Universiteit Leiden

Master Computer Science

Hypervisor Isolation

Name:	Mathé Hertogh
Student ID:	1521748
Date:	15/07/2022
Specialisation:	Advanced Computing and Systems
1st supervisor:	Kristian Rietveld Leiden University
2nd supervisor:	Cristiano Giuffrida Vrije Universiteit Amsterdam
Daily supervisor:	Manuel Wiesinger Vrije Universiteit Amsterdam

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Contents

Acknowledgements

I would like to thank my supervisors, Cristiano Giuffrida, Kristian Rietveld and Manuel Wiesinger, for their help throughout the project. The initial project idea was from Cristiano, who also later on gave interesting ideas for improvement, as well as practical advice. Kristian gave valuable suggestions and was a very nice conversation partner to discuss my project with. Manuel was always ready to help me, whatever the problem was, big or small. He was eager to share his kernel knowledge with me, which was very helpful. Thank you all very much.

Abstract

Virtual machines are supposed to enforce a rigid security boundary between guest and host. Transient execution attacks have however been shown to enable virtual machines to compromise their hypervisor. This thesis explores a new defense, *hypervisor isolation*, for hypervisors against microarchitectural attacks from malicious guests. Hypervisor isolation guarantees that a hypervisor runs on different physical cores than the virtual machines it manages. This eliminates the sharing of any core-local micro-architectural resources and hence all attacks that rely on them.

1 Introduction

The disclosure of Spectre [38] and Meltdown [50] in 2018 kicked off the era of transient execution attacks. These attacks abuse the design of modern processors to leak secret data across arbitrary security domains. Impactful examples are user applications reading kernel data and virtual machines (VMs) reading hypervisor data. Transient execution attacks form a severe security thread, as they can compromise even the most privileged security domains.

Mitigating transient execution attacks is complicated, because the root cause lies in the hardware design of modern processors: software from different security domains share microarchitectural resources. Fundamentally solving this problem in silicon would require a complete redesign of modern processors, discarding decades of processor research and engineering efforts, probably resulting in severe performance degradation. Furthermore, vulnerable hardware is already deployed in billions of devices around the world and these devices must also be protected.

These concerns have led to mitigations which do not tackle the fundamental problem of sharing microarchitectural state with untrusted parties. Instead, minimally invasive mitigations were deployed throughout the whole computer ecosystem: in-silicon fixes [6], microcode updates [32], kernel patches [35] and application updates [64]. They can be described as *spot mitigations*, defending against *specific* attacks. This renders them ineffective against (slightly) different attacks, such as yet unknown ones. Many new transient execution attacks have been discovered in the past four years [10, 40, 36, 74, 28, 56, 31, 70, 67, 65, 66, 12, 62, 68, 21, 11, 72, 69], which time after time showed that the existing spot mitigations did not provide comprehensive protection. The response has been incrementally applying more spot mitigations.

The current state-of-the-art defense against transient execution attacks is a plethora of spot mitigations. This has an inherent complexity which is unfavorable for maintenance. Moreover, all of these spot mitigations induce a major combined performance overhead [47, 45, 46]. Persistent effort will have to be put into developing new spot mitigations for the new attacks yet to be discovered. And such additional mitigations will hinder maintenance and performance even more. Lastly, and most importantly, the current spot mitigations are not guaranteed to defend against attacks yet unknown to the scientific or industrial community—but perhaps are known to people with bad intentions.

Crucial security boundaries that have been compromised by transient execution attacks are the hypervisor versus VM boundary and the VM versus VM boundary. Virtualization is nowadays one of the most, if not the most, used technology for safely running untrusted code. Cloud environments use virtualization as their main security enforcement and unikernels even make deploying user applications inside VMs an efficient isolation primitive [42]. The security boundary isolating a VM from the rest of the system should be a rigid one, but multiple transient execution attacks have penetrated this boundary in the past [70, 67, 62, 68, 69]. The aforementioned shortcomings of the state-of-the-art defense consisting of spot mitigations threaten the maintainability and performance of hypervisors, as well as in all likelihood leaving VMs and hypervisors vulnerable to yet unknown attacks.

This thesis explores a future proof defense for hypervisors against core-local transient execution attacks: *hypervisor isolation*. Hypervisor isolation is the opposite of a spot mitigation: it protects hypervisors and VMs against a whole class of attacks, even yet unknown attacks. It does so by eliminating the key source of any core-local transient execution attack: sharing core-local microarchitectural state with untrusted parties. Contrary to contemporary hypervisor designs, hypervisor isolation enforces that VMs run on separate physical cores from their hypervisor, as well as from each other. This mitigates all, even yet undiscovered, core-local transient execution attacks between hypervisors and VMs, since they do not share any core-local microarchitectural state anymore.

We implemented a prototype of hypervisor isolation for KVM [37], the hypervisor built into the Linux kernel. Our prototype shows the practicality of hypervisor isolation on real world systems. We evaluate its effectiveness and performance.

2 Background

This section discusses background material that is relevant for understanding the remainder of this thesis.

2.1 Pipelining and Out-of-order Execution

All major modern instruction set architectures (ISAs) are based on the Von Neumann execution model. A program consists of an ordered sequence of instructions, which are one by one executed by the processor in the specified order. Microarchitectures implementing such ISAs are however not bound by these limitations: they may do anything they want, as long as they satisfy the ISA. This freedom enables powerful performance optimizations that utilize instruction level parallelism at the microarchitectural level, such as pipelining and out-of-order execution and speculative execution. These techniques abandon sequential instruction processing and instead approximate a data-flow execution model at the microarchitectural level, executing instructions as soon as their source operands become available. All performant modern processors today employ these optimizations, thereby executing multiple instructions at the same time as well as out of program order. Re-order buffers at the end of the pipeline are used to retire instructions in program order, thereby committing their architectural changes in order. Hence from the point of view of the programmer, it looks like the instructions are executed one by one and in program order, as specified by the ISA.

Control flow instructions pose a challenge for pipelined processors. Instruction are fetched at the first pipeline stage, and decoded at a later stage. Therefore, before the next instruction gets decoded, the processor must already decide which instruction to fetch next. But the previously fetched and yet undecoded instruction might divert control flow and hence the processor does not know the correct next instruction. Even after decoding of the previous instruction, the next instruction may still be unknown: conditional branches are resolved at a much later stage in the pipeline. These problems are amplified in out-of-order processors, which try to execute future instructions ahead of time: what future instruction path do you choose if you (possibly) encountered control flow instructions earlier? Modern processors use branch predictors to make choices for them in these uncertain situations. Based on a branch prediction, the processor continues to execute instructions speculatively and maximally utilizes its pipeline. Sophisticated branch predictors are accurate enough to make this a very powerful optimization technique. In case of mispredictions, the pipeline is flushed and execution restarts at the correct instruction after the mispredicted branch.

Even instructions on the correct program path may get flushed from the pipeline sometimes, due to exceptions or interrupts. For example a page fault in a user program causes a pipeline flush and a jump to the kernel, which will handle the page fault. In case of a valid page fault, the kernel will map the missing page in and resume the user program, which will start executing the same instruction path as before.

All in all modern processor design gives rise to *transiently executed instructions*: instructions which at the microarchitectural level execute for some time, but which never retire and hence do not change the architectural state. Such transient instructions may or may not be the correct instructions to be executed according to the ISA.

2.2 Observing Microarchitectural Traces

Although transient instructions leave no architectural traces, observable through the ISA, they do leave microarchitectural traces. Processor designers have assumed for a long time that no (valuable) microarchitectural state can be observed architecturally. Hence, processors do not flush their complete microarchitectural state upon a pipeline flush. Such microarchitectural flushing would be very expensive. Moreover, many optimization techniques depend on persistent microarchitectural state after speculative execution, such as runahead execution [58].

Whereas processors do not provide a direct interface to inspect microarchitectural state, a lot of information about the microarchitectural state can actually be extracted via side channel analysis. An extensively studied example is given by cache timing side channels: based on memory access latency one can determine whether or not certain memory is cached, and even in which level of cache it resides. Many different methods have been explored to create covert channels based on memory caching, such as Flush+Reload [79], Flush+Flush [25], Flush+Prefetch [26], Evict+Reload [60], Evict+Time [59], Prime+Probe [60], Reload+Refresh [9], and Collide+Probe [52].

We elaborate on Flush+Reload [79], which is used by both Spectre and Meltdown, as well as many subsequent transient execution attacks. A sender encodes data into the cache, which a receiver retrieves via timing. The sender and the receiver share a read-only page of memory, e.g. due to copy-on-write optimizations or page deduplication. First, the receiver performs the Flush step: she flushes the first two cache lines from the shared memory, e.g. using the x86 instruction `CLFLUSH`. Next, the sender sends a bit by reading an address from one of the two cache lines: the first cache line signals a 0, the second cache lines signals a 1. Lastly, the receiver does the Reload step: she reads both of the two cache lines and times the respective latencies. Only the cache line accessed by the sender will be in cache, hence its access latency will be significantly lower. This enables the receiver to determine which cache line the sender accessed, and therefore the bit that was sent.

Next to memory caches, many other microarchitectural resources have also been used to construct timing side or covert channels: translation look-aside buffers [24], [12]; execution ports [73], [5], [8]; line fill buffers [67]; AVX units [68]; branch target buffers [1], [3], [2], [17], [49]; directional branch predictors [18]; cache way predictors [52]; AES accelerators [30]; memory buses [75]; DRAM row buffers [61]; and random number generators [16]. Non-timing side channels have been explored as well, which are for instance based on power analysis [51] or transactional memory [15].

2.3 Transient Execution Attacks

Transient execution attacks leverage transient processor execution and the observability of microarchitectural state to leak confidential data. At a high level, a transient execution attack consists of three phases:

1. Reset the microarchitectural state to a known baseline.

2. Transiently encode secret data in microarchitectural state.

3. Recover the secret data architecturally from the microarchitectural traces.

To familiarize the reader with transient execution attacks, we will take a look at these three phases in some known attacks: Spectre, Meltdown and RIDL.

