Principles and Practice of Hypervisor-Assisted Real-World Hardware Partitioning

Ralf Ramsauer*, Jan Kiszka†, Daniel Lohmann‡ and Wolfgang Mauerer*†
*Technical University of Applied Sciences Regensburg
†Siemens AG, Corporate Technology, Munich
‡University of Hanover
ralf.ramsauer@othr.de, jan.kiszka@siemens.com, lohmann@sra.uni-hannover.de, wolfgang.mauerer@othr.de

***Abstract*—Industrial real-time appliances are often driven by oversized general purpose commercial of the shelf (COTS) hardware components. Cost pressure and minimality of the system are only subordinate design factors. This suggests a natural consolidation of multiple workloads, in particular workloads of mixed criticality, onto a single system. Safe isolation is a quintessential requirement for this purpose, but isolation guarantees have recently been endangered by a whole class of attacks on speculative execution of CPUs, and will continue to be threatend by similar imperfections.**

Nevertheless, the availability of unused hardware resources allows for shifting traditional isolation primitives from scheduled tasks with OS-assisted, moderated abstract shared hardware access towards unmoderated, independent computing domains that run on strictly isolated hardware segments.

In this paper, we extend and refine well-known general requirements for virtual machines towards *ideal hardware partitioning*, and establish necessary criteria for hardware and software to achieve such a state. We discuss technical hardware issues that currently prevent zero-trap designs in current real-world systems of realistic complexity.

We discuss the implementation of a partitioning hypervisor that supports three popular architectures with a focus on the absence of hypervisor interaction. We show that the design protects against cross-domain side channel attacks, and inherently maintains real-time capabilities on real-world hardware suitable for use in industrial deployments.

I. Introduction

Industrial real-time control systems are often built by extending general purpose commercial off the shelf (COTS) hardware components to reduce development effort in time and cost by maximising the re-use of existing solutions. The approach is commonly taken in many industrial domains, for instance automation and control systems [19], civil infrastructure projects [12], medical appliances [20] or robotics [32].

The approach is beneficial if flexibility in system capabilities is more important than potential reductions in cost that can be achieved by mass-producing tailored devices that precisely satisfy requirements, but usually never exceed them. Such scenarios often appear, for instance, in the automotive industry, but are rarely applicable to low-volume domains like medical appliances, industrial control, or even home automation.

Currently, CPUs with multiple physical (and virtual) cores are a de-facto standard in modern COTS hardware for non-microcontroller appliances, and their specifications and capabilities often considerably exceed the least demand for a given set of requirements. Consequently, most systems provide unused, excess hardware resources that can be used to integrate tasks.

Systems of increasing complexity and software intensiveness need to deal with workload that contain tasks at different levels of criticality; the resulting scenarios have received substantial attention during the last decade [4], and the *conceptual* advantages and disadvantages of the many possible approaches to build such systems are well researched.

One particular scientific focus of analysis for classically tailored embedded systems is on schedulability, fault tolerance or optimal work balancing to achieve deterministic and an optimum utilisation of the hardware. Minimum system requirements determine the most cost-effective choices for the hardware. For high volume systems, it is not unusual that special-purpose CPUs or SOCs are designed to satisfy very specific use cases, which then necessitates intensive, software-moderated sharing of resources. With few exceptions [9, 29], this resource management is typically implemented by operating systems.

A strong industrial requirement for real-world systems is that unwanted, unintentional interference between domains of different criticality must be absent, and that this absence has to be certified by either formal or informal criteria. The class of recently discovered attacks on speculative execution [33, 28, 40, 42, 25, 22, 5] that have not only received substantial academic consideration, but have even reached the attention of the general public, highlights the risks of hardware resource sharing, particularly when workloads of mixed criticality are scheduled on the same physical execution units. All variants of the above-mentioned speculative execution attacks are, roughly speaking, side-channels on speculative execution of CPUs sided by timing attacks on CPU caches [14, 43, 30]. They open a covert channel that can be used to leak confidential data between payloads of different criticality by the exploitation of fundamental CPU primitives. This violates or subverts many guarantees that are given by formerly trusted hardware units on which architectures of mixed-criticality solutions usually rely upon.

To mitigate the attacks, system software needs to implement computationally expensive countermeasures, assisted by

This work was partly supprted by the German Research Council (DFG) under grant no. LO 1719/3-1

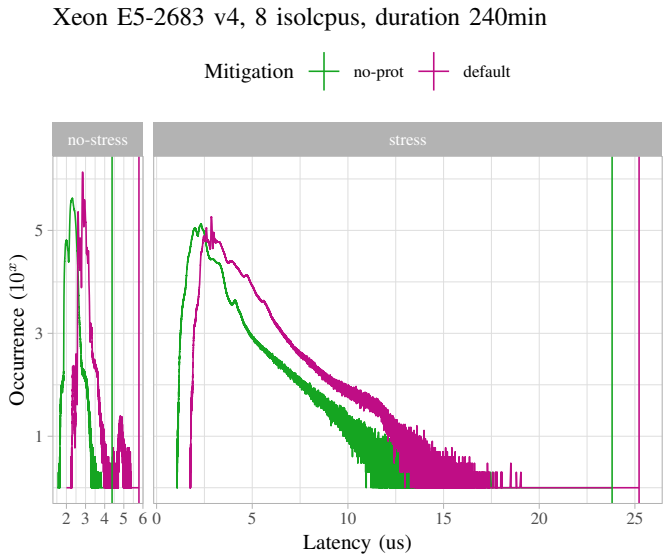

Figure 1. Illustration of the influence of mitigations against speculation attacks on determinism and response latencies of a Preempt-RT extended Linux based system. Left side: without additional system load (besides real time tasks). Right side: system is stressed by non real time load.

changes in CPU microcode. To quantify the cost, consider the measurements in Figure 1 that compare time determinism of a Linux kernel with Preempt-RT [24] real-time extensions as measured by the standard cyclictest testbench [13] for a kernel configuration with and without protection against Spectre and Meltdown attacks. The approximate increase in maximum (about 3 us) and average latency (about 5us) provides a natural "cost budget" for the overhead inflicted by said protections that can alternatively be invested into hypervisor respective partitioning overhead (it is self-evident that a system operating at maximal capacity will not be able to satisfy its original constraints once such countermeasures wil be in place, which gives additional justification to the use of over-provisioned hardware).

