just as with speculative execution it is difficult to predict, when these will happen. Most platforms allow to disable SMT.

2.2 Virtualizing Mixed-Criticality Systems

Modern machines, be it a car, an industrial robot, or a rocket, use hard real-time and soft real-time components for their operation. As such they are called mixed-criticality systems. Hard real-time is required for safety guarantees in operational logic, which consist of deadlines to be fulfilled. Failure to do so may be followed by catastrophic consequences. In order to avoid them, the hardware and software must be strongly deterministic. In software this is maintained by utilizing a specialized real-time OS (RTOS).

Soft real-time is mostly found in control components for HMIs. HMIs handle untrusted inputs, manage access and can often have complex logic for visualization of the machine state. As their target hardware can vary, the OS must be versatile with device support, calling for the deployment of an General Purpose OS (GPOS). Such complexity naturally increases risk of failure. Because of that, soft real-time hardware units are spatially separated from safety units.

However, such separation is not without significant costs: each hardware unit needs customized software, interconnection, power and space. Due to the special requirements, they often have proprietary, closed design, forcing developers to use subpar tools.

2.2.1 Hardware Consolidation

While SMP processors are dominating the market, most of the cores are often idling as concurrency bears risks. This invites to consolidate multiple hardware of units various criticality into a single SoC. Each unit gets one or more dedicated CPUs cores assigned, allowing unimpeded operation for hard real-time units. The resulting system is more

- simpler: single target decreases maintenance by consolidating development resources; network infrastructure is reduced.
- cheaper: less hardware means also less costs; safety certification requires less steps.
- efficient: less power consumption, space utilization and weight.

Nevertheless, the migration of existing systems can be a challenging task. The reuse of existing software components supports migration efforts. Because their design assumes full control over underlying hardware, their environments still need spatial isolation and the access to indivisible I/O components must be managed.

2.2.2 Real-Time Virtualization

Virtualization is an established technique to subdivide and manage resources of a single hardware unit between multiple systems. This is achieved with two key elements:

- virtual machines (VMs): isolated environments where either an OS or bare-metal application can run, also called *guest*.
- virtual machine monitor (VMM): the managing software underneath, also known as *hypervisor*.

To maintain isolation between the guests and the host, early hypervisors employed special techniques with immense performance overhead [11]. With the introduction of hardware virtualization extensions (e.g. VT-x, AMD-V), this overhead was reduced.

Access to hardware can be partitioned and/or emulated. Depending on hardware support, direct, trap-free access to devices can be permitted.

When devices are partitioned, each gets assigned exclusively to a single guest. Ideally the hypervisor would only activate on access violations to investigate.

When emulating, a virtual device is created inside the VM, which simulates the desired functionality. The hypervisor manages its behavior and models its I/O inside the running system. Virtual network interfaces become part of a software defined network. Virtual CPUs (vCPUs) are scheduled as tasks in the host operating system. Figure 2.1 shows this, with KVM as an example.

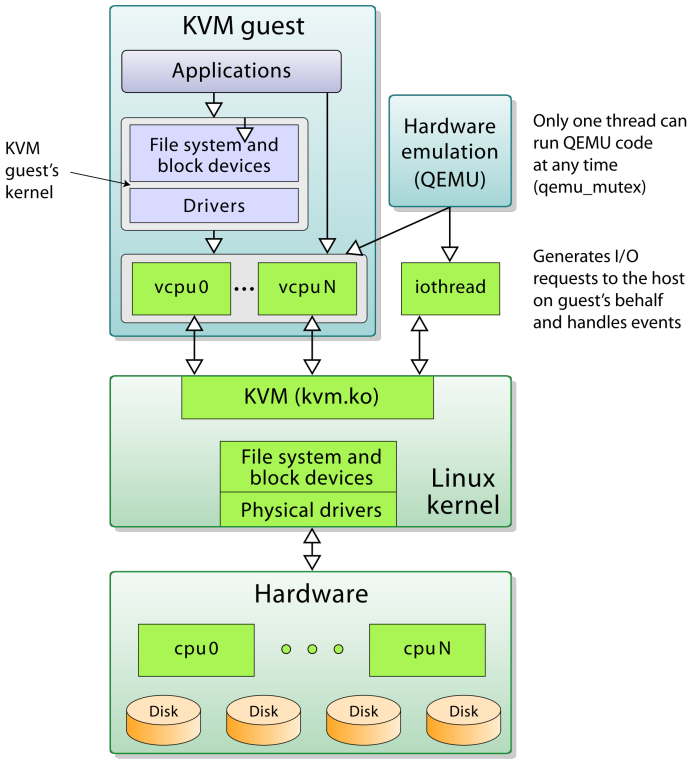

Figure 2.1: A high-level overview of the Linux Kernel Virtual Machine (KVM) + QEMU virtualization environment. (©V4711, CC BY-SA 4.0)

Depending on its type, the hypervisor is either a bare-metal system booting and running directly on native hardware (type 1) or an OS dependent userland application (type 2) [20]. With OS

support (i.e. KVM on Linux) the latter may reach the performance of the former, but its VMs still needs to be scheduled as any other application on the system. By design the former is the first to start after system firmware and acquires full hardware control. However this creates a dilemma. "With great power, comes great responsibility" and so a type 1 hypervisor has also to initialize the system by itself. Doing so requires it to have drivers for core system components, increasing its complexity.

Running the RTOS as a guest means, the hypervisors will inherit its real-time requirements. Between type 1 and type 2 hypervisor only the former would be a reasonable choice, due to type 2 having a design with non-deterministic aspects (e.g. a scheduler).

Depending on the system architecture and hardware support, we can choose both, full hardware control and simple code base without drivers. Such are some of the features included in the Jailhouse hypervisor, which is elaborated on in section 4.

2.3 Quantifying Real-Time Properties

Real-time properties depend on ability to react to an outside stimulus and produce a response in a timely manner. For hard real-time systems the most important property is to react deterministically, i.e. produce a response with the same delay every time. To quantify it, the latency between the expected response delay and the actual delay is measured.

But, our goal is to measure the system overhead and not our own payload. Thus the measurement scope can be reduced to the duration between interrupt request (IRQ) and application code reentry. This duration is called *wake-up latency* [12]. Unfortunately, timestamping the IRQ inside the system is impossible without inducing artificial overhead or costly hardware modifications.

A more feasible solution is constructing a test scenario, where the IRQ is scheduled and his point in time is known. This is achievable with high-precision timers (HPTs) on board. The key points in the scenario are as follows.

1. A continuous, HPT is used to query the starting time t_0.
2. A sleep period for a fixed duration d is entered. The application is unscheduled.
3. After the awakening, the HPT is queried again to save the actual wake-up time t_w.
4. The wake-up latency L is calculated as $L = t_w - (t_0 + d)$.

Depending on system complexity, this latency will vary on each test run. On microchips in a static environment running with a deterministic program this latency will always be nearly zero. Modern processors, as elaborated on in Section 2.1, have many non-deterministic factors. As

such, multiple runs are required to estimate the full latency distribution, also known as the *jitter*. If its variance is high, the worst-case latency needs to be determined. The number and sleep duration of the test runs to find the worst-case is under active research.

2.4 Spectre Vulnerability

The Spectre vulnerability is a local attack, that exploits speculative execution inside an CPU. In the event of a branch misprediction the caches are not reverted [14]. They can be used as side channel (or *covert channel*), to extract usually unaccessible target data from memory [27].

To fully mitigate Spectre, speculative execution would have need to be disabled, issuing severe performance penalties. But this is not even possible, as it is one of the cornerstones in high performance processors.