2.3.1 Spectre

Spectre variant 1 [38] uses speculation on conditional branches to trigger transient execution and uses memory caches to encode secret data. For the covert channel we focus on the Flush+Reload technique in this example. We will attack the code snippet in Listing 1. Our goal will be to perform an out of bounds read on the `keys` array, say `keys[limit]`.

```
if (0 <= i && i < limit) {
        uint8_t key = keys[i];
        item_t item = store[key];
}
```

Listing 1: Spectre variant 1 gadget

The reset phase has to get both the conditional branch predictor and the cache into a baseline state. The branch predictor is trained to always take the branch, by repeatedly calling the code snippet with different valid values of `i`. This has the additional side effect that the contents of `keys` will reside in the cache. We make sure that the `store` array is flushed from the cache (the Flush step in Flush+Reload), and additionally we flush `limit` from the cache.

Next, the transient encoding phase is done by simply calling the code snippet with `i` having value `limit`. Architecturally, the `if`-statement will detect our invalid `i` and will hence not retrieve `keys[limit]`. Microarchitecturally however, the following will happen. To resolve the conditional branch, the processor needs to load `limit` from memory. This takes a long time, as is not cached, so in the meantime the processor starts speculative execution. We trained the branch predictor to choose the branch taken path. There, `keys[i]` (i.e. `keys[limit]`) is likely cached, either since it is in the same cache line as `keys[limit-1]` (which we cached during the reset phase) or due to the memory prefetcher. Hence the load to `key` is resolved very quickly. Then the processor issues the next load to `item`. This is where the encoding happens: as a side effect `store[key]` is cached. At some point the value of `limit` will be retrieved from memory and the processor will realize that it mispredicted a branch. The pipeline is flushed, but our microarchitectural trace of `store[key]` in the cache remains.

Lastly we perform the recover phase. Let us assume that `item_t` is at least the size of a cache line. Then our reset phase together with our transient encoding phase made sure that only a single entry of `store[0]`, `store[1]`, ..., `store[255]` is cached, namely `store[keys[limit]]`. By measuring all of their access times (the Reload step of Flush+Reload), we can leak the secret byte `keys[limit]`.

2.3.2 Meltdown

Meltdown [50] also uses memory caches for secret data encoding, but contrary to Spectre it triggers transient execution via exceptions. We consider a user process whose page tables contain the kernel mappings as well, protected via page table entry access bits (as was the case pre-Meltdown). Listing 2 shows how the user can leak kernel data via Meltdown, in particular the secret byte at kernel address `secret_addr`.

```
1  char *secret_addr = some_kernel_address;
2  char *buf = mmap_physically_contiguous(256*64);
3  flush(buf);
4  let_kernel_access(secret_addr);
5  buf[(*secret_addr) * 64];
6  char secret = reload(buf);
```

Listing 2: Meltdown attack

Assuming a cache line size of 64 bytes, we allocate a buffer consisting of 256 physically consecutive cache lines. Our reset phase firstly performs the Flush step from Flush+Reload on this buffer, to make sure none of its contents are cached. Secondly it tricks the kernel into accessing the secret, i.e. by issuing a system call that uses the secret, such that the secret gets cached.

Next, Line 5 causes a page fault, as we try to access the secret. But on the microarchitectural level, Line 5 transiently encodes the secret value in the cache. What happens is that the processor quickly retrieves the cached secret and forwards that value to the instructions in the pipeline which depend on it. The multiply by `64` can be performed and next the load from `buf` is issued to the memory system. Since these operations are so quick, they may happen *before* the load from `secret_addr` retires and raises a page fault.

We end up with the secret byte being encoded in the cache, recoverable via a Reload step from Flush+Reload. The only thing the user should ensure is that it actually reaches the reload step, i.e. that it recovers from the page fault. There are multiple mechanisms for doing so, such as Linux' `userfaultfd` or Intel's TSX.

2.3.3 RIDL

Rogue In-flight Data Load (RIDL) [67] transiently leaks data from the so-called line fill buffer (LFB). Modern Intel processors contain one LFB per core, which holds data that is being loaded from or stored to memory by any of the (hyper)threads on that core. During transient execution the processor occasionally uses data in the LFB as a quick prediction for data that is needed but not readily available. This enables an attacker to obtain, within a transient execution window, any (secret) data within the LFB, possibly belonging to sibling threads. By transiently encoding this data in microarchitectural state, for example in a data cache, the data can be leaked architecturally, e.g. using Flush+Reload. Hence RIDL can effectively leak arbitrary data that is being read from or written to memory by any sibling thread. RIDL is an example of a Microarchitectural Data Sampling (MDS) attack. MDS attacks are a subclass of transient execution attacks which sample data from microarchitectural buffers to leak information from sibling threads.

2.3.4 Locality of Transient Execution Attacks

We have seen three examples of transient execution attacks. They all traverse the three phases of transient execution attacks: reset, transient encode and recover.

At the heart of transient execution attacks lies transiently encoding secret data into microarchitectural state. This requires triggering transient execution. As we have seen, branch mispredictions and exceptions provide entry points into transient execution. Other entry points, like microcode assists and self modifying code, have been systematically examined by Ragab et al. [63]. If transient execution is to be triggered by the victim itself, such as with Spectre, then the attacker may need control over the victim's execution.

Once inside a transient execution window, one needs to acquire secret data. If the attacker performs transient execution, e.g. Meltdown or RIDL, then he/she must share microarchitectural resources with its victim, in which secret data resides.

Lastly, this secret data must be transmitted across a microarchitectural covert channel. If secrets are transiently encoded by the victim, then the attacker needs to share access to the covert channel with its victim.

In essence, transient execution attacks are enabled by the sharing of resources. Spectre needs to share access to the covert channel, e.g. the L1d cache, between attacker and victim. A Meltdown attacker depends on sharing its page tables with its victim. And RIDL relies on sharing the LFB with a victim. Hypervisor isolation prevents transient execution attacks by considerably reducing the amount of shared resources between distrusting parties, e.g. host and guest, namely all core-local resources.

We call a transient execution attack *core-local* if it can only be performed by an attacker that resides on the same core as the victim. RIDL clearly is a core-local attack, as the LFB is core-local. The original version of Spectre that uses the core-local L1d cache as its covert channel is core-local. But using a last level cache (LLC) shared among cores as your covert channel, Spectre can also be turned into a cross-core attack. In practice, Meltdown only seems to work if the secret data is loaded into the L1d cache by the victim during the reset phase, making Meltdown a core-local attack. But we cannot say with certainty that Meltdown is impossible with the secret data residing in the LLC, which would enable it to operate cross-core. Xiong et al. [77] examine the core-locality of (components of) known transient execution attacks. All 5 branch predictors they analyzed were core-local, which is relevant for the reset phase. Of the 14 covert channels they analyzed, 10 were core-local. And 19 out of 20 transient execution attacks they analyzed used a core-local covert channel. This indicates that a substantial part of transient execution attacks is core-local, and hence motivates our defense hypervisor isolation.

2.4 Traditional Microarchitectural Attacks

Transient execution attacks form a subclass of a more general type of attacks: microarchitectural attacks. Microarchitectural attacks are attacks that retrieve secret information from microarchitectural state to compromise victims. Although this thesis focuses on defending against transient execution attacks, our defense actually mitigates core-local microarchitectural attacks in general.

Traditional microarchitectural attacks, which do not abuse transient execution, have been known for a long time: already in 1996 seminal work by Kocher [39] showed how to leak secret cryptographic keys using a microarchitectural attack. Kocher used the timing characteristics of the memory hierarchy and CPU instructions to retrieve information about secret cryptographic keys.

Let us sketch an example of such a traditional microarchitectural attack. Suppose a cryptographic library is using a secret key for its encryption. Based on the first bit of the key, the library will use a different subroutine, which both take the same amount of time. An attacker could call the encryption function, which will execute only one of the two subroutines. After the encryption, one of the two subroutines is in the instruction cache, while the other is not. Next the attacker calls both of the two subroutines, and times their execution time. One call will finish considerably faster and the attacker can conclude that this subroutine must have been in the cache, and hence must have been called by the cryptographic library. In turn, this knowledge leaks the value of the first bit of the secret key.

2.5 Virtualization

In this section we will give a brief introduction to modern virtualization technology. This is the technology that lies at the heart of modern hypervisors and is therefore crucial for understanding hypervisor isolation.

2.5.1 Hardware Virtualization Extensions

Most modern computers provide hardware support for virtualization: the ability to run VMs on a physical machine. Usually, a VM runs its own fully fledged operating system (OS), also called the guest OS. The guest OS believes that it is running with the highest privilege level on a bare metal computer, directly on top of the hardware. Hence, it also thinks it has full access to all available resources, such as the CPU, memory, and hard disks. But in fact, the hypervisor controls all the actual hardware and only gives the guest access when it decides so. This is possible by means of hardware virtualization extensions, such as Intel's Virtual Machine Extensions (VMX) and AMD's Secure Virtual Machine (SVM).

These extensions introduce a new processor mode, called *guest mode*. Guest mode is entered via a dedicated *VM-start* instruction, such as Intel's `VMENTER` or AMD's `VMRUN`. In guest mode, certain instructions or events will be intercepted, which causes an exit from guest mode, called a *VM-exit*, that hands control over to the hypervisor. Typical examples of intercepted instructions are writes to control registers and accessing I/O ports. Exceptions and interrupts are commonly intercepted events.

Hypervisors maintain control over VMs by manipulating the machine state before a VM-start. Moreover, architectures like x86 provide explicit *VM Control Objects* (VMCO) that the hypervisor can use to control a VM. These are memory resident data structures whose contents influence the behavior of a VM. Among other things, a VMCO can be used to inject virtual interrupts into the guest, to configure which events cause a VM-exit and to inspect what caused a VM-exit after it happened.

2.5.2 The VM Life Cycle

In its life time, a VM repeatedly keeps VM-starting and VM-exiting. Before VM-starting the VM, the hypervisor must perform some VM-start preparation:

properly setting up the machine state for the VM. This includes setting up the VMCO properly, loading the guest's CPU state into the CPU registers, as well as updating the hypervisor's own bookkeeping. After VM-start preparation, the hypervisor executes the VM-start instruction. This causes a *world switch* from host to guest, switching the last essential CPU state like the stack pointer and the instruction pointer. From that moment onward, the guest runs directly on the hardware.