We have already remarked that many real-world systems do not operate at the brink of their capacity, and will retain the ability to appropriately respond to events even in the presence of Spectre-type mitigations. However, as we will discuss in this paper, the partitioning of systems is an architecturally more attractive alternative that can handle both, the inadvertent establishing of covert side channels and the safe coexistence of workloads of mixed criticality at the same time. This benefits, for instance, existing certified industrial codes that can only be modified at the expense of re-certification, which is both substantial in terms of monetary investment and required time to market: Instead of ensuring protection by adding explicit countermeasures to the code, it is run inside a isolated partition. Virtualisation technologies of modern CPUs provide mechanisms for strict and full isolation of computing domains [39, 17, 35, 41], and the overhead (usually caused by lack of hardware capabilities and imperfections) imposed by the cost of partitioning only marginally differs from the cost of mitigations. Since partitioning provides additional possibilities to system architects, we perceive the approach to be a preferable solution as compared to only rectifying erratic CPU behaviour.

An important industrial requirement on real-world that is that it must be possible to guarantee (under a reasonable definition of assurance) that *partitioning* implies *freedom of interference* between the partitioned cells. Not only on partitioned, but also on conventionally scheduled systems, the surface of potential cross-domain interference is determined by the degree of interaction between different computing domains. This includes interactions between tasks, tasks and operating systems, or operating systems and an underlying hypervisor.

We discuss (contributions to architecture, implementation, measurement and verification of) a partitioning industrial open source hypervisor, that allows us to build real-world systems that offer effective interference guarantees derived from hardware partitioning. Motivated by the reduction of hypervisor interaction, low latency and minimal overhead, our goal is to implement a *zero-trap partitioning hypervisor*. The resulting system maintains real-time capabilities by design, and completely eliminates OS/guest-hypervisor interactions. Virtualisation technologies are used to statically and exclusively assign hardware resources to computing domains to achieve strict and safe isolation of computing domains.

While contemporary CPUs provide sophisticated virtualisation extensions, they still cannot provide suitable interfaces to implement a trap-free approach. Consequently, we elaborate on widely underestimated hardware requirements necessary to implement hardware partitioning without hypervisor interception. Our approach presents a solution to enable safe coexistence of workloads of mixed criticality on a single system for many general purpose use cases. Furthermore, we explain how our architectural decisions reduce the attack surface for cross-domain low-level hardware attacks such as Spectre.

II. The Jailhouse Approach

A common industrial requirement is to safely run real-time workloads of mixed criticality on multicore systems [3] aside Linux. Typically, more CPUs than different workloads are available. Therefore, critical tasks can be exclusively assigned to dedicated CPUs, and the availability of Linux (and its feature-rich ecosystem) allows for running uncritical tasks on the remaining CPUs.

One possibility to realize this conceptual design is to purely rely on Linux-based isolation mechanisms. The SIL2Linux approach uses the PreemptRT [24] kernel extension to add real-time capabilities to the Linux kernel. It is combined with static workload-based CPU affinities for pinning tasks to otherwise isolated CPUs, control groups (cgroups) and and seccomp for isolation, as well as bank aware memory allocation [44] to reduce interferences from memory operations. The main assumption of this approach is that the Linux kernel primitives

provide a sufficient level of isolation. Yet, all computing domains share the same kernel: a critical system failure in any computing domain of the system leads to an overall system failure. Hence, parts of the system software (including the kernel and parts of the userspace) must be qualified when the system undergoes a formal certification. Given the large size of all involved software components, we believe that this is a challenging task.

Embedded virtualisation is an alternative approach. Domains of mixed criticality run as guests of a hypervisor, including Linux. This approach is, for example, implemented by [8, 38, 18].

Static hardware partitioning is a special case of embedded virtualisation that exclusively assigns hardware resources to computing domains. It makes the assumption that available resources are greater or equal than the required computational power.

No scheduler (and hence, no scheduling overhead) activity is required by the hypervisor, as computing domains are statically assigned to CPUs. Nevertheless, operating systems running as guests of the hypervisor may implement scheduling. Virtualisation techniques ensure safe cross-domain isolation. This approach is, for example, implemented by [23, 38].

Our approach is based on Jailhouse, a thin Linux-based partitioning hypervisor that targets many real-world systems. Motivated by the exokernel concept [11], our aim is to reduce the hypervisor to a minimum level of abstraction. Our goal is to minimise the hypervisor's interaction with guests, with the intention of preserving key quality parameters of any guest software regardless of if it is executed natively, or under the presence of a VMM.

A slim code base is a precondition for certifiability. The reduction of guest interaction ensures the maintenance of the platform's real-time capabilities by design—if no interceptions are present, no additional latencies can be introduced by the hypervisor. Running Linux in uncritical partitions of the system is a requirement for many real-world use cases. Therefore, we partition a booted Linux system, instead of booting Linux on a partitioned system (cf. Figure 2). This shifts any complex hardware initialisation to Linux, and ensures a small code base of the hypervisor as only a few platform specific drivers are required (during the operational phase, Linux is lifted into the state of a virtual machine).

Another advantage of the approach relates to running certified payloads: Many industrial codes are, for historical reasons, designed to run on single-core systems, and would require substantial porting efforts to leverage multi-core execution environments. Such changes would demand a re-certification of the codes; likewise, a time- and cost-consuming re-certification would be required if workloads are equipped with protection against, for instance, Spectre-type CPU weaknesses. Executing such legacy payloads in a partitioned cell has the advantage that the code does not require protection against said CPU weaknesses, because they are already implicitly required by the partitioning hypervisor. When no code changes are necessary, existing certifications can be retained, which is a clear

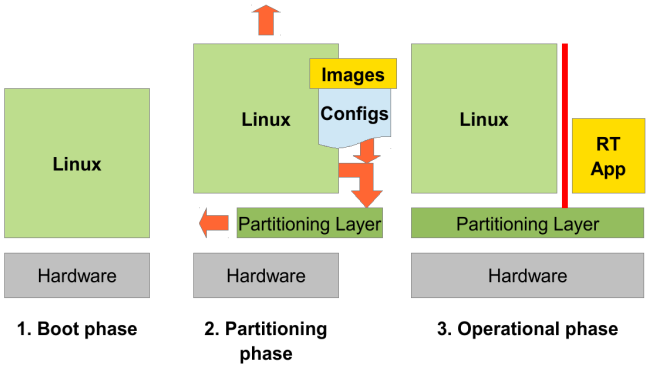

Figure 2. During the boot phase, Linux is booted on raw hard hardware. Jailhouse is inserted in the partitioining phase and lifts Linux to the state of a virtual machine. Configurations parameterise partitioning aspects of the system. In the operational phase, secondary operating systems run aside Linux on isolated hardware resources.

and substantial commercial advantage.