2.4.1 Caches as Covert Channel

Via the covert channel an attacker can find out the contents of the target data, without reading them directly. To do so, the channel needs to be constructed inside the cache [27]. For simplicity we extract only a byte each speculation phase. The channel shall consists of an contiguous array of 256 elements, the number if values a byte can hold. The byte value itself is to be used as the array index. Because the CPU only stores whole lines in cache, the size of each element needs to be scaled by the size of a line. Now each value of the byte is mapped to its own cache line.

Before the target data extraction, the array needs to be removed from the cache via `cflush` instruction or eviction by loading other data [27]. Then the target byte is used as an index to access an element. The CPU will reload that specific cache line. As mentioned in section 2.1.3, the access to cached data is far faster than fetching from RAM. Accessing the whole array, while measuring access times with the high-precision timer, will reveal which lines were loaded and which were not. The index of the loaded line can be mapped back to byte contents. A byte of target data was extracted from the cache.

Due to the shared nature of the cache, the extraction may not be reliable, when executed once [27]. Multiple tries while scoring the confidence of each one, however, will increase the certainty.

2.4.2 Attack Variants

Spectre has multiple variants [14]. They all exploit speculative executing, but have different targets. Due to constrains in resources, the scope will be limited to variants affecting AMD processors only.

Bounds Check Bypass (Variant 1)

A bounds check usually happens before accessing an array element or other arbitrary data inside the application address space. If it fails, the access is denied. However, if the condition for the check is not cached, it will be predicted and the pipeline will continue until its arrival. If the prediction allows access, the CPU will speculatively read targeted data. During transient execution, the covert channel in cache can be used to saved that data [14]. After the misprediction was detected and the transient operations are reverted, the cache will still contain the covert channel, allowing target data extraction. The bounds check is bypassed.

This attack works only, when the system allows access to the target data. Using arbitrary addresses across process boundaries will trigger a fault. As such this attack is only critical for sandboxed environments running untrusted code [14].

Mitigation Mitigation is accomplished by hardening the bounds checks. Before accessing sensitive data, usage of serializing instructions [1] guarantees to abort speculation. The pipeline is stalled until the condition check is complete. Unfortunately this but also imposes a performance penalty on each check.

Branch Target Injection (Variant 2)

Indirect branches are execution paths, that are not immediately visible for the CPU, e.g. a function pointer, which target function address needs to be loaded. While waiting for the load, the branch predictor can take a lookup in the *Branch-Target Buffer*, what function to call speculatively. With sufficient mistrainings this buffer can be modified to point into a malicious code segment [14]. Though still in transient operation, the attacker can execute instructions to build a cache covert channel and leak data.

This can also happen, when returning from a function into the caller. The return address lies on the stack, needs to be fetched from memory. This, again, can stall the pipeline and issue speculation. The CPU has very accurate predictor for return addresses, the Return-Stack-Buffer. However, with enough call stack depth, it can overflow and the CPU falls back to the Branch-Target Buffer. Together with branch target injection it allows speculative exploits via Return-Oriented Programming [24].

Mitigation Indirect calls or jumps can be turned into return trampolines (retpolines) to go to the target addresses [25]. The process is automatic, if the compiler supports it. Speculative execution paths under retpolines are trapped in an infinite loop to prevent any speculative execution. Additionally the `ibpb` instruction (short for Indirect Branch Prediction Barrier) can inhibit speculation by clearing the Branch-Target Buffer.

2.4.3 Expected Impact on Real-Time Latencies

The mitigations mostly inhibit speculation in specific code paths. Without it, the overhead depends on the locality of speculated values. In the best case (excluding registers) they are located in the L1 cache, continuing the pipeline almost immediately. In the worst case the pipeline is stalled until the value is fetched from RAM. However, if the measured code paths are identical, aside from mitigations, the impact should be a constant factor depending on the workload.

Chapter 3
3 Evaluation of Existing Measurement Tools

For the measurements, a tool meeting our requirements needs to be selected first.

3.1 Requirements

A base requirement for the tool is to run on the target OS, Linux. To measure the latencies with sufficient resolution and the lowest amount of noise, additional requirements must be met:

- CPU affinity: Support to tie each sleeping thread to exactly one CPU core, ensuring clear data bounds.
- FIFO scheduling: For the lowest possible scheduler overhead, the sleeping threads need to run under a real-time scheduling class. `SCHED_FIFO` from the POSIX-API [23] has a comparably simple, non time-slicing scheduler which will preempt current task if a higher priority task changes to a runnable state. As such a high priority must be specifiable, to ensure the sleeping tasks are scheduled to run immediately after wakeup.
- static memory: To count latency occurrences, the memory must be allocated and initialized before measurement begin and never be reallocated later. Initialization ensures actual allocation, because Linux will usually defer page allocation until a write access. Reallocation, especially for enlargement, will copy data. This results in cache pollution and therefore noise for other sleeping threads. The solution is a contiguous counting bucket list with a constant length $l = \lceil log_{10}(L_{max}) \rceil$ where L_{max} is the highest acceptable latency. This also means, that it cannot be written in language with automatic memory management, such as a garbage collector. These have complex, non-deterministic clean up cycles, where they suspend all threads to check for unused memory references.
- memory locking: Unpredictable latency can be caused by major page faults (e.g. when a page was evicted to swap). To prevent this, the process memory location must be locked inside the RAM. On POSIX systems this is possible with the function `mlockall` [19].

- nanoseconds resolution: An accuracy of microseconds is usually enough for most real-time application. However, other measurements on comparable Intel machines had shown sub-microsecond latencies. This prompted us to include nanoseconds resolution into the list of requirements.

Other optional, "nice to have" features include a histogram output in a easily parseable format and automatically disabling CPUs switch into C-states during the measurement.

3.2 Cyclictest

To measure interrupt latency jitter on a Linux system, *Cyclictest* from the *rt-tests* collection seems like the OSS tool to go. It was originally written by Thomas Gleixner, the principal maintainer of the Linux-RT patch, in 2005 and is now maintained by the Linux-RT community. After 14 years of maintenance, it can be said to be sufficiently reliable. As it is written in C programming language, we can trust the code to have no hidden side effects. Thread count, affinity and priority are set via command line parameters, as is memory locking, nanosecond accuracy and histogram output. Thread scheduler is automatically set to `SCHED_FIFO` and switch to C-states disabled by clearing `/dev/cpu_dma_latency` restoring it after program exit [2]. The histogram format is space-delimited table in plain text, with first column representing the time and the others latency occurrences of individual threads. At first glance Cyclictest seems to fulfill the stated requirements. However, after some initial testing, we detected some flaws in histogram output and CPU affinity specification.

3.2.1 Flaws

In the resulting histogram, Cyclictest creates superfluous time entries, where no latencies occurred. When measuring with nanosecond accuracy, this produces unnecessarily bloated histograms and requires additional parsing and a reduction step before statistical analysis. This can easily be prevented by filtering such lines in code before output.

The CPU affinity specification fails to take offline CPU cores into account and is also not sound from a user design perspective. To select the cores to measure on, Cyclictest provides two separate, yet logically bound parameters: `-affinity` specifies the cores to run on; `-threads` the number of sleeping threads. Each thread is assigned one core, but if their number is higher than the core count, which the parameter allows, the assignment continues in a round-robin fashion. A thorough search of the relevant literature yielded no results for a need to run more than one thread per core in this context. As `-affinity` could properly specify the number of threads, this would turn `-threads` obsolete. On a system with one or more offline cores,

mapped threads will (without any warning) get the affinity for all online ones instead, silently creating noise for other threads.[1] A solution would be to create threads in a 1:1 mapping from the intersection between specified and enumerated online cores, and to remove the `-threads` option entirely.