Upon a VM-exit the hardware performs a world switch back from guest to host, giving control back to the hypervisor. The hypervisor will then handle the VM-exit, i.e. inspect the cause of the VM-exit and take the appropriate action. For example a read from a control register could have caused the VM-exit, and the hypervisor may handle this by overwriting the guest's CPU register in which the result should arrive. Or the VM-exit may be due to an I/O interrupt, and the hypervisor may inject a corresponding virtual interrupt in the guest, or handle the interrupt itself. After the VM-exit has been handled, the VM can be resumed, as described above.

2.5.3 KVM

The hypervisor to which we applied hypervisor isolation is KVM: Kernel-based Virtual Machine [37]. KVM is a subsystem of the Linux kernel, enabling the kernel to function as a hypervisor. Within the infrastructure of Linux, a VM is a regular process. Different virtual CPUs of the VM correspond to threads of the VM-process. KVM exposes its API to userland via the `ioctl` system call on the KVM file `/dev/kvm`. This enables a user to, for example, create VMs and their virtual CPUs (vCPUs), run them, and receive information about the resulting VM-exits. Some VM-exits can be handled by KVM directly, such as writing to an MSR. Other VM-exits must be handled by the user process that created the VM, a typical example being the handling of peripheral device interaction. Note that aside from KVM, the user process driving KVM is also part of the hypervisor. QEMU is a typical example of such a user process.

3 Threat Model

We assume a modern computer is running a hypervisor without any traditional software vulnarabilities. The hypervisor facilitates potentially multiple untrusted clients, who are allowed to run VMs directly on top of the hardware. We assume the attacker to be one of these clients. The attacker has full control over its VMs, including what OS and applications are being run. In particular, as these VMs run directly on the hardware, the attacker is capable of executing microarchitectural attacks, incuding transient execution attacks. Using such attacks, the attacker tries to steal secret data belonging to either the hypervisor or one of the other clients.

We restrict the attacker to perform core-local attacks only: we do not defend against cross-core attacks. Although currently the majority of the known transient execution attacks is core-local [77], this is a limitation of hypervisor isolation.

4 Design

4.1 Design Goal

We have a single trusted and privileged security domain, the hypervisor, and multiple untrusted and unprivileged security domains, one for each client. The ideal goal of hypervisor isolation is to run each security domain on an isolated set of cores. More precisely, each security domain may only run on a designated set of cores, and no other security domain may run on this set of cores. This guarantees that no two security domains ever share any core-local resources and hence eliminates any core-local microarchitectural attacks across security domains.

This is in stark contrast with the design of contemporary hypervisors. Although separating VMs of different clients on separate cores is not new, modern hypervisors always run on the same cores as their VMs, as is depicted in Figure 1. This leaves the privileged hypervisor vulnerable and hence the entire system.

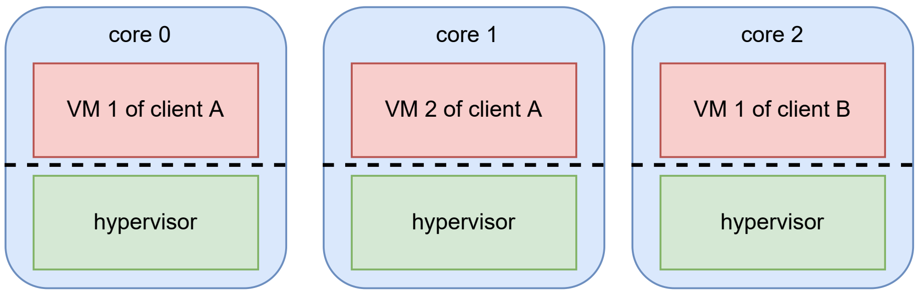

Figure 1: A contemporary hypervisor. The hypervisor runs on the same cores as its VMs.

With ideal hypervisor isolation, the picture looks as in Figure 2. The available cores are divided into host cores and guest cores. Host cores only run the hypervisor, while guest cores only run VMs. Additionally, every guest core only runs VMs from a single client. By design, this mitigates all core-local transient execution attacks across security domains.

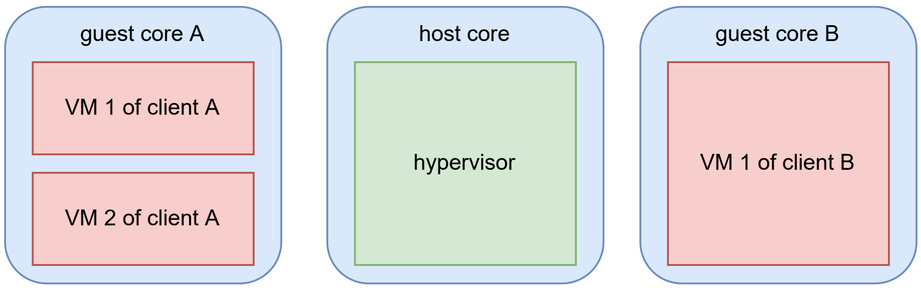

Figure 2: Ideal hypervisor isolation. Each security domain is isolated on a separate core.

Current hardware does however not support this ideal design. Virtualization technology assumes the hypervisor to run on the same core as its VMs. Starting up a VM can only be done from the same core and a VM-exit returns control to the hypervisor on the same core. And in order to maintain control over guest cores, the hypervisor must at least perform some scheduling there, which on modern processors can also only be done from the same core.

These hardware limitations force us to adjust our goal in practice. We still fully isolate VMs on guest cores. The hypervisor does however not solely run on host cores, but must also execute small *hypervisor stubs* of code on guest cores. The main responsibilities of these stubs are running VMs, communicating with host cores, and core-local scheduling. As the hypervisor stubs remain vulnerable to core-local microarchitectural attacks, we want to keep them as small as possible. Our hypervisor isolation prototype shows that it is practically feasible to let only a minor fraction of the hypervisor reside on guest cores, cf. Section 6.5. In the next sections we elaborate on our final design, which is depicted in Figure 3.

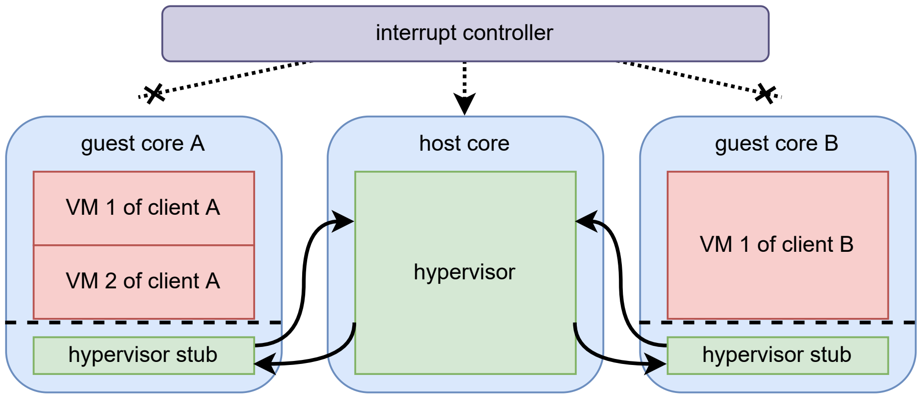

Figure 3: Final design of hypervisor isolation. Clients are fully isolated from each other on guest cores. The hypervisor runs isolated on host cores, except for its minimal hypervisor stubs on guest cores. The solid arrows represent VM-start and VM-exit messages. The dotted arrows represent interrupts.

4.2 Core Distribution

At hypervisor start up time we statically partition the available cores: part of the cores are marked host cores and a separate part is marked guest cores. Furthermore, the guest cores are statically distributed among the available clients.

We choose for a static core distribution for the sake of simplicity. We argue that a static distribution is sufficient in many cases, especially for the host versus guest core distribution. Modern virtualization technology is optimized for running the minimal amount of time in the host and the maximal amount of time in the guest. For hypervisor isolation this results in very few host cores being required compared to the amount of guest cores. On machines with a small or modest number of cores, a single host core will be sufficient for most workloads, as we shall see in Section 6.3. How to distribute clients among guest cores depends highly on the use case. A cloud computing scenario where customers rent fixed amounts of computing infrastructure from a cloud vendor would fare well with our static policy. But other use cases will need dynamic policies. Hypervisor isolation could be extended with such dynamic core distribution capabilities, using proper microarchitectural state flushing upon switching a core to a different security domain.

4.3 Physical Isolation

We enforce physical isolation of security domains to their respective cores on a process granular basis. We make the hypervisor's scheduler security domain aware: each process is scheduled only on the cores of the security domain to which that process belongs. In particular, almost all of the hypervisor's processes are scheduled on host cores only.

Upon each VM creation by a client, the hypervisor spawns a so-called *runner process* associated with that VM. The main responsibility of this runner process is running its VM. Although this runner process executes hypervisor code, like starting up its VM and regaining control upon VM-exits, it is marked for the scheduler as belonging to the client's guest security domain. This ensures the physical isolation of the corresponding VMs on the correct guest cores. Note that the runner processes themselves also run on guest cores and hence are part of the hypervisor stubs in Figure 3. All in all, each VM is managed by two hypervisor processes: a main process running on host cores, and a minimal runner process running on guest cores.

4.4 Communication between Host and Guest Cores

VMs need support from their hypervisor to keep running: their VM-exits must be handled, after which they need to be VM-started again. The runner processes on guest cores are in charge of performing VM-starts and catching VM-exits, but the handling of the VM-exits is done by the hypervisor on host cores. This requires communication between host and guest cores.

Hypervisor isolation employs per VM communication channels. Each runner process maintains a communication channel with the main hypervisor process on host cores in charge of the same VM. This communication channel facilitates two types of messages: *VM-start messages* and *VM-exit messages.* VM-start messages are sent from host to guest cores and signal the runner process to start up the VM. VM-exit messages are sent from guest to host cores and signal the hypervisor that the VM has VM-exited. The solid arrows in Figure 3 depict the VM-start and VM-exit messages.

Both host and guest cores need access to partly the same data, such as the VMCO and hypervisor maintained VM meta data. All of this data is shared via shared memory between the hypervisor on host cores and runner processes on guest cores.

4.5 The VM Life Cycle

Under hypervisor isolation, the VM life cycle looks as follows. Suppose the hypervisor, running on a host core, wants to start running a VM. As the first step, it performs as much VM-start preparation as possible from the host core. This minimizes the amount of work done in hypervisor stubs, hence it minimizes our residual attack surface. Next, the hypervisor sends a VM-start message to the VM's runner process. Upon reception, the runner performs the last core-local part of the VM-start preparation, like loading the guest's CPU state. Then it performs the actual VM-start instruction to run the guest. At some point, the VM will VM-exit and control flow will be returned to the runner. The runner saves the guest's CPU state and restores its own state. Subsequently, it sends a VM-exit message to a host core. Once the hypervisor on the host core receives the message, it handles the VM-exit. Once the VM-exit has been handled, we can restart the VM as above.