To create new isolated domains, specific hardware resources (e.g., CPUs, memory, peripheral devices) are offlined removed from Linux. The hypervisor is called to create a new domain has raw access to these resources. Secondary real-time operating systems, including Linux, or even bare-metal applications can be loaded to the domains. Jailhuse does not paravirtualise any resources as it exclusively assigns resources to computing domains.

The hypervisor shall ony be active during its boot phase (the installation and initialisation of the hypervisor) and during the partitioning phase (creation, initialisation and boot of new domains). During the operational phase (system is partitioned, and all partitions are running), there shall be no further hypervisor interaction required.

III. Requirements on Hardware Partitioning

In 1974, Popek and Goldberg postulated *Formal Requirements for Virtualizable Third Generation Architectures* in their seminal work [31]. They provide a fundamental formal definition of Virtual Machine Monitors (VMMs), and give requirements on their (efficient) implementation. Virtual machines have to satisfy three properties: equivalence, resource control and efficiency.

Equivalence implies that any program must behave the same, whether it is run on a virtual machine or on real hardware – exceptions to this principle are permitted for timing issues, and for the availability of physical resources. This is obviously problematic for the application domains we consider in this paper, in particular real-time critical workloads. *Resource control* implies that the virtual machine monitor is responsible for the allocation and moderation of hardware resources. *Efficiency* implies that *most* instructions should be natively executed without the need of hypervisor interception; notably, the definition of "most" is left unspecified.

As the efficiency criteria would exclude, for instance, emulated systems, Smith and Nair [37] confine VMM requirements to the equivalence and resource control criterion. In addition,

they call VMMs that fulfil the efficiency requirement *efficient VMMs*. Note that the efficiency criterion is satisfied if it only holds for *most* instructions, as by the definition above, which, in turn, necessarily implies that the criterion can only relate to average case efficiency. Citing Popek and Goldberg [31]: "Because of the occasional intervention of the control program, certain instruction sequences in K may take longer to execute, so assumptions about the length of time required for execution might lead to incorrect results." Consequently, the definition of efficient VMMs does not imply transitivity for timing- and latency guarantees given by the hardware, as required by real-time use cases.

A. Efficiency of VMMs

To provide a more quantitative version of the above efficiency requirements, consider a measurement M that is performed on a program P that, in turn, defines an observable property of the system (we will be using both terms interchangeably). M is, in our case, restricted to measure a temporal duration: The time value t_0 records that starting time of the measurement, and time t_1 records when the measurement is finished (the criterion for "finished" is given by the arrival of some external event, or by a satisfied internal logical condition). The value M_P of the measurement is then given by $M_P = \Delta t = t_1 - t_0$. An ideal system that is not subjected to any other loads than the measurement proper, repeated measurements deliver identical values for all runs: $M_P^{(i)} = m = \text{const.}$, where the superscript (i) indicates the i-th measurement. The criterion does, of course, not hold for systems that provide asynchronously triggered computational services (for instance, performing interrupt service routines, performing scheduling, ...) besides executing the subject program P. Such activities effectively influence the measurement in the form of noise, which we model by a stochastic parameter b, drawn from some probability distribution that must be provided depending in the actual circumstances. M_P^b represents a measurement subject to such noise.

Given a set of measurements $\mathcal{M} = \{M_P^{b,(i)}\}$ of the observable quantity P under noise b, we define that the observable is *transitive* for operation op if $\mathrm{op}(\mathcal{M}_{\mathrm{HW}}) = \mathrm{op}(\mathcal{M}_{\mathrm{VMM}})$ holds ($\mathcal{M}_{\mathrm{HW}}$ and $\mathcal{M}_{\mathrm{VMM}}$ denote that the measure is performed without and under the influence of the VMM). If transitivity for a given operation holds for all observables, we say that the observable itself is transitive.

For a throughput-optimized system, "avg" is arguably the operation of highest interest because the average-case performance is crucial. For real-time systems, "max" is relevant operation because worst-time behaviour is the essential characteristic of such systems.

Trivially, transitivity for avg and max is guaranteed by *ideal VMMs* that do not require any traps during the execution of guests. We tighten the definition of *efficient VMMs* and call a VMM an »*ideal VMM*«, if no traps are required during the operational phase:

> A VMM is ideal, if all instructions are natively executed during the operational phase. Only maintenance operations may be intervented by the hypervisor. Instructions that cause hypervisor intervention are considered to be violations.

This means that cost of ideal VMMs only is limited to the hardware costs of virtualisation [10].

While the definition of an ideal VMM is hard to satisfy by hypervisors that premise on hardware resource sharing and rely on software intensive hardware overcommitting (implemented by, for instance, device emulation, paravirtualisation [1] or domain scheduling), it is a realistic goal for partitioned setups. For partitioned systems, we further define:

> A partitioned system is ideal, if exclusive resource access is granted by an ideal VMM.

Consequently, the *ideal VMM* criterion can only apply to a *subset* of partitions of a partitioned system, for reasons that we discuss in the next section. We call a partition that runs as a guest of an ideal VMM an *ideal partition*.

While ideal partitions can already be achieved with modern virtualisation extensions for constrained environments, complex real-world scenarios still require occasional intervention.