3.2.2 Legacy obstacles

As a tool ages, the number of users dependent on its behavior increases. Interface stability is important in software engineering. Patches degrading this stability are less likely to be accepted. This became evident as our first patch for filtering empty occurrences in histogram output was rejected. It created a breakage in *RTeval*, a complementing tool for histogram analysis. Because of this and the patchy legacy code impeding a proper fix for affinity specification, we abandoned the idea of fixing Cyclictest.

3.3 Jitterdebugger

Jitterdebugger [2] is a much newer and more minimalist alternative to Cyclictest. It is also written in C. The tools authors motivation behind its creation is driven by a similar frustration with Cyclictest, as in our attempts to fix it. Same as Cyclictest, it disables the switch to C-states, sets the sleeping thread scheduler automatically to `SCHED_FIFO`, and also locks memory. If an output directory is specified, the histogram and relevant system information (e.g. CPU info, kernel configuration, interrupt counts, etc.) will be placed there. The histogram itself is saved in JSON format.

At the begin of our evaluation Jitterdebugger lacked the ability to assign threads a core affinity and could only measure in microseconds. Due to the small and clear code base (563 SLOC[3]), compared to Cyclictest (2556 SLOC[4]), it was reasonable to implement these ourselves.

3.3.1 Switch to Nanoseconds

For systems with sub-microsecond latencies, the option to measure and provide results with nanosecond accuracy is a necessity. Although Jitterdebugger already samples in nanoseconds,

[1] The affinity selection was partially fixed after our evaluation, rejecting offline cores in the passed argument. Commit: `https://git.kernel.org/pub/scm/utils/rt-tests/rt-tests.git/commit/src/cyclictest?id=966b4eeda61b4c623a5ef423236dc710b31c1532`

[2] `https://github.com/igaw/jitterdebugger`

[3] Jitterdebugger commit 6e27c49143fe3fa4255a7d45476c557b020e34fa:
`$ tokei jitterdebugger/jitter{debugger.{c,h},utils.c}`

[4] rt-tests v1.3: `$ tokei rt-tests/src/{cyclictest,include}`

the latency is floored to microseconds before incrementing the occurrence counter. Additionally the length of the counter list is hard-coded to 1000, allowing only a range from 1 µs to 1 ms.

For switching to nanoseconds, the accuracy reduction was removed and the list length increased by a factor of 1000 to maintain the upper bound of 1 ms. This latter change also adds 6.8 MiB RAM usage per thread.

3.3.2 CPU Affinity Specification

Per default Jitterdebugger uses all cores with a 1:1 mapping for each thread. For the `-affinity` option we assume, that the ranges entered, represent a subset of not only active, but all logical cores. To specify theses ranges in a human friendly matter, we reuse the *cpu-list* format from the kernel [7, cpu-lists], as Cyclictest does. It consists of comma delimited, inclusive ranges of natural numbers, each representing a core ID e.g. `1-4,5,6,8-9`. The internal storage of the cpu-list is `cpu_set_t`, a bitset structure, where every bit represents a core, and part of the POSIX-API [9]. After parsing a cpu-list into `cpu_set_t`, its API is used to calculate the required sleeping thread count. During thread creation, each core number is checked against said `cpu_set_t` and skipped, if the check fails.

To ensure offline cores are taken into account, Jitterdebugger queries the kernel for online core numbers. This information is provided in cpu-list format by *sysfs*-API and located in `/sys/devices/system/cpu/online`. We can use it as a default affinity by parsing it into another `cpu_set_t`. If the user specifies a custom affinity set, it and the default are conjoined bitwise. If the resulting set has less cores, than the specified one, a warning is issued.

3.4 Summary

After the implementation of required futures and further minor improvements, the measurement results between Jitterdebugger and Cyclictest were compared. No considerable difference was found and Jitterdebugger was ready for operation in the test setup. Additionally, the affinity implementation was reviewed and accepted by the Jitterdebugger maintainer.

Chapter 4

4 Virtualization with Jailhouse

Jailhouse is a partitioning, Linux-based, OS-agnostic, hypervisor [22] which Open Source development was started by Jan Kiszka (Siemens AG) in late year of 2013 [10]. It supports three most widespread architectures: x86_64, ARM and ARM64. Partial guest support of its infrastructure was merged into Linux kernel in early 2018 [15], marking significant progress.

Contrary to many general purpose hypervisors, Jailhouse is designed specifically for virtualized mixed-criticality systems. Guided by simplicity as its core philosophy, Jailhouse has very minimalist design and only around 32 k source lines of code (SLOC)[1]. This eases security audits and safety certification processes, a key criteria for fast adoption. Resource overcommiting and scheduling are explicitly set as non-goals. The authors consider these popular features problematic in a real-time context [22] [26]. Jailhouse relies on a number of existing techniques in hardware, to keep hypervisor overhead to a minimum.

4.1 Static Partitioning

In common hypervisors, access to virtualized hardware in a VM invokes the hypervisor routines, increasing operation latencies. The approach of Jailhouse is different. Instead of virtualizing hardware resources, they are partitioned. To quote the authors [22]:

> [Jailhouse] transforms symmetric multiprocessing (SMP) systems into asymmetric multiprocessing (AMP) systems by inserting "virtual barriers" to the system and the I/O bus. From a hardware point of view, the system bus is still shared, while software is *jailed* in *cells* from where the guest software, so-called *inmates*, can only reach a subset of physical hardware.

Based on the assumption, that the physical hardware has no need to be shared between inmates during their runtime [22], each cell contains a subset of the system. The composition of these subsets consist some memory, indivisible peripherals specific to the cell, and a minimum of one CPU core.

[1] Jailhouse v0.11: `$ tokei jailhouse --exclude configs`

4.1.1 I/O Access Moderation

Almost all hardware control is done via memory-mapped I/O (MMIO) nowadays. The handling of devices and the exchange of data is done via reads or writes to memory addresses. This allows Jailhouse to employ the Memory Management Unit (MMU) as a warden. Accessing addresses outside of cell bounds would cause a fault (*trap*), calling the hypervisor to investigate [22]. In case of access violation, it will trigger a fatal error in the cell, terminating the inmate.

Unfortunately some devices often acquire addresses, which share a single page. Accessing them will also trigger a fault and call the hypervisor. It will analyze the infringing instruction and decide based on configuration whether to permit or deny it. However, access to fully owned pages will never trap, significantly increasing performance.

The described warding technique is utilized in the same fashion with the help of the IOMMU for direct memory access (DMA) based I/O streams originating from devices.

On x86 architecture, in addition to MMIO, port I/O (PIO) exists. It is an legacy device addressing, from times before MMIO came to be. But access moderation of PIO is still covered by the MMU.

Special care is given to the PCI devices handling. PCI devices are configured via MMIO registers in the PCI configuration space, where they also describe their class and their capabilities [8, Chapter 12]. The PCI capabilities are well specified and each can be enabled and configured via register access. One special capability is Message Signaled Interrupt (MSI). When assigning a device with MSIs to a cell, additional work is required to remap the interrupts. Other capabilities (e.g. power management) can be overreaching and influence devices of other cells. Therefore Jailhouse needs to track each device capability and allows only read-only access per default.

4.1.2 CPU Binding

To avoid scheduling, Jailhouse passes CPU cores directly into the cell, enabling nearly full control [22]. If the CPU supports it, specific IRQs are also remapped. This has huge advantage for real-time properties: inmates can acknowledge IRQs directly with no overhead. If the support is missing, as on ARM GIC v2 and v3, the hypervisor will function as intermediate handler and reinject the IRQ into the cells.