4.6 Interrupt Rerouting

On guest cores we would like to only run runner processes. But interrupts on guest cores will transfer control flow to interrupt handlers, whose code belongs to the hypervisor. Hypervisor isolation tries to minimize this host code that is run from interrupt context on guest cores by means of *interrupt rerouting*. Interrupt rerouting means programming the system wide interrupt controller to send most interrupts to host cores only. For example, we reroute all I/O interrupts to host cores. So when, say, a host side page fault occurs, the induced disk interaction is properly isolated on host cores. Interrupt rerouting is depicted with the dotted arrows in Figure 3. Some interrupts must still be delivered to guest cores. An example is the timer interrupt, which is needed by the hypervisor to maintain control over guest cores.

5 Implementation

We built a prototype of hypervisor isolation, called HYPISO, on top of KVM. During implementation we kept our changes non-invasive, keeping KVM almost entirely in tact. In total we changed/added about a thousand lines of code in 23 different files. As hypervisor isolation changes the fundamentals of KVM's design, architecture and vendor specific changes were necessary. As we had AMD hardware available, we decided to develop specifically for AMD, that is, the x86 architecture using AMD's SVM. Note that hypervisor isolation is a general concept that is not bound to AMD: it can also be implemented for other vendors or architectures, like Intel or ARM.

5.1 Initialization

During kernel boot up the available cores are split up into host and guest cores. HYPISO exposes a `sysfs` interface to configure the host versus guest core distribution before any VM has been created. Once the first VM has been created, dynamically changing this configuration is not supported anymore.

By hooking into the VM-creation `ioctl`-handling of KVM we spawn runner processes. This spawning is done by `clone`ing the user process that creates the VM, sharing as much as possible with the user process. For example, the cloned runner process must share its page tables with the user process, as we discuss in Section 5.5. But contrary to normal `clone` system call invocations, we make the runner process into a kernel thread and redirect its control flow to our own `hypiso_runner` kernel function. One can view runner processes using tools like `ps`: their names match "`runner-*`".

Process isolation on specific cores is implemented using the CPU affinity functionality of the Linux scheduler. For interrupt rerouting HYPISO uses Linux' SMP IRQ affinity functionality. In particular, all architecture independent interrupt requests (IRQs) are rerouted to host cores. This takes care of the overwhelming majority of all interrupts, as this for example includes all I/O IRQs.

5.2 VM Control Flow

We visualize the control flow of running a VM with KVM under hypervisor isolation in Figure 4. The user process orders KVM to run a VM via an `ioctl` system call. KVM first prepares most of the VM-start on a host core. Then a VM-start message is sent to the VM's runner at a guest core. The runner accepts the VM-start message and performs the CPU-local VM-start preparations, after which it performs the actual VM-start.

From here on, the guest takes control and executes its code in guest context until the occurrence of a VM-exit event. Such an event will return control to the runner on the guest core, which saves the guest state, recovers its own state and then sends a VM-exit message to the host core. On the host core KVM receives this message and is then in charge of handling the VM-exit. Some VM-exits are handled by KVM itself, while others must be handled by the user process owning the VM.

Handling a VM-exit involves checking the VM-exit code in the VMCB, indicating why the guest exited, and taking the appropriate actions. Suppose for example that the guest exits due to a page fault. Then KVM first checks if this concerns a “host-side” page fault, caused by the host swapping that page out. If this is the case, KVM issues a disk read and blocks the VM until the page has been swapped in, after which the VM-exit has been handled. If the page fault was a regular “guest-side” page fault, then KVM injects a page fault into the guest by setting a flag in the VMCB. Upon next VM-start, the guest will immediately be virtually interrupted by this injected page fault, which will be resolved by the guest’s own page fault handler. A typical example of a VM-exit that cannot be handled by KVM itself is virtual device emulation: this is the responsibility of the user process, such as QEMU.

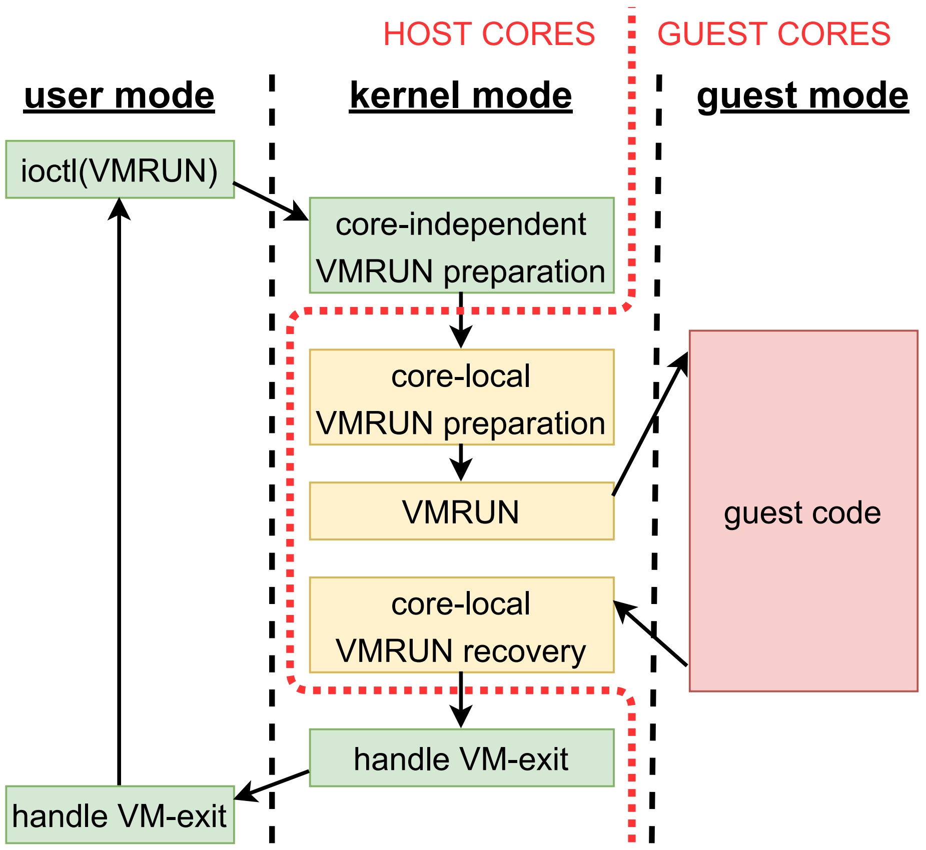

Figure 4: VM control flow in HYPISO. The dashed black lines separate CPU modes. The dotted red line separates code run on host and guest cores. The green blocks represent the protected part of the hypervisor. The yellow blocks are still vulnerable.

Figure 4 also visualizes the physical hypervisor isolation boundary. The red line separates code that runs on host cores versus code that runs on guest cores under hypervisor isolation. The arrows crossing these red lines correspond to VM-start and VM-exit messages, which essentially transfer control flow between cores.

5.3 Minimizing Hypervisor Stubs

The yellow blocks in Figure 4 resemble the KVM code that is run on guest cores, therefore breaking ideal isolation between hypervisor and VMs. To minimize these parts, we performed a thorough locality analysis of KVM's code path towards and after a VM-start and determined which parts of the code are CPU-independent and which parts require being run on the guest's CPU. Based on this analysis, we initially decided to only run the code in Listing 3 on guest cores.

```
vcpu->srcu_idx = srcu_read_lock(&vcpu->kvm->srcu);
preempt_disable();
static_call(kvm_x86_prepare_guest_switch)(vcpu);
local_irq_disable();
...
VMRUN
...
local_irq_enable();
preempt_enable();
```

Listing 3: KVM's critical section around a VM-start

We see that KVM uses a critical section around AMD's VM-start instruction `VMRUN`: preemption and interrupts are being disabled. This allows KVM to perform the very last VM-start preparation as well as the very first VM-exit recovery, like switching register state. Clearly, the code from Figure 3 must be run on the guest's CPU. We tried to let runner processes only run this critical section on guest cores, but this failed. KVM performs some time management before this critical section, which turned out to be CPU-dependent as well. Moving this time management code to the guest cores resulted in a functioning prototype.

Based on our core-locality analysis, we argue that the code executed by our runner processes on guest cores is minimal, within the bounds of non-invasive KVM changes and available hardware support.

5.4 Cross-core Communication

The VM-start and VM-exit message channels are implemented using shared memory. In particular, KVM maintains a data structure `struct kvm_vcpu` for each vCPU and we embedded our per-vCPU communication channel into this data structure, which is shared across host and guest cores.

Reception of VM-start messages is done by *collaborative polling* of the runner processes. Runners execute a loop in which they check for arrival of a VM-start message and give up the CPU in case of no message, i.e. calling `schedule`. As guest cores almost solely execute runner processes and there are usually a small (typically one) number of runners per guest core, this results in a low latency polling mechanism without hogging the CPU by a single runner.

VM-exit messages are received via (traditional) polling by the host core. This gives the lowest latency, but comes at the expense of many CPU cycles on the host core. To utilize host cores more efficiently, we experimented with different methods as well and we elaborate on these in Section 5.7.

When using shared memory as a cross-core synchronization primitive, as we do, one needs to be very careful with memory ordering. To maximize the memory system utilization, modern processors do not adhere to the strongest memory ordering models. Instead, they usually allow one core to perceive another core's memory operations out of program order. AMD does this as well, see Chapter 7 of [13] for details. This can raise problems, such as the one depicted in Table 1. In order to avoid these memory ordering problems, we made use of appropriate memory barriers [29].

host core	**guest core**
`vmcb->int_vector = INT_PF;`	`while(!vcpu->vm_start);`
`vcpu->vm_start = true;`	`vmrun(vmcb);`

Table 1: A memory ordering problem. The host core inserts a virtual page fault into the guest by writing to its VMCB. The guest core may perceive the write to the VMCB after the VM-start message, resulting in a `VMRUN` that reads an incorrect VMCB.