B. Architectural System Limitations

Modern hypervisors usually try satisfy the efficiency criterion by using various hardware based virtualisation extensions provided by modern architectures (e.g., VT-x [39], VT-d [17], SVM [35], VT [41], ...) that allow for executing *most* instructions natively. MMU enhancements [39, 35]) of those extensions (e.g., page-table virtualisation) assign host physical memory to guests. Address translation of guest adresses to host physical adresses is transparently performed by the MMU and does not require any hypervisor interception—it will only trap in case of access violation. Furthermore, those extensions introduce an OS-superior privilege level in which all hardware resources are accessible. The hypervisor may, for instance, moderate access to shared resources, or directly assign resources to guests.

Other hardware based extensions target the reduction of interrupt overhead [17, 35, 16, 26]. *Interrupt remapping* allows to directly route selected interrupts to virtual machines without the need of hypervisor interception. Without interrupt remapping support, interrupts trap the hypervisor, which will dispatch the interrupt, and, if necessary, reinject it to guests. If a device is directly assigned to a guest, or if a platform specific interrupt (e.g., a platform timer interrupt) arrives at the CPU interface, interrupt remapping will directly send the interrupt to the virtual machine, if running. The aim of those extensions is further reduction of VMM overhead.

Nevertheless, depending on their semantic, a hypervisor may, for instance, be required to moderate the access to sensitive system registers, such as model specific registers (MSRs) or different control registers (CRs) on x86, or coprocessor registers (CPs) on the ARM architecture.

The motivation of any of hardware based virtualisation extension is to reduce the activity of the hypervisor by trap reduction in order to increase the performance of the system – frequently required policy decisions are offloaded to

hardware. Nevertheless, the development of those extensions is often driven by throughput-oriented general purpose systems (optimised on the average case): it is sufficient to offload *most* decisions, while for real-time systems it is essential that *all* decision can be offloaded to hardware.

During the development of a hypervisor that aims towards zero traps, we elaborated concrete system requirements for ideal partitioned systems. In the next section, we present device specific and platform specific requirements for real-world systems. For any requirement, we present examples that violate the requirement, as well as potential software-based workarounds. Such workarounds are, of course, contrary to the envisioned concept, but required due to hardware limitations.

C. Device Specific Requirements

Peripheral devices (e.g., SPI, I^2C, UART or ethernet controllers) are essential components of any real-world setup, but they are often ignored and underestimated during systems development under laboratory conditions. Peripheral devices are partitionable entities, if they can be spatially and logically isolated.

Requirement 1: Logical Device Partitioning

The platform must provide means to transparently assign device control to guests

In their simplest form, a device consists of control structures and a signalling interface. The platform must provide means to assign those interfaces to guests without hypervisor intervention.

On many architectures, device control structures are accessed through Memory-Mapped I/O (MMIO). The MMIO address space of a device is backed by the device's registers. The typical page size of almost all modern architectures is 4 KiB or more, and represents the finest granularity of memory that can be assigned by the MMU. Hence, devices need to be spatially isolated by the granularity of the page size.

32 bits can be seen as the de-facto lowest limit of physical address space of modern CPUs. While this provides enough space to place different devices on separate pages, hardware manufacturers often place multiple devices on one single page, even different types of devices.

This becomes problematic for hardware partitioning, when those devices need to be assigned to different domains, since only pages can be assigned to guests without the need to trap and dispatch memory access.

A software based workaround to overcome this issue is *sub-paging*, a technique where the hypervisor allows for mapping memory areas to guests that are smaller than the page size. The hypervisor traps on any access and only forwards the request if the guest has access permission. Any other access is a violation. This leads to noticeable and undesired slow-downs.

Spatial isolation can be solved by hardware manufactures by assigning different devices to separate pages.

Furthermore, devices need a signalling interface, typically implemented by interrupts. The platform must provide means that interrupts may directly arrive at the guests without hypervisor intervention. This technique is called interrupt remapping and is already supported by the virtualisation extensions of an increasing amount of architectures [17, 16, 35, 26].

Requirement 2: Hierarchical Autonomy of Devices

Any device metafunctionality must be isolated from other devices and must be logically partitionable

Metafunctions of devices (e.g., device power and reset control, speed, baudrate) are often controlled by secondary devices, such as clock or reset controllers. Such instances must be partitionable on a device scope level. Any modification within a device scope must not affect other devices.

On many ARM-based platforms, for example, the above-mentioned clock and reset controllers are not partitionable without hypervisor interception. They are a) located on single memory pages, and b) control all peripheral devices of the system. Complication this situation, they c) may also include clock and reset lines, which are often implemented as complex dependent hierarchical structures.

Currently, guest access to those functions is not possible without complex hypervisor intervention. While guests should be allowed to change device settings during runtime, one workaround (without traps) is to statically set up the device settings during the boot phase of the hypervisor and to forbid any further modification. This can be inconvenient for some scenarios. A complex alternative is to paravirtualise those devices.

Nevertheless, the these issues must be addressed by hardware manufacturers by designing device control instances in a partitionable way: One possible implementation would be to place all metafunctions of a device to a single page and to reduce inter-device dependencies of hierarchically structured clocks. This provides configurational flexibility, as the page can simply be hidden if a guest shall not be permitted to acess these functions. Another approach is to use system specific registers in a standardised manner[1] for device reconfiguration. Whitemaps can be used to grant fine-grained permission to functionalities.

However, during the implementation of our hypervisor, we observed platforms[2] where access to disabled devices stops the whole platform. Hardware manufactures must ensure, that erroneous device access from within a domain must not affect the whole platform.

Logical isolation of a device is not limited to clock and reset controllers. Any instance that interacts with a device (e.g., DMA controllers) must be designed in a partitionable way. This means, usage, access or configuration of the instance must not interfere with any other device or CPU.

[1] yet to be defined

[2] disclosure would identify authors

D. Platform Specific Requirements

Requirement 3: Platform Resource Partitioning

System platform resources must be partitionable with respect to their domain affinity

A CPU interface must not be able to change the state of a CPU of another computing domain. This includes, e.g., power management such as sleep states or frequency scaling, memory management or interrupt delivery. The platform must provide CPU local control structures, or structures that are restrict to the local computing domain.

Many traps on a platform result from the lack of (full) virtualisability of platform specific resources. Access to sensitive system registers, reconfiguration of CPU power management settings or interactions with interrupt controllers are typical causes for frequent traps that require hypervisor assisted moderation. The hypervisor must ensure that any access must not cause any unintended side effects to other domains. Simple policy-based decisions can be resolved by hardware support.