4.2 Configuration

The composition of all Jailhouse cells is described by binary blobs containing structs in the C-ABI format. They are created, respectively compiled, from editable C source files. This form was chosen to avoid parsing text or similar fragile formats. The binary data inside the blob is directly load into the hypervisor as is.

While this approach makes sense from the developers perspective, not everyone is savvy enough to understand configurations in C code, like in listing 4.1. We can assume, that configuration via C-source-editing is only a temporary step in the early development and that the options to configure Jailhouse will improve with time.

Listing 4.1: Heavily shortened Jailhouse root-cell configuration source.

```
2
struct {
        struct jailhouse_system header;
        __u64 cpus[1];
        struct jailhouse_memory mem_regions[39];
        struct jailhouse_irqchip irqchips[2];
        __u8 pio_bitmap[0x2000];
        struct jailhouse_pci_device pci_devices[37];
        struct jailhouse_pci_capability pci_caps[96];
} __attribute__((packed)) config = {
        .header = {
                .signature = JAILHOUSE_SYSTEM_SIGNATURE,
                .revision = JAILHOUSE_CONFIG_REVISION,
                .flags = JAILHOUSE_SYS_VIRTUAL_DEBUG_CONSOLE,
                .hypervisor_memory = { /* physical memory location */ },
                .platform_info = { /* platform description, PCI config, IOMMU location */ },
                .root_cell = {
                        .name = "RootCell",
                        .cpu_set_size = sizeof(config.cpus), // ...
                },
        },
        .cpus = { 0x00000000000000ff },
        .mem_regions = {
                {       /* MemRegion: 00000000-0009ffff : System RAM */
                        .phys_start = 0x0,
                        .virt_start = 0x0,
                        .size = 0xa0000,
                        .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE
                                | JAILHOUSE_MEM_EXECUTE | JAILHOUSE_MEM_DMA,
                }, // more memory regions...
        }, // more descriptor arrays...
};
```

To ease the creation of the root-cell configuration for the users, the command-line tools contain a generator. It will query the kernel for information, mostly from `/proc/iomem`, and generate initial configuration as C source file. While on ARM architectures, the device tree tables cover the whole system, this is not the same for PIO on x86 architectures. The PIO tables in the kernel (`/proc/ioports`) may often be incomplete. This leaves the user with the guessing game to detect where unsolicited port accesses come from.

4.3 Linux as Bootloader

Jailhouse avoids maintenance of countless drivers by using *deferred activation* [22]. Booting is a tedious endeavor, as many system components need to be initialized for reasonable operation. Linux is an extremely powerful kernel, having support for enormous number of devices [4] and countless contributors maintaining it. It enables Jailhouse to run on nearly any platform, as long the architecture is supported. Due to deferred activation and low-level hardware management, Jailhouse is not a Type-1 nor a Type-2 hypervisor, but rather a a mixture of both. [22]

4.3.1 Deferred Activation

When the system is fully booted, the hypervisor can be activated via its command-line tools. To do so, a static root-cell configuration, which reflects the system's topology, must be provided. Any device not specified there, is isolated from any system access. For activation, the helper kernel module loads the hypervisor into RAM and injects its code the CPU [22]. By utilizing hardware assisted virtualization, Jailhouse hijacks the CPU control from Linux. The running OS becomes encapsulated in the *root-cell*, as shown in Figure 4.1. Linux is able to continue operation, as the kernel still has access to hardware, But said access is now moderated according to root-cell configuration. Violations might crash the kernel and destabilize the whole system.

The deactivation of the hypervisors destroys all cells, if possible, and reverts the described activation steps, leaving the system in the same pristine state, as before.

4.4 Cell Management

Creation and destruction of non-root cells is managed in the root-cell via command-line tools. The non-root cell configuration has the same format, but different structure than a root-cell configuration. The "warden" and other global components (e.g. IOMMU, xAPIC) required for virtualization management, cannot be specified. As per rules of partitioning, the list of "divisible" components for the cell must be a subset of the root-cell [22].

Upon invoking the cell creation command, the Jailhouse kernel module will disengage that component subset from Linux. While devices in the PCI-bus support hot-plugging and their drivers are deinitialized accordingly, the access to other, bus-less MMIO devices will result in a fault. For such devices it is important to disable their drivers beforehand. As Linux supports hot-plugging of CPU cores, the cores designated for the cell are shut down before handover.

Now the cell is allocated and under construction inside the hypervisor. The cores are brought up again, reinitialized and parked. The MMU and IOMMU configured to trap the hypervisor

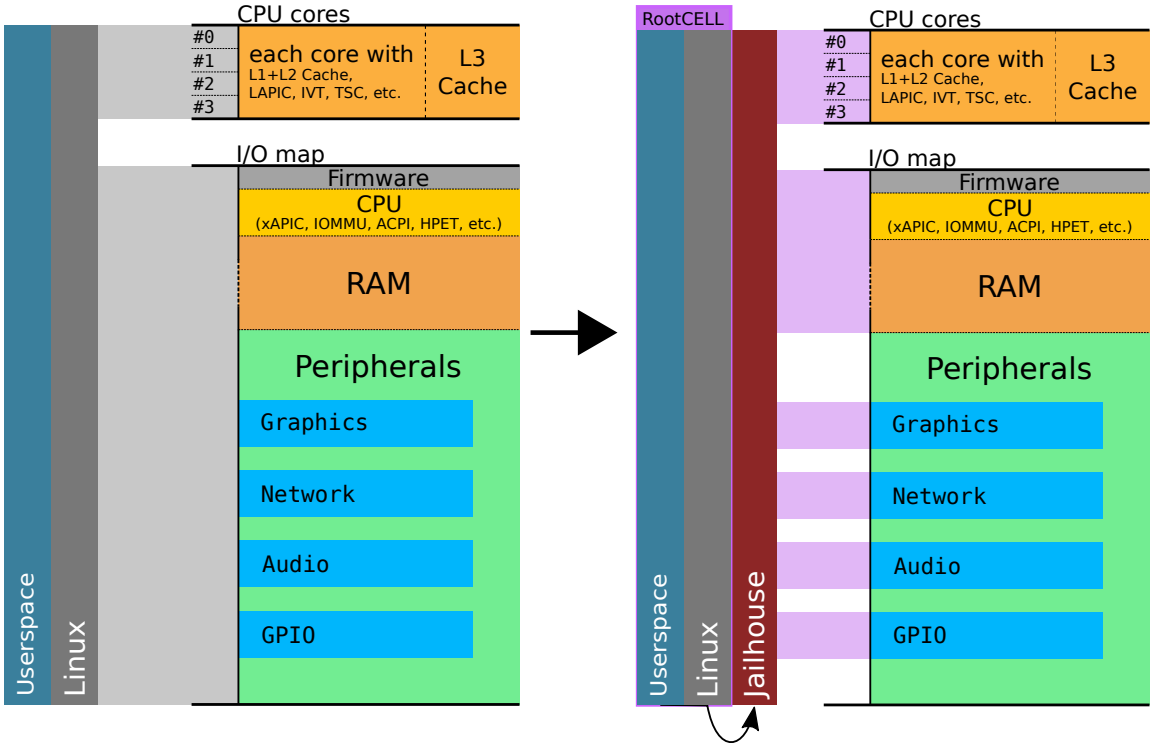

Figure 4.1: Linux activating the Jailhouse hypervisor. After activation the I/O access becomes restricted.

on access violations from the cores. The user needs to supply the inmate payload and start the cell. The latter will unpark the cores into the starting address of the inmate, starting its operation. The cell is now active.