One last other important detail about our cross-core communication regards locks. VM-start and VM-exit messages cause a control flow switch between different CPUs. Since lock-ownership is CPU-bounded in Linux, hypiso ensures that a sender releases all of its locks before sending a message and that the corresponding receiver acquires the same locks again.

5.5 User Context Dependencies

KVM assumes that its code is being run in the context of the user process owning the VM. We break this assumption by running the VM in the context of a runner process. This forced us to patch some particular pieces of code. For example, KVM assumes at various locations that the CPU-local `current` variable, which always points to the process currently being run on that CPU, points to the user process which spawned the VM. We patched such code to use a pointer to the owning user process, instead of `current`.

Another subtle example is given by the page tables of the owning user process. KVM assumes that it has access to the complete virtual address space of the user process. Therefore, hypiso lets its VM-owning user processes share their page tables with their corresponding runner processes.

5.6 Implementation Limitations

hypiso is a research prototype, in contrast to a production ready product. Its goal is to explore the practical feasibility of hypervisor isolation for real world hypervisors. We think hypiso satisfies this goal. Nonetheless, hypiso lacks some properties of a completely sound implementation, which we discuss here.

First and foremost, we are certain that hypiso contains at least one bug. Its effects are very rare and nondeterministic user process crashes within guest VMs. Neither the hypervisor nor the guest OS has ever crashed and their state always seemed sound. We have seen userland processes getting killed due to several reasons: segmentation faults, virtual memory exhaustion, and "non-existent" files (which did exist). We have also seen (seemingly indefinite) hangs of user processes. Due to the nondeterministic behavior of these anomalies, we suspect the root cause to be a concurrency problem. Despite considerable debugging effort, we have not found the bug.

Although this bug may sound severe, we emphasize that these crashes are very rare. Moreover, when crashes do not occur, the system seems to function correctly in all aspects. During our evaluation of HYPISO, we discarded a few benchmarking runs which crashed. Apart from that, everything ran correctly and we think there is no reason to believe this bug invalidates our results.

Lastly HYPISO lacks some functionality one might expect to be present. HYPISO is not capable of properly shutting down virtual machines. Also HYPISO only supports rerouting interrupts upon host kernel boot and via a `sysfs` interface before any VM has started. HYPISO does not automatically make sure that interrupts keep being rerouted during the entire lifetime of the system. It may happen that a new device driver re-balances some interrupts at run time. During benchmarking we manually made sure this did not happen.

5.7 Core Multiplexing

The sections above described the implementation of our main prototype HYPISO. As HYPISO actively polls for a VM-exit message from a specific runner process while that runner process is executing its VM, the host core is not available while the guest core executes the VM. Our evaluation of HYPISO shows in Section 6.3 that the large majority of the time is spent in the guest. So next we tried to improve our prototype to efficiently multiplex the host core instead: let it serve multiple guest cores at the same time. This section describes these alternative prototypes.

Our first alternative, HYPISO-SCHEDULE, constantly calls `schedule` during its VM-exit message polling loop, i.e. it performs collaborative polling like our guest cores do as well for VM-start messages. We expect this design to still be fast when one host core serves a small number of guest cores, as the scheduler will run through all the VM-owning user processes on the host core that are collaboratively polling quite quickly, still resulting in a low latency. But for larger numbers of guest cores per host core, we do not expect this design to scale well.

A better scalable design, we predicted, is HYPISO-SLEEP. Under HYPISO-SLEEP, a VM-owning user process on the host core puts itself to sleep once it sends a VM-start message. Specifically it takes itself off the run queue, ensuring that the scheduler will not schedule it. The runner process on guest cores wakes up that user process on the host core once it sent its VM-exit message. The scheduler will reschedule the owning user process, which will consequently receive the VM-exit message.

Thirdly, we tried Linux' wait queue functionality instead of sleeping "by hand". Wait queues enable processes in Linux to wait for certain events to take place, such as, say, a VM-exit message. In particular, we used Linux' high level completion interface, which is built on top of wait queues. The corresponding prototype is called HYPISO-COMPLETE.

The designs above are simple and very compatible with the entire Linux kernel, hence we tried them first. We did however fear that they might induce a substantial overhead to the VM-exit message latency, if the scheduler's response time is lacking. We already had ideas on a fourth alternative prototype if this would turn out to be the case. This fourth, unimplemented prototype would bypass the Linux scheduler partly. We would deploy one kernel thread per host core, which would poll for VM-exit messages on a VM-exit queue.

Runner processes would enqueue their VM-exit messages on this VM-exit queue. Upon reception of a VM-exit message on the host core, the kernel thread would perform a context switch directly into the user process owning the VM that VM-exited, bypassing the Linux scheduler. We expect this to give better VM-exit message latencies, at the expense of compatibility. Given the implementation effort required, we focus on surveying the first three prototypes in this thesis. The fourth prototype can be considered in future work.

6 Evaluation

6.1 Setup

As our baseline to which we compare HYPISO we choose the vanilla Linux 5.15 kernel from which HYPISO was forked off. Both kernels run with the Linux kernel command line option `mitigations=off`, so we compare hypervisor isolation without any existing defenses enabled.

For our evaluation we use a *test machine* with an AMD Ryzen 5 5600X 6-Core Processor with 2 SMT threads per core and with 16GB of memory. This machine runs Ubuntu 20.04.4, modified to use either the HYPISO or the baseline kernel. Using QEMU 4.2.1 and their respective versions of KVM, we run an Alpine Linux 5.15.12-0-virt *test VM* with 2GB of memory.

For our nginx web server benchmark we generated a workload from an external *client machine*: an Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz and 32GB of memory running Ubuntu 20.04.4. The client machine is via Mellanox ConnectX 100GB NICs connected to our test machine, which passes its NIC directly through to the test VM using `vfio`.

We configure HYPISO to use one host core and one guest core and we give the test VM one vCPU. For the baseline we use two configurations, called optimistic and pessimistic, which run the test VM with respectively one and two vCPUs. Hence the optimistic baseline can only utilize one physical CPU, while the pessimistic baseline can utilize two. The optimistic baseline approximates a high core setup with very few host cores compared to many guest cores, whereas the pessimistic baseline assumes we need as many host cores as guest cores. Section 6.3 investigates which one is more realistic.

We ran all of our benchmarks 11 times and report the medians.

6.2 Performance

6.2.1 lmbench

We tested our implementation using the micro benchmark suite lmbench, which stresses the internals of an operating system. As lmbench is a single threaded benchmark, the optimistic and pessimistic baselines perform equally well, so we only display the pessimistic one. Table 2 shows the results of running lmbench from within our test VM.

For "Simple open/close" HYPISO incurs a significant performance overhead of 10.31%. But on all other benchmarks, HYPISO has roughly equal performance to the baseline. We conclude that hypervisor isolation does not incur significant overhead on lmbench.

To get deeper insight into the impact of VM-exit handling offloading to separate cores, we measured the VM-exit latencies during a full run of lmbench. Here we define VM-exit latency as the time between a VM-exit and a consecutive VM-start, so this includes all host code, including the hypervisor stubs executed on guest cores. The results are depicted in Figure 5.

We see that the VM-exit latency behavior is strikingly similar. The VM-exits take 3.2k cycles longer under hypervisor isolation, which is an increase of only 0.5%. Although on a microbenchmark scale, 3.2k cycles is quite a lot, this effect gets masked because VM-exits are so costly and, therefore, virtualization technology is designed to minimize the number of VM-exits.

Benchmark	baseline (μs)	HYPISO (μs)	overhead
Simple syscall	0.10	0.10	0.21 %
Simple read	0.14	0.14	0.36 %
Simple write	0.13	0.13	0.16 %
Simple stat	0.48	0.48	0.32 %
Simple fstat	0.19	0.19	0.48 %
Simple open/close	0.77	0.84	9.78 %
Select on 10 fd's	0.27	0.27	0.22 %
Select on 100 fd's	0.84	0.84	0.12 %
Select on 250 fd's	1.73	1.78	3.14 %
Select on 500 fd's	3.27	3.28	0.18 %
Select on 10 tcp fd's	0.31	0.31	0.20 %
Select on 100 tcp fd's	2.36	2.38	0.77 %
Select on 250 tcp fd's	5.79	5.85	1.06 %
Select on 500 tcp fd's	11.58	11.62	0.39 %
Signal install	0.14	0.14	0.35 %
Signal handler	0.49	0.49	0.37 %
Protection fault	0.27	0.27	1.62 %
Pipe latency	2.49	2.52	1.37 %
AF_UNIX sock stream	3.90	3.93	0.87 %
Process fork+exit	25.40	25.42	0.08 %
Process fork+execve	72.83	72.81	-0.02 %
Process fork+/bin/sh -c	175.71	176.77	0.61 %
Pagefaults	0.10	0.11	1.82 %
Local UDP latency	3.85	3.89	1.19 %
Local TCP latency	4.94	5.04	2.06 %
Local TCP/IP connection	13.32	13.64	2.35 %

Table 2: lmbench performance: baseline versus HYPISO.

6.2.2 nginx

We also tested our implementation on a long running real world application: the nginx web server. In our test VM we run nginx 1.20.2, configured to a maximum of 1024 concurrent connections and serving a static 64 byte file. Using the benchmarking tool `wrk` [20] on the client machine, we generate a workload for the nginx server in the test VM. Each time we run `wrk` for 30 seconds with four `wrk` client threads and an increasing number of connections. The results are summarized in Figure 6.

We see that the performance of HYPISO is most similar to the optimistic baseline. This is because they can both utilize a single nginx worker thread at any point in time. In contrast, the pessimistic baseline can run two nginx worker threads concurrently. The CPU utilization of all versions saturates at 128 concurrent connections. At that point, HYPISO experiences 4.9% and 49.0% performance degradation compared to the optimistic and pessimictic baselines respectively. For nginx we conclude that hypervisor isolation induces small overhead compared to the optimistic baseline, but has half the throughput of the pessimistic baseline.