On x86 platforms, for example, a hypervisor can conditionally trap model specific register (MSR) access, based on permission bitmaps. It allows either unmoderated access to insensitive registers, to trap on reads, writes, or to trap on both. depending on the semantics of the register. Platform resource partitioning requires that any interarction with machine specific registers must not leak information of other domains, or affect them.

The x2APIC implements interrupt controller virtualisation support for Intel x86 platforms. It uses MSR-based register access instead of conventional MMIO-based access. While in a partitioned setup, a hypervisor may allow unmoderated access to insensitive registers, access to sensitive registers, such as the interrupt command register (ICR) must be intercepted. The ICR is used to send inter-processor interrupts (IPIs) to other CPU interfaces. Hence, raw write access must be forbidden, as CPU interfaces of other domains can be adressed. Access must be intercepted by the hypervisor, which will check permissions and forward the request. Other architectures like ARM [27, 26] have similar interfaces that require moderation by the hypervisor.

Interception of platform devices can generally be avoided, if CPU local interfaces could be parameterised by the hypervisor with the scope of the domain. Similar fine-grained conditional register trap that is, for exapmle, based on bitmasks, is possible and already supported for various other CPU control registers (CR) on x86 [39].

Requirement 4: Cross Core Independence

The microarchitectural state of a core must not be affected by neighboured cores

Many publications and successful attacks on microarchitectural and speculative attacks underline the risks of shared hardware resources [22, 40, 42, 33, 28, 25]. For real-time performance reasons, and for security reasons [40, 42, 33], parts of the execution unit must not be shared. Symmetric Multithreading, for example, violates cross core independence.

Besides SMT, many microarchitectures implement further carriers of potential coverage channels: caches. Last level cache (LLC) is often shared across different physical CPUs. Depending on the architecture's cache organisation, this can result in sharing of the LLC across different domains. Sharing caches can lead to performance and security issues and should be considered dangerous due to following reasons.

1. On many Intel CPUs, the LLC is an inclusive cache. This means, the LLC includes all data from lower cache levels. Consequently, the eviction of an entry in the LLC causes the eviction of the entry in all lower levels. Aimed memory traffic generated by a CPU can cause consequent overwrites of the whole shared LLC. As the LLC is inclusive, it will invalidate everything in the L1 cache of all other CPUs [15]. With this, a CPU can cause cache misses of another CPU that is assigned to a different domain. This causes unintended and unacceptable slow downs.

2. Furthermore, (shared) caches are a common target for many microarchitectural attacks [33, 43, 42, 14]. Yarom et al. have shown that their FLUSH+RELOAD side channel attack can be used to reconstruct the control flow of programs, if two independent processes share the same pages (e.g., shared libraries). In their paper [43], they conclude to cryptographic secrets by the analysation of the control flow. The FLUSH+RELOAD pattern is the foundation of many further microarchitectural attacks [42, 22, 33]. Shared caches increase the attack surface.

In partitioned setups, there is no sharing of common physical pages across cores. Therefore, partitioned systems do not benefit from shared caches. This protects them against attacks mentioned in 2), but still exposes them to threats mention in 1).

To overcome the scenario explained in 1.), Intel implements the Cache Allocation Technology (CAT) [15] as part of their Resource Director Technology (RDT) [7]. CAT allows to partition the LLC by the exclusive assignment of dedicated cache portions to cores. Nevertheles, we believe that their implementation should be considered inconsistent: While a core may only allocate and evict cache lines only within its scope, "a read or write from a core may still result in a cache hit if the cache line exists anywhere in the LLC." [15] This, in turn, opens a new potential[3] attack vector for side channel attacks: an attacker can FLUSH+RELOAD a cache line. The data is in user by neighboured cores, if the access time measurement confirms L3 presence right after the flush.

There are too many indicators that shared caches misbehave in certain situations, yet there are no benefits in partitioned scenarios. Platform should either not support shared caches, or implement cache partitioning in an consequent nonreactive manner.

[3] unaudited

Shared system resources and traces in the microarchitectural state of a CPU endanger many modern computing systems. It requires careful analysation if and to what degree partitioned systems might be affected.

IV. Cross-domain protection against speculative execution exploits

Performance, throughput and efficiency of almost all modern CPUs rely on aggressive microarchitectural optimisations. Pipelining, speculative execution and out-of-order execution are prominent and effective optimisation techniques.

Out-of-order execution allows single CPUs to efficently reorder instructions in order to achieve an optimal utilisation of the CPU pipeline. CPU pipelines allow parallel execution of different stages of multiple independant instructions. Branch prediction is a speculative execution technique, to achieve optimal utilisation of the CPU pipeline. A CPU that implements branch prediction speculatively executes instructions in advance of conditional branches with yet unknown results. It may execute instructions that may not be needed or that are not allowed. High utilisation of all execution units in parallel is one of the elementary reasons of the high performance of modern CPUs.

Naturally, speculative execution inherently leads to erroneous decisions. Thus, executed mispredictions are transparent to users as they are rolled back in order to preserve an accurate external state. However, they leave microarhitectural traces in the internal state of the CPU that open potential covert channels. Misdirection in combination with internal state analysis allow an attacker conclude the external state of the CPU. In 1995, Sibert et al. indicate the existence of such microarchitectural state dependant covert channels [36].

Two decades later, in the beginning of 2018, independant researchers present a whole new class of microarchitectural attacks: the family of Spectre attacks. Since then, many researchers found new methods or variations of attacks on speculative execution of CPUs.

All Spectre attacks and their variations violate fundamental guarantees on the confidentiality of data that is given by (core-local) protection mechanisms of a CPU. Software based solutions in operating systems and system firmware, as well as processor microcode updates are required to mitigate attacks. Many of those numerous mitigations are cost-intensive.

A. Overview: Attacks and their Mitigation

a) Spectre: One pattern of Spectre attacks is to mislead execution units to perform dependant loads. Transient execution attacks [22] try to speculatively load memory where the address *depends* on the offset of a secret (dependent loads). This intentional misguidance leads to mistaken speculative execution and the external state is rolled back. While this preserves external consistency, attackers can draw conclusions on the secret by analyzing the internal state that was modified by the execution of transient instructions. Many attacks analyse the state of caches to leak informations on the internal state: evaluation of memory access time (e.g., FLUSH+RELOAD attacks [43]) to adjacent memory cells can be used test if data is present in caches. A valid cache line can be loaded through a transient execution. The number of the warm cache line carries the original secret.