And example of the result can be seen in Figure 4.2.

4.4.1 Linux in a Non-Root Cell

To boot the Linux kernel, the procedure is generally the same, but with key one difference: Linux needs system information to boot and will probe for central components (e.g. IOMMU, xAPIC, etc.), only which the hypervisor has access to. This will cause an immediate fault. Fortunately, Linux has Jailhouse non-root cell support since version 3.12. If activated, the kernel will detect the hypervisor via emulated `cpuid` instruction and read the system information from the `setup_data` block, provided by Jailhouse. With an supplemented *initramfs* archive[2] during cell start, the linux non-root cell will stay active.

[2] A small userland filesystem loaded into RAM, which has the necessary `init` binary for continuous system operation and usually some user tools.

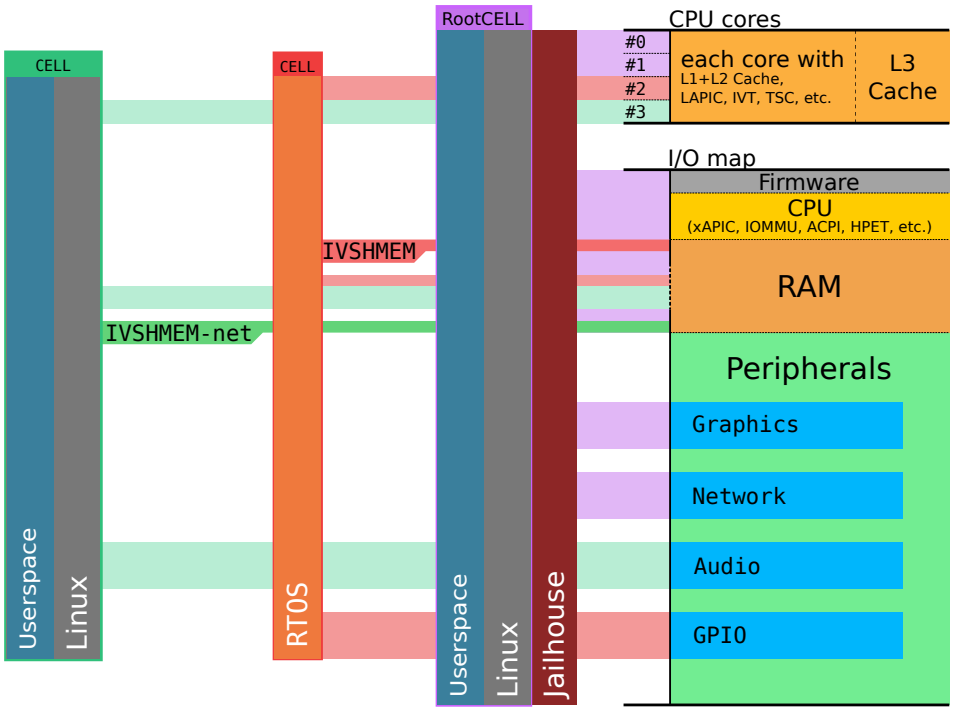

Figure 4.2: Multiple inmates running in Jailhouse on a single system. Bars indicate device control. IVSHMEM channels are connected to the root-cell.

4.5 Inter-Cell Communication with Inter-VM Shared Memory

Sometimes the inmates might need to exchange information through the isolated barriers. Jailhouse allows this via shared memory regions and a signaling interface based on interrupts [22]. For controlled data exchange, the *ivshmen* virtual device can be used. It is implemented as a virtual PCI device due to already existing PCI infrastructure in Jailhouse as described in Section 4.1.1. Also the emulation of a PCI bus for system without a real one is trivial. Both cell each require a ivshmen device with the same BDF, memory location, and memory size to establish a link. IVSHMEM is generic enough to allow any format of data change. On Linux the IVSHMEM-net driver realizes IP based communication between cells.

Chapter 5
5 Evaluation

As stated in the topic, the aim is to measure the effects of Spectre mitigations on the real-time capabilites of virtualized real-time systems. For that, a Spectre affected system is required.

5.1 Test System

The measurements are executed on an *AMD Ryzen 2700X* processor running on *Gigabyte B450 AORUS M* mainboard with firmware version F5. Is is affected by Spectre Variant 1, Variant 2 and Variant 4.

Linux with `PREEMPT_RT` patch, v4.19.50-rt22 [1], is used as the system OS. It includes full mitigations for the aforementioned vulnerabilities. For virtualization we use Jailhouse v0.11 with some [2] patches for our system.

5.1.1 Latency Noise Reduction

To ensure only the impact of the Spectre mitigations is measured, external latencies need to be reduced as far as possible. Linux has various configuration options to achieve that. The first CPU core, *CPU0*, will act as the central control point of the system.

Core Isolation

Task schedulers will utilize all CPU cores when distributing load from processes. On measured cores this might introduce unwanted latencies and increase the jitter. Linux provides the `isolcpus=` kernel parameter to isolate specified cores from general scheduling [7]. `isolcpus=1-7` are passed at boot time to restrict the processes to CPU0 only. To run on isolated cores, the process has to explicitly set a CPU affinity and request permission from the kernel, which Jitterdebugger does.

[1] `https://github.com/siemens/linux/tree/jailhouse-enabling/4.19-rt`
[2] `https://github.com/lfd/jailhouse/tree/ryzen`

Interrupt Remapping

Unsolicited IRQs are one of the main reasons for latency noise in the measurement. During IRQ acknowledgment, task scheduling is impeded, possibly delaying the wakeup from Jitterdebugger and increasing measured latency [18, chapter 14, page 874]. As with the processes, IRQ servicing is also distributed among cores. By passing the `irqaffinity=0` parameter to the kernel during boot time, all IRQs are remapped to CPU0 [7].

Disabling Paging Optimization

Linux will continuously optimize physical page utilization in the RAM during system runtime. This is done with the techniques of Page Compaction and Page Migration. Page Compaction reduces memory fragmentation by grouping free and occupied pages [5]. Page Migration improves memory latency, by moving moving them near processor, where they are used [6] While these techniques improve performance in the long run, their asynchronous, transparent operations create hidden latencies. Memory regions, where thread stacks of Jitterdebugger reside, might be in migration during a wake-up, delaying their reentry. To disable compation and migration, the `CONFIG_COMPACTION` and `CONFIG_MIGRATION` build options must be unset.

Hardware Latencies

To ensure no anomalous latency is created by the hardware, the *hwlat* (short for Hardware Latency Detector) tracer was employed. Linux has an extensive tracing subsystem with various tracers for debugging purposes. The hwlat tracer specializes in detecting hardware induced latencies [3]. When running, it will busy-wait on one of the CPU core, with interrupts disabled, for a specified duration by polling the CPU Time Stamp Counter. Any gap in time indicates a hidden delay on hardware level. The tracer will continue by switching to the next core and repeat the detection.

As preparation for the tests, we ran the detector for about 48 h. No hardware latency was found.

5.2 Challenges of the Jailhouse Activation

As Jailhouse has to yet reach a stable release, the setup proved to be challenging.

To debug Jailhouse without a hardware debugger, a serial interface on the target system connected to an external machine, is of paramount importance. Jailhouse logs informative

(and debug) messages into a ringbuffer and on the serial port. However, the former cannot be read, when the kernel has panicked because of a fatal error in the hypervisor. With the build option `CONFIG_CRASH_CELL..._ON_PANIC`, Jailhouse will set the instruction pointer to 0, when a fatal error occurs. This will force Linux inmates to panic immediately, dumping a stack trace. This simplifies search for responsible code paths inside the kernel.