As we did for lmbench, we measured the VM-exit latencies while running nginx. We choose to run with 256 concurrent connections, as this gives full CPU saturation. As the VM-exit latency behavior per vCPU of both of our baselines was almost identical, we only report the pessimistic baseline. See Figure 7 for

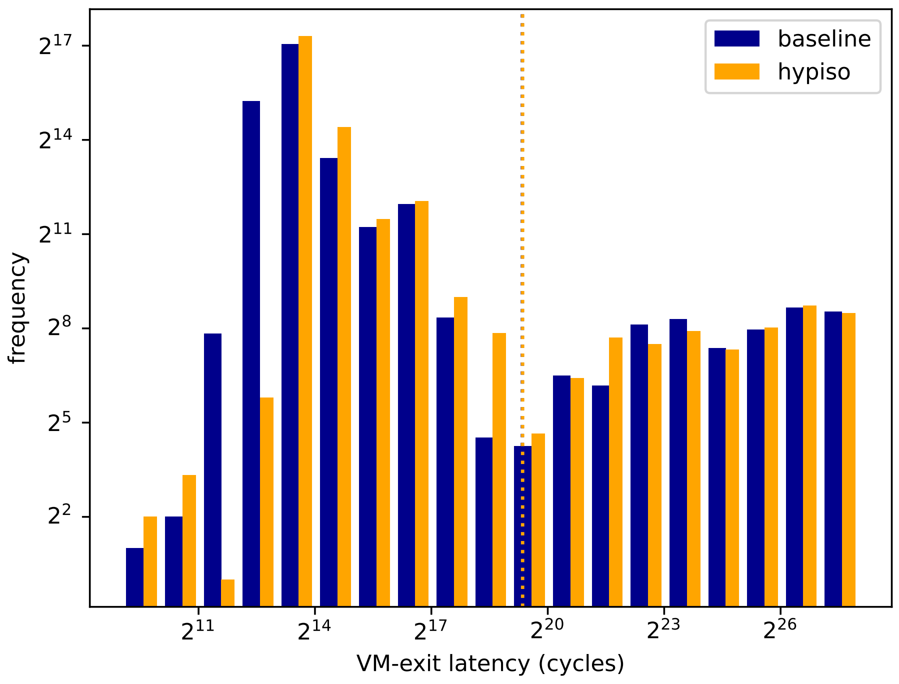

Figure 5: VM-exit latencies during a full run of lmbench. The dotted vertical lines indicate the average VM-exit latencies, which are 665.7k for the baseline and 668.9k for HYPISO.

the corresponding VM-exit latency histogram.

The VM-exit latencies are substantially higher for HYPISO: a factor 4.2. This factor 4.2 stems from two things: HYPISO VM-exits 2.2 times less often then the baseline, while it spends 1.9 times more time in host context. We interpret these numbers as follows. An 3.2k cycles increase in latency was very minor for lmbench—whose VM-exits are very expensive—but are major for nginx: more than doubling its VM-exit latency. For nginx, most of these VM-exits are caused by interrupts from the NIC. If the time to react to these interrupts doubles, then the amount of network packets that is ready to be handled increases. Hence a single VM-exit processes more network packets, but also takes longer. These effects barely harm the single vCPU throughput of nginx, as we saw above.

6.3 Core Utilization

We investigate the core utilization of HYPISO, to see which of our baselines is more representative for real world deployments. To this end, we measured the time we spent on host and guest cores respectively during the VM life cycle of Figure 4. So for the host core we accumulate the time between reception of a VM-exit message and the transmission of a VM-start message, excluding the time we spent polling for the VM-exit message. Similarly for the guest core we measure the time between receiving VM-start messages and sending VM-exit messages. We did this for both lmbench and nginx (at saturation), with setups

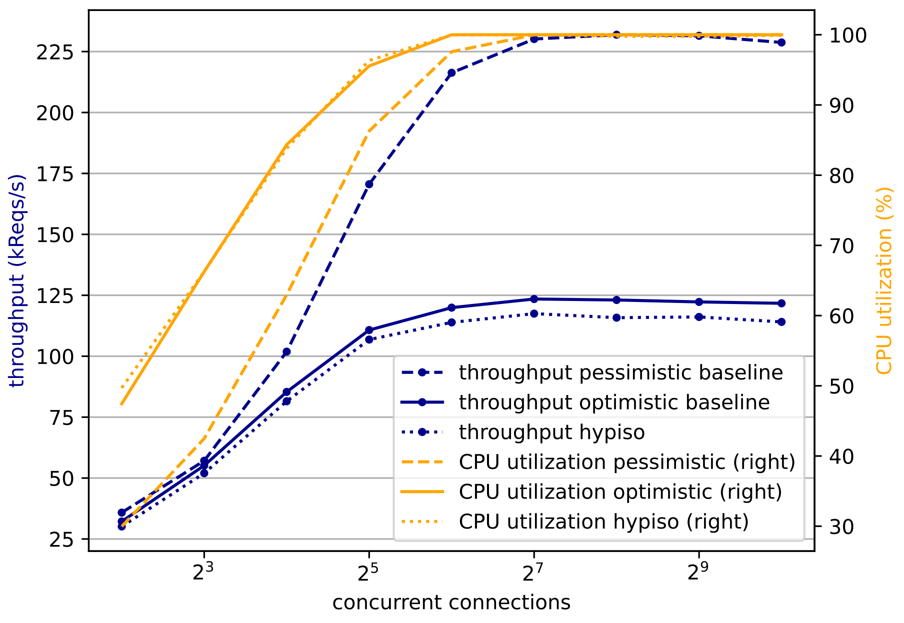

Figure 6: Performance of nginx on HYPISO compared to the baselines

	host core	guest core
lmbench	18.3%	81.7%
nginx	7.2%	92.8%

Table 3: Time HYPISO spent on host and guest cores.

the same as above. The results are summarized in Table 3.

We see that for lmbench, designed to stress the OS, we still only spent 18.3% of our time on the host core. For nginx, a more realistic real world application, even less time is spent on host cores: only 7.2%. This indicates that our current design can be significantly improved upon. Instead of polling for a VM-exit message from a runner process on host cores, we could try to concurrently handle VM-exit messages from multiple guest cores. If this could be efficiently implemented, then for a real world application like nginx one could serve 14 guest cores per host core. This would mean the optimistic baseline would be closely approximated. Our current version of HYPISO however still needs support from two physical CPUs to run a single virtual CPU, so in that case the pessimistic baseline is appropriate.

We foresee two potential new performance problems that could arise due to multiplexing of one host core to serve multiple guest cores. First of all, since VM-exits need to be handled using the page tables of the owning user process, serving multiple VMs from one host core would require the host core to perform a large number of context switches: in the worst case upon every VM-exit message. We simulated this behavior for our saturated nginx benchmark by flushing the TLB of the host core upon reception of every VM-exit message.

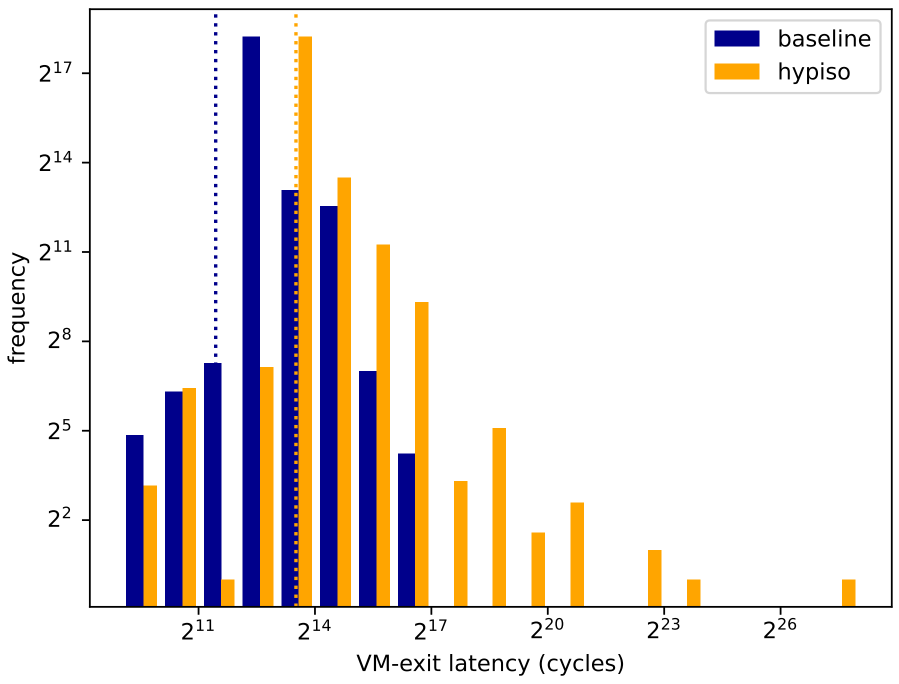

Figure 7: VM-exit latencies on one vCPU for 30 seconds of nginx handling 256 concurrent connections. The dotted vertical lines indicate the average VM-exit latencies, which are 2781.5 for the baseline and 11666.8 for HYPISO.

This did not degrade performance at all. This simulation is far from perfect of course, e.g. memory caches might also get trashed.

Secondly, instead of actively polling for a VM-exit message, the host core would part of the time be busy handling VM-exits of other VMs. This may increase the average VM-exit latency.

This led us to design the alternative prototypes from Section 5.7, which the next section evaluates.

6.4 Scalability

Based on our observations from Section 6.3, we implemented the alternative designs from Section 5.7 with the hope to better utilize a many core machine. This section discusses the evaluation we performed on those alternative designs.

Instead of the relatively small 6 core testmachine, we moved to an AMD Ryzen Threadripper 2990WX 32-Core Processor with 2 SMT threads per core and with 128GB of memory. It ran Ubuntu 20.04.4 on top of one of our prototype Linux kernels. We ran the same Alpine Linux 5.15.12-0-virt test VM, this time with 64GB of memory, using QEMU 4.2.1. During our experiments we varied the number of vCPUs of the test VM. This VM ran nginx 1.20.2 configured to as many worker threads as the VM has vCPUs, a maximum of 1024 concurrent connections and serving a static file of 64 bytes.

Due to unforeseen hardware problems, we did not have access to an external client machine connected to our test machine to generate the workload. Therefore, the same machine also ran `wrk`, on separate cores from the cores that ran the VM. With 8 worker threads and 1024 concurrent connections, `wrk` ran for 30 seconds.