Those attacks are mitigated by CPU microcode and system firmware updates that introduce speculation barriers, by compiler-assisted convertion of indirect branches to return statements, and by OS-based protection against speculation on user-controlled data in kernel space and others.

b) Meltdown: A similar attack is Meltdown [25] (aka. Spectre v3 or Rogue Data Cache Load). It exploits out-of-order execution to bypass illegal memory access to areas protected by memory management units (MMUs) on many Intel and ARM processors. While access to protected memory will cause an exception, out-of-order execution bypasses MMU-based protection mechanisms. Again, the secret can indirectly be used to warm up a cache line that remains as an artefact of the internal microarchitecutral state. Meltdown is able to leak data from present, but protected privileged pages (e.g., data from kernel space).

For performance reason, many operating systems share the same page table for user and kernel space. Kernel space pages are marked as privileged and not accessible from user space. This saves cost-intensive page table switches on privilege level switches. Meltdown overcomes this security barrier. Hence, it is mitigated by the isolation of user and kernel pages: Page Table Isolation (PTI). When software runs in user space, only a small priviliged trampoline pages is mapped that hands over to kernel space pages. This requires page table switches on every privilege level switch.

c) Foreshadow / L1TF: Foreshadow [40] and Foreshadow-NG [42] are attacks on SGX (secure enclaves) and MMUs of modern Intel CPUs. Foreshadow allows to read secret data from SGX enclaves, and Foreshadow-NG (also known as L1TF or Level 1 Terminal Fault) allows to read any data from the core-local level 1 cache. Foreshadow-NG exploits additional design flaws of MMUs: Intel MMUs speculatively use physical addresses of invalid page table entries (i.e., entries with cleared 'present'-bit). Intel is hypothesised to "implement L1 tag comparison in parallel with the adress translation process for performance reasons" [42]. While access to invalid page table entries raises an exception (i.e., Terminal Fault), the data of the L1 cache is already used for transient out-of-order execution of the following instructions. Analogously to other microarchitecural attacks, change of the internal microarchitectural state is used to leak secrets. L1TF is able to leak any data that is present in the core-local L1 Cache.

User-space processes may speculate on previously available pages that are not present (e.g., swapped pages). Operating systems running as virtual machines are able to a) leak data from the hypervisor and b) leak data from other virtual machines that are scheduled on the same core and leave data traces in the L1 cache. Secrets can also leak through neighboured simultaneous multithreading (SMT) siblings as they share the L1 cache.

To mitigate L1TF, operating systems implement page table entry (PTE) inversion and conditional cache flushes. PTE inversion applies a bitmask to the physical address of unpresent pages in order to point to invalid physical addresses. This protects operating systems from users that speculate on unpresent pages. To protect hypervisors against malicious virtual machines, and to protect virtual machines against each other, operating systems implement (expensive) conditional L1 cache flushes on privilege level switches. A full prevention of cross-VM exploits requires to disable SMT.

d) Microarchitectural Data Sampling: RIDL (Rogue In-Flight Data Load) and Fallout present microarchitectural data sampling (MDS) attacks that target CPU-internal buffers (e.g., Store Buffer, Fill-Buffer or Load-Port) of Intel CPUs. During its execution, a victim process utilises CPU internal buffers with private data. Later, the scheduler of the operating system replaces the victim process with the attacking process. "When the attacker also performs a load, the processors speculatively uses in-flight data from the Line Fill Buffers (LFBs) (with no addressing restrictions) rather than valid data" [33]. Covert channels, e.g. the FLUSH+RELOAD attack, finally reveal the secret of the victim process.

Recent Intel CPU microcode versions patch instructions to perform flushes of various exploitable internal CPU buffers. For virtualised environments, an alternative, yet more cost-intensive mitigation are L1D cache flushes. Nevertheless, this is the prefered mitigation for systems that are vulnerable to L1TF, as they need to conditionally flush L1 caches on the same paths in either case. "The mitigation is invoked on kernel/userspace, hypervisor/guest and C-state (idle) transitions." [6]

While store buffers are partitioned across SMT threads, entering or leaving sleep states repartitions the buffers and data can be exposed between SMT threads. Depending on the workload, full mitigation requires SMT to be disabled as fill buffers are shared between SMT threads. [21]

B. Jailhouse and Speculative Execution Attacks

All known speculative execution attacks exploit CPU-local interfaces. At the moment of writing, there are no known speculative execution attacks across physical CPU boundaries. To attack the victim, it needs to temporarily share the same core with the attacker. Naturally, CPUs can not leak data they do not know or data they can not see.

To isolate domains of different criticality, Jailhouse exclusively and statically assigns CPUs to its guests, i.e., to different execution domains. By design, Jailhouse does not schedule domains. It has no means built in to share a logical CPU between multiple guests. This differentiates Jailhouse significantly from the hypervisors that run, e.g., in cloud environments.

This fundamental architectural decision provides a strong cross-domain protection layer against speculative execution attacks. Nevertheless, the following scenarios have to be carefully assessed: a) Inter-Guest attacks, and b) Attacks on the hypervisor.

a) Inter-Guest Attacks: While Jailhouse does not schedule guests, a guest (e.g., an operating system) may schedule different processes. Hence, malicious code can be used to leak secret information of other processes of the same execution domain.

Nevertheless, if a domain needs protect itself against attacks from within the domain, the operating system may implement countermeasures. On the ARM64 architecture, speculation barriers are implemented by the secure monitor running in exception level 3 (the hypervisor runs in the lesser privileged exception level 2). Calls from exception level 1 (OS / kernel) of a guest to exception level 3 require interception and moderation by the hypervisor. On affected ARM64 systems that implement secure monitor-based speculation barriers, Jailhouse calls anti speculation barriers on every trap.

It is the decision of the guest whether further mitigations are required.

b) Hypervisor Attacks: One design goal of Jailhouse is to setup hardware partitioning, which, ideally, requires no further hypervisor interception for regular operation. Conceptually, the hypervisor should only be active during its boot and partitioning phase, and only handle unrecoverable critical exceptions during its operational phase. The architecture provides strong protection of the hypervisor: speculative execution attacks can only work in cases, where the victim executes code and operates on secrets.