5.2.1 Configuration Generation

The configuration generator for x86 platform has still room for improvements. Currently the PIO permission list is static not permissive for non-PCI devices (with minor exceptions). Due to this, the root-cell will crash most of the time, when activating Jailhouse with the initial configuration. Users then have to add devices from `/proc/ioports` manually and calculate the right bitmask, to permit access.

This step can be automated in the generator. By reading `/proc/ioports` itself, it can construct a PIO list, which reflects the whole topology of the system. This would ease manual adjustment of the configuration. However, in the long run a friendlier user interface is preferable.

5.2.2 AMD Specific

While trying to enable Jailhouse on an AMD machine, we encountered some challenges. This is probably due to the resource dependent development focus on Intel platforms.

Unknown Instructions

When the MMU traps the hypervisor, the instruction register needs to be parsed to extract the address. If the instruction is unknown, the execution will be terminated. But due to the nature of CISC systems, nearly any instruction can store or load values to memory or MMIO. That is why Jailhouse only supports the most common instructions. The list is expanded as needed.

As some of the instructions on the AMD system were unknown, one of steps to enable Jailhouse was to expand the instruction parser.

5.3 Test setup

To detect and quantify the impact of the Spectre mitigations, we compare the worst-cases of following 3 variables: mitigation, workload, virtualization. As each of them are binary in nature, to create a combined comparison matrix, one test run has 2^3 different measurements. To have

sufficient confidence in the results, each measurement lasts for 180 min. Only CPU cores 4, 5, 6, 7 are used, as they had the least anomalous worst-case latencies.

Jitterdebugger is running with a priority of 98, to force immediate scheduling on time. The tool was started with the following parameters: `--duration 180m --affinity 4-7 --priority 98 --output <..>` The last one is needed to enable the histogram output.

To induce the worst-case latency, the same CPU cores are stressed with various workload types. For that we use *stress-ng* [3] (v0.10.00) benchmark suite. To cover most of the non-deterministic factors from Section 2.1, the workload are kept diverse by utilizing various stressors [4]. All stressors run sequentially, with an equal amount of running time each. The specific stress-ng parameters are `--job stressor-list.job --sched fifo --sched-prio 30 --taskset 4-7`

For measurements without Spectre mitigations, the system was booted with the `mitigations=off` kernel parameter passed at boot time [7].

5.4 Results

First a baseline latency of the system, without any mitigations or virtualization, is established, as seen in Figure 5.1.

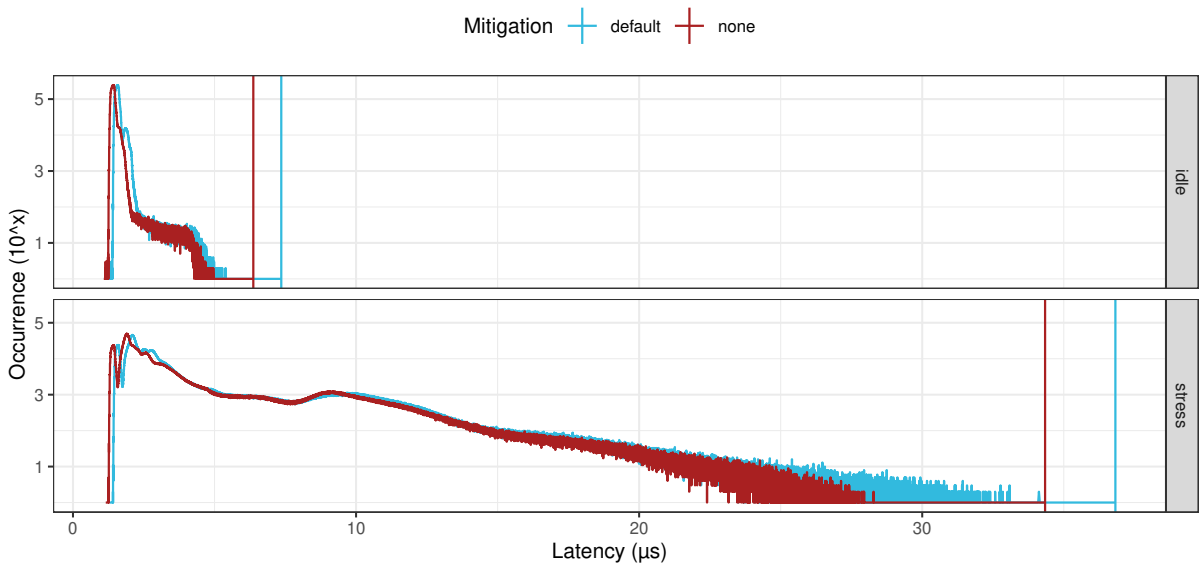

Figure 5.1: Spectre mitigations overhead on bare-metal system.

[3] `https://github.com/ColinIanKing/stress-ng`

[4] Used stressor list: `cpu, cache, sem, switch, itimer, sigfd, timer, vm, full, qsort, bsearch, hsearch, heapsort, mergesort, tsearch, memcpy, lsearch, memfd, mincore, remap, rmap, tlb-shootdown, zlib, str`

As expected, the worst-case is higher in the stress scenario, due to the CPU complexity. Of all stressors, `vm` was observed to significantly increase the upper bound. It stresses the RAM access (and therefore the caches), by creating mebibytes of memory allocations and data writes. This demonstrates, that the cache locality is a critical factor for real-time properties.

The mitigation overhead does exist, but is probably negligible for most real-time requirements. The standard deviation of all proportions between mitigated and non-mitigated extrema is only 1.10 µs, indicating a constant overhead factor. This confirms the prediction in Section 2.4.3.

Enabling and measuring on Jailhouse provided some unexpected results. Common latency histograms usually show one jitter accumulation, but Figure 5.2 differs in that aspect by depicting two accumulations.

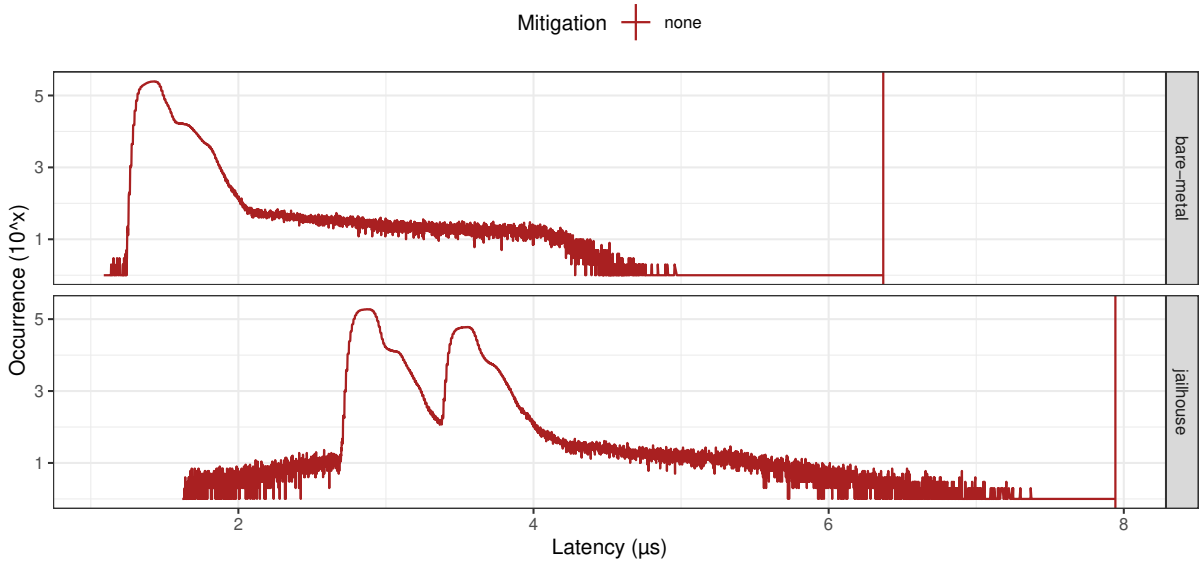

Figure 5.2: Virtualization overhead under Jailhouse with an anomaly.