We ran the benchmark using the different prototypes, described in Section 5.7, multiple times with an increasing number of vCPUs for the test VM. We configured hypervisor isolation to use one host core and as many guest cores as the VM had vCPUs. Our baseline is running with hypervisor isolation disabled, i.e. the vanilla Linux kernel. We measured the throughput and latency of the web server, as well as the CPU utilization of the different cores. We also instrumented our kernels to track the latency of six specific phases in the VM life cycle: the "host" latency on the host core, the "start_msg" latency between sending and receiving a VM-start message, the "start_stub" latency of preparing the VM-start on the guest core, the "guest" latency of the guest VM running, the "exit_stub" latency of the guest core recovering from the VM-exit, and the "exit_msg" latency between sending and receiving the VM-exit message. We hoped to get detailed insight into which parts of this life cycle take a long time in our different prototypes.

Figure 8 shows the relative time we spent in each of these phases during a full 30s benchmark. The HYPISO-COMPLETE prototype had very similar results to HYPISO-SLEEP, so for visibility we excluded it from the figure. We see that we spent most of our time in the host. This is problematic for our setup, as we expected to run only approximately 7% of the time in the host, and could hence serve multiple guest cores with a single host core. Now that we spent the majority of the time in the host, this setup is not very meaningful anymore. The cause of this problem lies in the local generation of the workload. Instead of using direct device assignment of the NIC to the test VM as in Section 6.2, the host must now simulate a virtual NIC for the test VM, which is relatively slow. Moreover, device emulation is done in userland, i.e. by QEMU, hence inducing context switches between kernel and user mode upon every device interaction.

To run this experiment properly, we should again use a dedicated NIC which we pass through to the test VM. As we did not have such a NIC available, we could not finish these experiments.

Although the above circumstances make interpreting these results difficult, we do see high amounts of time spent in the "exit_msg" phase for the hypervisor isolation prototypes. This indicates that the VM-exit message latency might indeed be a problem in these designs. These initial results warrant this problem to be further investigated in future work.

6.5 Effectiveness

HYPISO uses hypervisor isolation to remove the most fundamental building block of any core-local transient execution attack: shared microarchitectural state. Without shared microarchitectural state between the host and the guest, it is impossible for the guest to launch any transient execution attack against the host—both known and future ones. In that sense, HYPISO provides a very strong protection mechanism. But as we saw, due to hardware enforced limitations, ideal hypervisor isolation is impossible and HYPISO aims to reduce the attack surface as much as possible. In this section, we quantify the remaining attack surface left in hypervisor stubs on guest cores. We do this for our main

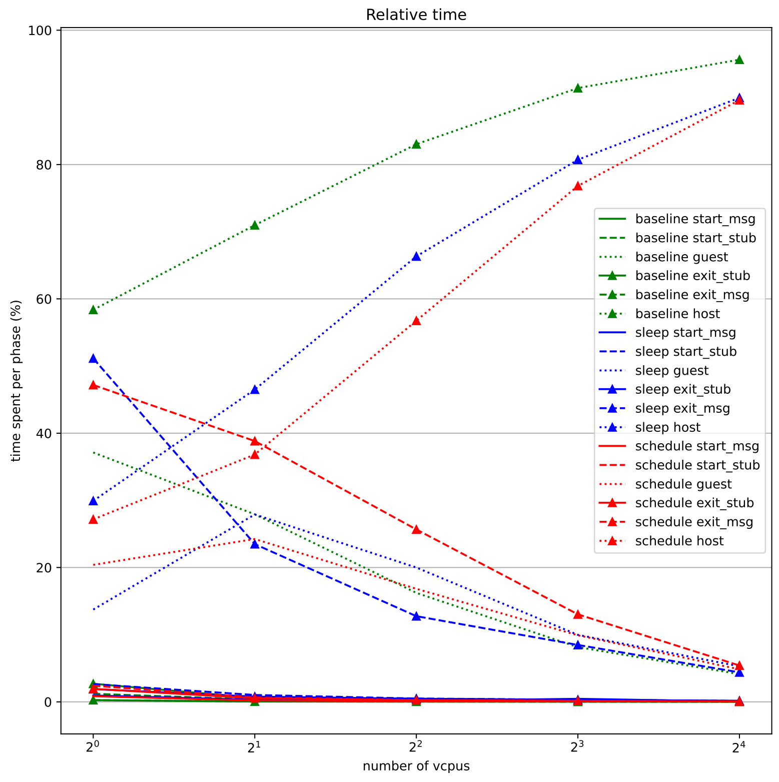

Figure 8: Relative time spent in each of the six phases in the VM life cycle, for our HYPISO-SCHEDULE and HYPISO-SLEEP prototypes, as well as the baseline.

prototype, HYPISO.

We measure the amount of code on a function granular scale, assuming the amount of potential transient execution gadgets is approximately proportional to the amount of vulnerable functions an attacker has access to. Our hypervisor consists of two parts: QEMU and KVM. We statically determined the number of functions within QEMU: 9831. Assuming a fictive example attacker can hire many different virtual devices on a cloud platform and use fuzzing techniques, we should assume that all of QEMU is potential attack surface.

A similar static analysis for KVM is harder, due to the tight incorporation of KVM into the Linux kernel. We think taking the whole kernel as potential attack surface would be a big overestimation. Hence for KVM we used Linux' function tracing capabilities to dynamically trace which functions are called, during another run of our nginx benchmark with 256 connections. This resulted in 2113 unique host kernel functions being called on for the baseline, while HYPISO only executes 297 unique host kernel functions on its guest cores.

Combining these results, we find that the baseline attack surface is 11944 host functions. Under HYPISO, the remaining attack surface is 297 functions, 2.5% of the baseline.

We inspected the 297 host functions that constitute the hypervisor stubs on our guest cores. The carefully chosen host code that our runner processes execute on guest cores contribute 48 functions. The scheduler is responsible for a big part of the hypervisor stub: 167 functions. With 5 functions being used by both the runner and the scheduler, this leaves 87 functions unaccounted for. These have a variety of origins: remaining interrupt handlers, the `eventfd` subsystem, wait queues, the watchdog, and the RCU subsystem.

As discussed in Section 5.3, we already tried to minimize the amount of host code in runner processes. And clearly, to remain in control of the guest cores, it is essential to run the scheduler on guest cores. A careful analysis might reveal that some of the leftover functions could still be isolated on host cores. But we expect the majority to be essential for the correct functioning of the Linux kernel, and hence KVM.

We conclude that HYPISO manages to achieve an impressive 97.5% reduction of attack surface.

7 Related Work

Already in 2007, Kumar et al. [41, 19] explored the idea of *sidecores*: cores dedicated to performing specific hypervisor functionality. Since then lots of research has been done on improving virtualized I/O performance by using I/O sidecores [54, 7, 23, 27, 78, 48, 43]. The main idea is to move the part of the hypervisor responsible for I/O to dedicated cores, in order to minimize the number of I/O induced VM-exits and hence optimize performance. This is in contrast with our goal of providing stronger security for the hypervisor. Instead of separating a specific part of the hypervisor to specific cores, we try to isolate the full hypervisor on isolated cores.

Landau et al. [44] explored the idea of splitting guest and hypervisor execution on separate cores in 2011. Their goal is to improve virtualization performance, in contrast to our goal of improving security. Moreover, they explain that such split hypervisor/guest execution is not possible on existing hardware and they describe a hardware model in which this would become feasible. HYPISO actually is a close approximation of "split execution" on contemporary hardware.

Manuel Wiesinger from VUSec recently did a project on system call isolation, which is essentially hypervisor isolation with the hypervisor replaced by the OS kernel and VMs replaced by userland processes. By isolating the kernel on a separate set of cores, system call isolation defends the kernel against transient execution attacks. His results indicate that system call isolation is prohibitively expensive to use in practice. In contrast, hypervisor isolation has comparable overhead to current state of the art spot mitigations combined, making it a potentially feasible defense in practice. This performance gap stems from the ubiquity of system calls in user applications, whereas virtualization technology is optimized to minimize the amount of VM-exits.

Google worked on `coresched` patches [22] for the Linux kernel to mitigate core-local transient execution attacks in the following way. Processes can be configured to (dis)trust each other and the scheduling patch enforces that distrusting security domains never run on the same core at the same time. The main motive is to keep simultaneous multi-threading alive in the transient execution attack era. This idea is similar to our efforts because it not only protects user-to-user attacks, but also user-to-kernel attacks. But `coresched` only prevents spacial core sharing, while still allowing temporal core sharing, in contrast to hypervisor isolation.

Back in 2010, Keller et al. [34, 71] identified core-local side channel attacks as a security risk for cloud deployments. In order to protect VMs from each other, they suggest a cloud architecture without a hypervisor in which physical resources, like cores, memory, and devices, are divided among VMs. Clearly, this design lacks many flexibility features of virtualization as we know it today. In contrast, HYPISO supports full modern hypervisor functionality.

Moon et al. [57] propose a moving target defense for virtual machines against co-residency side channel attacks. They try to prevent co-residency by nomadizing VMs. By only moving VMs around, their design leaves the hypervisor itself vulnerable. The novelty of this thesis lies in the elimination of co-residency of the hypervisor with its VMs.

An orthogonal defense method which provides similar security guarantees as hypervisor isolation is the minimization of the amount of memory that the hypervisor keeps mapped in its page tables. Xia et al. [76] demonstrated this with their secret-free Xen hypervisor and similar efforts are being tried out for KVM under the name Address Space Isolation (ASI) [4]. As unmapped memory is not accessible, not even for the microarchitecture during transient execution, secrets residing in unmapped memory are safe.

8 Future Work

Implementing our first hypervisor isolation prototype HYPISO already took us a considerable amount of time, due to the complexity of the Linux kernel and in particular KVM. This left multiple ideas, additional features and alternative designs unexplored, which we will sketch in this section.

First of all, the implementation limitations discussed in Section 5.6 should be addressed, in particular the bug(s) causing sporadic userland crashes within the guest. Furthermore, finishing the experients of Section 6.4 is essential to understand the scalability of hypervisor isolation. If experiments indicate poor performance, one could try to implement the fourth alternative design from Section 5.7. Performing a more elaborate evaluation of our prototypes would in general be beneficial, for example with more diverse benchmarks.

Most transient execution attacks until now have used the L1 or L2 caches for their covert channels. Hypervisor isolation mitigates such attacks, as these are core-local caches. In theory such attacks could however make use of the LLC instead, which is shared across cores. By using cache partitioning technology, such as Intel's CAT [53] or AMD's PQE [14], one could separate the LLC into host and guest parts. The host (respectively guest) part may only be used by host (respectively guest) cores. This would improve hypervisor isolation's security guarantees significantly, as it would completely eliminate the memory cache as covert channel. Future work could implement this, investigate different host/guest partition size ratios, and evaluate the performance implications.