The zero-trap goal is already achieved for some use cases on Intel x86 systems, it is generally limited by current hardware support. These limitations were presented in Section III. Generally, all speculative execution attacks require victim code to run.

Nevertheless, in order to implement Jailhouse on common architectures the hypervisor needs to intercept or moderate certain situations, that depend on the target architecture's virtualisation capabilities. Furthermore, Jailhouse implements a slim hypercall interface for management tasks. This involves hypervisor activity that can potentially be used in speculative execution attacks.

A CPU can only leak what it can see. In case of Jailhouse, this is includes its binary code, configuration, and sensitive guest state information. System configuration contains partition information and information on the platform's topology. This does not contain secret data that needs protection. As the hypervisor is developed as an Open Source project, hypervisor binary code does not need protection.

Malicious guests may use or even synchronously control hypervisor activity to prepare for speculative execution attacks. By design, Jailhouse only exposes a minimum attack surface to guests as it only maps a small subset of guest pages into its address space that is required to perform its duties. Jailhouse maintains isolated core-local address spaces and does and does not share CPU private pages across CPUs. Core-local CPU state is not visible to other CPUs. Only a small set of uncritical management information (e.g., the hypervisor state) is shared across all CPUs. Because of its simplicity, address space isolation was implemented with reasonable effort. Currently,

Linux's KVM ungergoes efforts of implementing a similar isolation strategy[4].

c) Attacks on SMT: Simultaneous multithreading is a further technique to optimally utilise available hardware resources. SMT transparently exposes multiple logical CPU interfaces to users, while parts of the underlying physical units are still shared (e.g., L1 caches). Execution units can be shared or duplicated between logical threads.

Sharing of execution units may lead to mutual contention between different threads. While SMT increases overall performance and throughput of a system, contention causes unintended latencies, which have negative impact on the real-time behaviour of systems. Hence, we share the opinion of [24] to disable SMT in any case, in order to maintain real-time capabilities.

However, if SMT remains active, logical threads are vulnerable to attacks that exploit shared execution units or shared chaches. Hence, we recommend to allocate threads of physical CPUs to the same execution domain. It is the decision of the guest if further OS-based mitigations are required.

In any case, in the Jailhouse architecture, all secret information remain in guests on isolated CPUs. Under ideal conditions, no secrets remain in the hypervisor.

V. Discussion & Related Work

The requirements of Popek and Goldberg were postulated in 1974, but are—almost half a century later— still applicable to modern systems. However, some adaptions and extensions are required to handle contemporary real-world use cases that need to satisfy real-time and mixed criticality requirements. with slight adaptions. We have presented some general, high-level criteria, and have also derived consequences for hypervisor assisted static hardware partitioning.

Many of the existing hardware virtualisation extensions reduce hypervisor interaction to optimise the average-case behaviour (throughput) of systems. As real-time systems are optimised for the worst-case, these extensions do not always necessarily meet real-time requirements in terms of low latency as avoidable hypervisor interception is required in many cases.

For statical hardware partitioning, any remaining interception causes are mainly solved by software by policy-based decisions that can fully be offloaded to hardware. Hardware/Software codesign can close the missing gap: software requirements on the system need to be carefully evaluated with systems designers.

The disclosure of Spectre and related attacks target the complexity of modern systems, and require, depending on the workload, expensive mitigation measures. Many of these attacks exploit sharing of resources: hardware units can be shared across multiple tasks. This includes, for example, physical CPUs, if different workloads are scheduled on the same execution unit, and caches, if different workloads on different execution units share the same caches.

[4]cf. https://lkml.kernel.org/lkml/20190514070941.GE2589@hirez.programming.kicks-ass.net/T/

With the existence of multi-core SMP CPUs, software-based sharing techniques, such as domain scheduling, can be avoided for many use cases. From a real-time perspecitve, this lowers system overhead and on the other hand, it avoids sharing of hardware resources and reduces the attack surface for attacks based on speculative execution.

Barrelfish [34] is an operating systems that is designed for heterogeneous multi- and many core systems. They focus on operating system scaleability aspects, as, for example, the number of CPUs on a system grows more than indivudual clock rates. They argue that multi-core systems can be seen as a network of independent cores, and that no sharing at lowest level is required [2].

Other researchers analyse operating system overhead and try to offload, for example, scheduling decisions to hardware [9]. Micokernel approaches follow similar goals: a significant fraction of decision should be offloaded to hardware.

VI. Conclusion

We presented the concepts of Jailhouse, a real-world Linux-based static partitioning hypervisor. For many real-world use cases that require Linux to run side by side with real time operating systems, we discussed that hardware partitioning is a viable alternative to classical OS-based isolation approaches, especially when legacy workloads must be protected against hardware weaknesses like Spectre-class speculation attacks without modifying certified components.

We defined the concepts of *ideal VMMs*, *ideal partitions* and *ideal partitioned systems* with the goal of establishing zero-trap hypervisors on real-world systems that only need to account for setting up partitions, but do not interact with the content of any partitions in the operational phase. Experiments with an implementation of the concept on multiple hardware platforms showed limitations inherent in current hardware. We discussed necessary improvements in future virtualisation techniques to facilitate a realisation of our approach on realistic systems.

References

[1] Paul Barham, Boris Dragovic, Keir Fraser, et al. "Xen and the Art of Virtualization". In: *Proc. of the 19th ACM Symposium on Operating Systems Principles*. 2003.

[2] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. "The multikernel: a new OS architecture for scalable multicore systems". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM. 2009, pp. 29–44.

[3] Lukas Bulwahn. "Is Linux Kernel Development Good Enough to Make Your Life Dependon it? Progress on Procedures & Methods to Qualify the Linux Kernel Development Process". In: *Embedded Linux Conference Europe (ELCE17)*. Oct. 2017.

[4] Alan Burns and Robert Davis. “Mixed criticality systems-a review”. In: *Department of Computer Science, University of York, Tech. Rep* (2013), pp. 1–69.