Subdividing it by CPU cores, produces Figure 5.3. For an unknown reason the maximum latency of the first core does not correspond to the regularity of the rest. Instead it has a constant offset by a factor of 1.23. The average, however, still follows the prediction. This raises the question of whether hidden latency factors would emerge under certain conditions.

When adding mitigations and workload into the picture, as shown in Figure 5.4, the worst-case of two different cores spikes in opposite mitigation cases. Also the second core has a better worst-case with mitigations, than without. This result clearly challenges the reliability of the Jailhouse hypervisor on the AMD system, as comparable results on other platforms are more predictable [21].

The last Figure 5.5 displays a comprehensive comparison of all variables. Unfortunately the measurement anomaly in the last case prevents a reasonable answer to the thesis question.

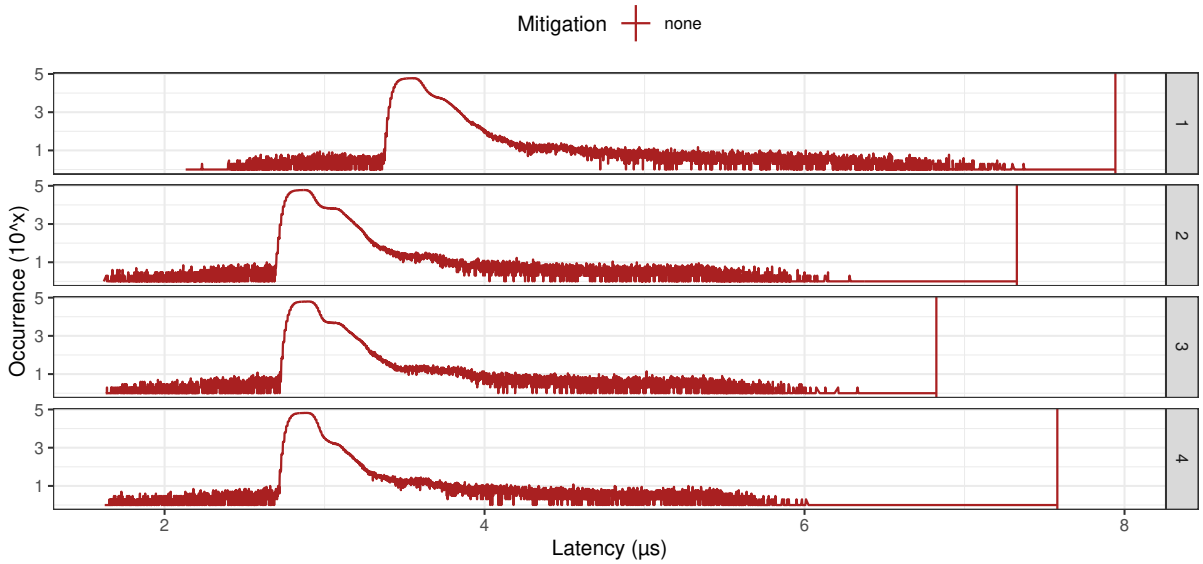

Figure 5.3: Latencies of individual CPU cores under Jailhouse without workload.

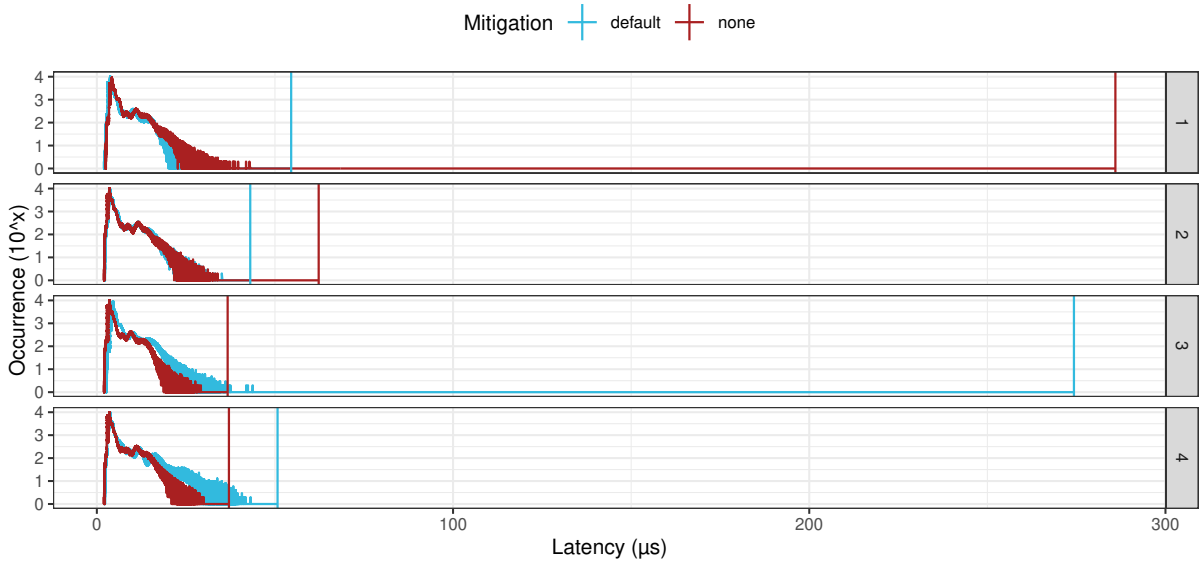

Figure 5.4: Latencies of individual CPU cores under Jailhouse during workload including mitigations.

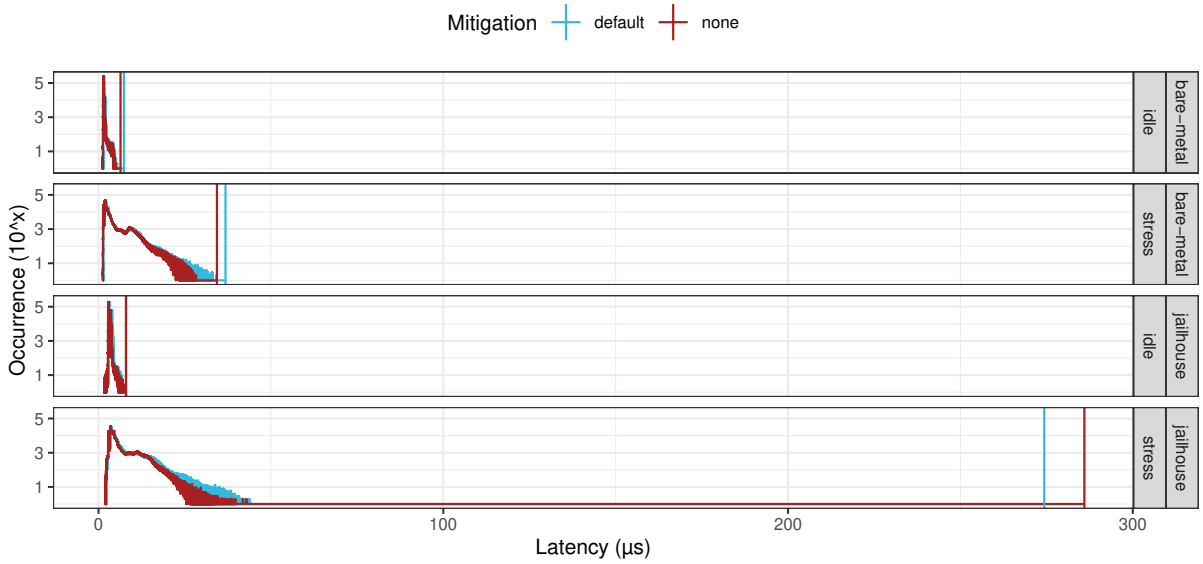

Figure 5.5: Comprehensive latency jitter including Spectre mitigations on a virtualized real-time system running on AMD Ryzen.

Chapter 6
6 Conclusion

Our goal top measure the impact of the Spectre mitigations on a virtualized mixed-criticality system running on a modern AMD system, was partially achieved. We elaborated on solutions to virtualized real-time environments and selected the Jailhouse hypervisor as the most promising, due to its simplicity. A test setup, based on existing knowledge and solutions was constructed. While doing said solutions were not only improved for our purposes, but also for the greater good, as common for all Open Source Software. Additionally, we outlined ways to reduce general system latency in the Linux system.

However, due to unreasonable measurement results in virtualized environment, a definite conclusion remains unfeasible. Subpar Jailhouse support for AMD systems compared to similar platforms of other vendors, may be one of the main factors behind it.

The others may actually come from the opposite direction. The consolidation of hardware with the help of a static partitioning hyper is a promising approach to create reliable mixed-criticality systems. If hardware vendors start direct more focus towards it, it might also become viable.

Acronyms

CPU Central Processing Unit

CISC Complex instruction set computer

DMA direct memory access

FIFO First-In, First-Out

GPOS General Purpose OS

HMI Human-Machine Interface

HPT high-precision timer

IoT Internet of Things

IRQ interrupt request

JSON JavaScript Object Notation

MCS mixed-criticality system

MSI Message Signaled Interrupt

MMIO memory-mapped I/O

MMU Memory Management Unit

OS operating system

OSS Open Source Software

PIO port I/O

RAM random access memory

RTOS real-time OS

SLOC source lines of code

SMP Symmetric Multiprocessing

SMT Simultaneous Multithreading

SoC System-on-a-Chip

VM virtual machine

VMM virtual machine monitor

Bibliography

[1] Inc. Advanced Micro Devices. *Software techniques for managing speculation on AMD processors*. 2018-07.

[2] *Are hardware power management features causing latency spikes in my application?* Red Hat, Inc. 2013-03-20. URL: https://access.redhat.com/articles/65410 (visited on 2019-08-03).

[3] The Linux kernel development community. *Hardware Latency Detector. The Linux Kernel Documentation*. URL: https://www.kernel.org/doc/html/latest/trace/hwlat_detector.html (visited on 2019-08-08).

[4] The Linux kernel development community. *Linux allocated devices. The Linux Kernel Documentation*. URL: https://www.kernel.org/doc/html/v4.19/admin-guide/devices.html (visited on 2019-08-06).

[5] The Linux kernel development community. *Memory Management - Concepts Overview. The Linux Kernel Documentation*. URL: https://www.kernel.org/doc/html/latest/admin-guide/mm/concepts.html#compaction (visited on 2019-08-08).

[6] The Linux kernel development community. *Page migration. The Linux Kernel Documentation*. 2016-03-28. URL: https://www.kernel.org/doc/html/latest/vm/page_migration.html (visited on 2019-08-08).

[7] The Linux kernel development community. *The kernel's command-line parameters. The Linux Kernel Documentation*. URL: https://www.kernel.org/doc/html/v4.19/admin-guide/kernel-parameters.html (visited on 2019-08-06).

[8] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. 3rd ed. O'Reilly Media, 2009-02.

[9] "CPU_SET(3)". In: *Linux Programmer's Manual*. 2019-03-06.

[10] Oliver Diedrich. *Jailhouse: Noch'n Hypervisor für Linux*. German. Heise Medien GmbH & Co. KG. 2013-11-19. URL: https://heise.de/-2050259 (visited on 2019-07-20).

[11] Ulrich Drepper. "The Cost of Virtualization." In: *Acm Queue* 6.1 (2008), pp. 28–35.

[12] Carsten Emde. "Long-term monitoring of apparent latency in PREEMPT RT Linux realtime systems". In: *RTLWS12* (2010).

[13] Taylor IoT Kidd. *Power Management States: P-States, C-States, and Package C-States*. Intel Corporation. 2014-04-17. URL: https://software.intel.com/en-us/articles/power-management-states-p-states-c-states-and-package-c-states (visited on 2019-08-03).

[14] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. 2019.

[15] Michael Larabel. *Jailhouse Guest Support To Be Included With Linux 4.16*. Phoronix Media. 2018-01-29. URL: https://www.phoronix.com/scan.php?page=news_item&px=Jailhouse-Guest-Plat-Linux-4.16 (visited on 2019-07-22).

[16] Michael Larabel. *Spectre/Meltdown/L1TF/MDS Mitigation Costs On An Intel Dual Core + HT Laptop*. Phoronix Media. 2019-05-21. URL: https://www.phoronix.com/scan.php?page=news_item&px=Spec-Melt-L1TF-MDS-Laptop-Run (visited on 2019-07-25).

[17] Deborah T Marr et al. "Hyper-Threading Technology Architecture and Microarchitecture". In: *Intel Technology Journal* 6.1 (2002).

[18] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Wiley Publishing, Inc., 2008.

[19] "mlockall(3P)". In: *POSIX Programmer's Manual*. 2013.

[20] Gerald J. Popek and Robert P. Goldberg. "Formal Requirements for Virtualizable Third Generation Architectures". In: *Commun. ACM* 17.7 (1974-07), pp. 412–421. ISSN: 0001-0782. DOI: 10.1145/361011.361073.

[21] Ralf Ramsauer, Jan Kiszka, and Wolfgang Mauerer. *Spectre and Meltdown vs. Real-Time: How Much do Mitigations Cost?* The Linux Foundation. 2013-11-19. URL: https://youtu.be/nqU4j2M_Ul4 (visited on 2019-07-25).

[22] Ralf Ramsauer et al. "Look Mum, no VM Exits! (Almost)". In: *CoRR* abs/1705.06932 (2017). arXiv: 1705.06932. URL: http://arxiv.org/abs/1705.06932.

[23] "sched(7)". In: *Linux Programmer's Manual*. 2019-03-06.

[24] Hovav Shacham et al. "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)." In: *ACM conference on Computer and communications security*. 2007, pp. 552–561.

[25] Paul Turner. *Retpoline: a software construct for preventing branch-target-injection*. 2018. URL: https://support.google.com/faqs/answer/7625886 (visited on 2019-08-06).

[26] Steve Vestal. "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance". In: *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*. IEEE. 2007, pp. 239–243.

[27] Yuval Yarom and Katrina Falkner. "FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack". In: *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 2014, pp. 719–732.

Declaration

1. Mir ist bekannt, dass dieses Exemplar der Bachelorarbeit als Prüfungsleistung in das Eigentum der Ostbayerischen Technischen Hochschule Regensburg übergeht.
2. Ich erkläre hiermit, dass ich diese Bachelorarbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Regensburg, August 9, 2019

Andrej Utz