[5] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. “A systematic evaluation of transient execution attacks and defenses”. In: *arXiv preprint arXiv:1811.05441* (2018).

[6] The kernel development community. *Linux Kernel Documentation on MDS - Microarchitectural Data Sampling*. see Documentation/admin-guide/hw-vuln/mds.rst.

[7] Intel Corp. *Intel Resource Director Technology (Intel RDT)*. 2019. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html.

[8] Alfons Crespo, Ismael Ripoll, and Miguel Masmano. “Partitioned Embedded Architecture based on Hypervisor: The XtratuM approach”. In: *Proceedings of the 8th European Dependable Computing Conference (EDCC)*. IEEE. 2010.

[9] Christian Dietrich and Daniel Lohmann. “OSEK-V: Application-Specific RTOS Instantiation in Hardware”. In: *Proceedings of the 2017 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES ’17)*. (Barcelona, Spain). New York, NY, USA: ACM Press, June 2017.

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

[11] Dawson R Engler, M Frans Kaashoek, et al. *Exokernel: An operating system architecture for application-level resource management*. Vol. 29. 5. ACM, 1995.

[12] Linux Foundation. *Civil Infrastructure Platform (CIP)*. 2019. URL: https://www.cip-project.org.

[13] Thomas Gleixne, John Kacur, and Clark Williams. “The Cyclictest Real-Time Benchmark”. In: ().

[14] David Gullasch, Endre Bangerter, and Stephan Krenn. “Cache games–Bringing access-based cache attacks on AES to practice”. In: *2011 IEEE Symposium on Security and Privacy*. IEEE. 2011, pp. 490–505.

[15] *Improving Real-Time Performance by Utilizing Cache Allocation Technology Enhancing Performance via Allocation of the Processors Cache*. Intel Corporation. Apr. 2015.

[16] *Intelő 64 Architecture x2APICSpecification*. Intel Corporation. Mar. 2010.

[17] *Intelő Virtualization Technology for Directed I/O*. Rev. 3.0. Intel Corporation. June 2018.

[18] Robert Kaiser and Stephan Wagner. “Evolution of the PikeOS microkernel”. In: *First International Workshop on Microkernels for Embedded Systems*. 2007, p. 50.

[19] Beckhoff Automation GmbH & Co. KG. 2019. URL: https://www.beckhoff.com/.

[20] Jan Kiszka. “A Linux/Xenomai Platform for High-Performance Magnetic Resonance Scanners”. In: *Xenomai User Meeting 2009 (XUM2009)*. 2009.

[21] Andi Kleen. *Linux Kernel MDS mitigation patches*. Available at https : / / git . kernel . org / pub / scm / linux / kernel / git / torvalds / linux . git / commit / ?id = 95310e348a321b45fb746c176961d4da72344282.

[22] Paul Kocher, Daniel Genkin, Daniel Gruss, et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *40th IEEE Symposium on Security and Privacy (S&P’19)*. 2019.

[23] Ye Li, Matthew Danish, and Richard West. “Quest-V: A virtualized multikernel for high-confidence systems”. In: (2011).

[24] “Linux Kernel PreemptRT real-time extension”. In: ().

[25] Moritz Lipp, Michael Schwarz, Daniel Gruss, et al. “Meltdown: Reading Kernel Memory from User Space”. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[26] ARM Ltd. *ARM Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0*. 2016.

[27] ARM Ltd. *ARM Generic Interrupt Controller, Architecture version 2.0*. 2013.

[28] Marina Minkin, Daniel Moghimi, Moritz Lipp, et al. “Fallout: Reading Kernel Writes From User Space”. In: (2019).

[29] Rainer Müller, Daniel Danner, Wolfgang Schröder-Preikschat, and Daniel Lohmann. “MultiSloth: An Efficient Multi-Core RTOS using Hardware-Based Scheduling”. In: *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS ’14)*. (Madrid, Spain). Washington, DC, USA: IEEE Computer Society Press, 2014, pp. 289–198.

[30] Dag Arne Osvik, Adi Shamir, and Eran Tromer. “Cache attacks and countermeasures: the case of AES”. In: *Cryptographers track at the RSA conference*. Springer. 2006, pp. 1–20.

[31] Gerald J Popek and Robert P Goldberg. “Formal requirements for virtualizable third generation architectures”. In: *Communications of the ACM* 17.7 (1974), pp. 412–421.

[32] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. “ROS: an open-source Robot Operating System”. In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[33] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. “RIDL: Rogue In-flight Data Load”. In: *S&P*. May 2019.

[34] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. “Embracing diversity in the Barrelfish manycore operating system”. In: *Proceedings of the Workshop on Managed Many-Core Systems*. Vol. 27. 2008.

[35] *Secure Virtual Machine Architecture Reference Manual*. Rev. 3.01. Advanced Micro Devices. May 2005.

[36] Olin Sibert, Phillip A Porras, and Robert Lindell. “The Intel 80x86 processor architecture: pitfalls for secure systems”. In: *Security and Privacy, 1995. Proceedings., 1995 IEEE Symposium on*. IEEE. 1995, pp. 211–222.

[37] Jim Smith and Ravi Nair. *Virtual machines: versatile platforms for systems and processes*. Elsevier, 2005.

[38] Udo Steinberg and Bernhard Kauer. “NOVA: a microhypervisor-based secure virtualization architecture”. In: *Proceedings of the 5th European conference on Computer systems*. ACM. 2010, pp. 209–222.

[39] Rich Uhlig, Gil Neiger, Dion Rodgers, et al. “Intel virtualization technology”. In: *Computer* 38.5 (2005), pp. 48–56.

[40] Jo Van Bulck, Marina Minkin, Ofir Weisse, et al. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution”. In: *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, Aug. 2018.

[41] Prashant Varanasi and Gernot Heiser. “Hardware-supported virtualization on ARM”. In: *Proceedings of the 2nd Asia-Pacific Workshop on Systems (APSys)*. 2011.

[42] Ofir Weisse, Jo Van Bulck, Marina Minkin, et al. “Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution”. In: *Technical report* (2018).

[43] Yuval Yarom and Katrina Falkner. “FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: *23rd USENIX Security Symposium*. 2014, pp. 719–732.

[44] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. “PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms”. In: *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2014, pp. 155–166.