Hypervisor stubs leave a small residual attack surface on guest cores. This particularly risks a hypervisor address being leaked to a VM on a guest core, which would break KVM's KASLR entirely, also on host cores. To mitigate this risk, one could deploy host/guest granular KASLR: randomize the host kernel's address space independently on host and guest cores. This prevents breaking KASLR on host cores by leaking a kernel pointer on a guest core. We expect this to have little, if any, performance impact while having significant security benefits, as opposed to a more drastic approach like Function Granular KASLR [55].

As ASI is an orthogonal defense method, it could be applied on top of hypervisor isolation. Current ASI implementation efforts face problematic complexity issues [33]. Also it is difficult to determine what pages KVM should (un)map, i.e. where sensitive data resides. Hypervisor isolation provides an opportunity for a very simplistic ASI implementation that does not suffer from these problems. The idea is demand paging hypervisor memory on guest cores. The hypervisor starts out with a very minimal page table on guest cores: just enough to handle its own page faults. We modify the page fault handler to map in any missing pages on guest cores. Hypervisor demand paging requires drastically less invasive changes to the kernel, and trivially solves the problem of what pages to (un)map: by design we only map the necessary pages.

Lastly, hypervisor isolation could greatly benefit from better hardware support, such as described by Landau et al. [44]. In particular, hardware support for scheduling and starting/stopping VMs from remote cores could greatly reduce the residual attack surface of hypervisor isolation, as well as improve its performance.

9 Conclusion

This thesis proposed hypervisor isolation as a comprehensive, future proof defense against core-local microarchitectural attacks, in particular transient execution attacks. Isolating the hypervisor on separate cores eliminates shared core-local resources and hence the attacks that rely on them. We described a design of hypervisor isolation on modern commodity hardware and presented an implementation HYPISO of hypervisor isolation in a real world hypervisor: KVM. Our performance evaluation shows that hypervisor isolation induces very low overhead if two physical cores are dedicated per virtual core. We presented designs that would improve upon this doubling of effective core utilization, but additional research is required to properly assess their performance in large scale core configurations. Our effectiveness analysis shows hypervisor isolation is capable of reducing the core-local transient execution attack surface by 97.5% in the real world hypervisor KVM. Further research into hypervisor isolation may result in an efficient and strong defense against transient execution attacks that could be deployable in production systems.

References

[1] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. “On the power of simple branch prediction analysis”. In: *Proceedings of the 2nd ACM symposium on Information, computer and communications security*. 2007, pp. 312–320.

[2] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. “New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures”. In: *IMA International Conference on Cryptography and Coding*. Springer. 2007, pp. 185–203.

[3] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. “Predicting secret keys via branch prediction”. In: *Cryptographers’ Track at the RSA Conference*. Springer. 2007, pp. 225–242.

[4] *Address Space Isolation for KVM.* `https://lore.kernel.org/lkml/20220223052223.1202152-1-junaids@google.com`. 2022.

[5] Alejandro Cabrera Aldaya et al. “Port contention for fun and profit”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 870–887.

[6] AMD. *Speculative Store Bypass Disable.* `https://developer.amd.com/wp-content/resources/124441_AMD64_SpeculativeStoreBypassDisable_Whitepaper_final.pdf`. 2018.

[7] Nadav Amit et al. “vIOMMU: Efficient IOMMU Emulation”. In: *2011 USENIX Annual Technical Conference (USENIX ATC 11)*. 2011.

[8] Atri Bhattacharyya et al. “Smotherspectre: exploiting speculative execution through port contention”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 785–800.

[9] Samira Briongos et al. “RELOAD + REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks”. In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 1967–1984.

[10] Jo Van Bulck et al. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution”. In: *USENIX Security’18*.

[11] Claudio Canella et al. “A systematic evaluation of transient execution attacks and defenses”. In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019, pp. 249–266.

[12] Claudio Canella et al. “Fallout: Leaking Data on Meltdown-resistant CPUs”. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM. 2019.

[13] Advanced Micro Devices. *AMD64 architecture programmer’s manual volume 2: System programming.* revision 3.38. Nov. 2021.

[14] Advanced Micro Devices. *AMD64 Technology Platform Quality of Service Extensions.* revision 1.00. Aug. 2018.

[15] Craig Disselkoen et al. “Prime+ Abort: A Timer-FreeHigh-Precision L3 Cache Attack using Intel TSX”. In: *26th USENIX Security Symposium (USENIX Security 17)*. 2017, pp. 51–67.

[16] Dmitry Evtyushkin and Dmitry Ponomarev. “Covert channels through random number generator: Mechanisms, capacity estimation and mitigations”. In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security.* 2016, pp. 843–857.

[17] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. “Jump over ASLR: Attacking branch predictors to bypass ASLR”. In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* IEEE. 2016, pp. 1–13.

[18] Dmitry Evtyushkin et al. “Branchscope: A new side-channel attack on directional branch predictor”. In: *ACM SIGPLAN Notices* 53.2 (2018), pp. 693–707.

[19] Ada Gavrilovska et al. “High-performance hypervisor architectures: Virtualization in hpc systems”. In: *Workshop on system-level virtualization for HPC (HPCVirt).* Citeseer. 2007.

[20] Will Glozer. *wrk*. `https://github.com/wg/wrk`.

[21] Enes Göktas et al. “Speculative Probing: Hacking Blind in the Spectre Era”. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security.* 2020, pp. 1871–1885.

[22] Google. *Core-scheduling.* `https://www.phoronix.com/scan.php?page=news_item&px=Google-Core-Scheduling-v9`, `https://lore.kernel.org/lkml/20201117232003.3580179-1-joel@joelfernandes.org/`. 2020.

[23] Abel Gordon et al. “Towards exitless and efficient paravirtual I/O”. In: *Proceedings of the 5th Annual International Systems and Storage Conference.* 2012, pp. 1–6.

[24] Ben Gras et al. “Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks”. In: *USENIX Security’18.*

[25] Daniel Gruss et al. “Flush + Flush: A Fast and Stealthy Cache Attack”. In: *DIMVA’16.*

[26] Daniel Gruss et al. “Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR”. In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security.* 2016, pp. 368–379.

[27] Nadav Har’El et al. “Efficient and Scalable Paravirtual I/O System”. In: *2013 USENIX Annual Technical Conference (USENIX ATC 13).* 2013, pp. 231–242.

[28] Jann Horn. *Speculative Store Bypass.* `https://bugs.chromium.org/p/project-zero/issues/detail?id=1528`. May 2018.

[29] David Howells et al. *Linux kernel memory barriers.* `https://www.kernel.org/doc/Documentation/memory-barriers.txt`.

[30] Casen Hunger et al. “Understanding contention-based channels and using them for defense”. In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA).* IEEE. 2015, pp. 639–650.

[31] Intel. *Rogue System Register Read / CVE-2018-3640 / INTEL-SA-00115.* `https://software.intel.com/security-software-guidance/software-guidance/rogue-system-register-read`. May 2018.

[32] Intel. *Speculative Execution Side Channel Mitigations.* https://www.intel.com/content/dam/develop/external/us/en/documents/336996-speculative-execution-side-channel-mitigations.pdf. 2018.

[33] *Issues with ASI and coresched's kernel protection.* https://lore.kernel.org/lkml/91dd5f0a-61da-074d-42ed-bf0886f617d9@oracle.com/. 2022.

[34] Eric Keller et al. "Nohype: virtualized cloud infrastructure without the virtualization". In: *Proceedings of the 37th annual international symposium on Computer architecture.* 2010, pp. 350–361.

[35] Russell King. *spectre-v2: harden branch predictor on context switches.* https://patchwork.kernel.org/project/linux-arm-kernel/patch/E1fMDH3-0006yt-Fn@rmk-PC.armlinux.org.uk/. 2018.

[36] Vladimir Kiriansky and Carl Waldspurger. "Speculative buffer overflows: Attacks and Defenses". In: *arXiv'18.* https://doi.org/10.48550/arXiv.1807.03757.

[37] Avi Kivity et al. "KVM: the Linux Virtual Machine Monitor". In: *Proceedings of the Linux symposium.* Vol. 1. Dttawa, Dntorio, Canada. 2007, pp. 225–230.

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

[39] Paul C Kocher. "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems". In: *Annual International Cryptology Conference.* Springer. 1996, pp. 104–113.

[40] Esmaeil Mohammadian Koruyeh et al. "Spectre Returns! Speculation Attacks using the Return Stack Buffer". In: *USENIX WOOT'18.*

[41] Sanjay Kumar et al. "Re-architecting VMMs for multicore systems: The sidecore approach". In: *Workshop on Interaction between Opearting Systems & Computer Architecture (WIOSCA).* Citeseer. 2007.

[42] Hsuan-Chi Kuo et al. "A Linux in unikernel clothing". In: *EuroSys.* 2020.

[43] Yossi Kuperman et al. "Paravirtual remote i/o". In: *ACM SIGARCH Computer Architecture News* 44.2 (2016), pp. 49–65.

[44] Alex Landau, Muli Ben-Yehuda, and Abel Gordon. "SplitX: Split Guest/Hypervisor Execution on Multi-Core". In: *3rd Workshop on I/O Virtualization (WIOV 11).* 2011.

[45] Michael Larabel. *Bisected: The Unfortunate Reason Linux 4.20 Is Running Slower.* https://www.phoronix.com/scan.php?page=article&item=linux-420-bisect. Nov. 2018.

[46] Michael Larabel. *In Light Of Spectre BHI, The Performance Impact For Retpolines On Modern Intel CPUs.* https://www.phoronix.com/scan.php?page=article&item=spectre-bhi-retpoline&num=1. Mar. 2022.

[47] Michael Larabel. *The Brutal Performance Impact From Mitigating The LVI Vulnerability.* https://www.phoronix.com/scan.php?page=article&item=lvi-attack-perf&num=1. Mar. 2020.

[48] Dongwoo Lee, Changwoo Min, and Young Ik Eom. "vCanal: Paravirtual Socket Library towards Fast Networking in Virtualized Environment". In: *IEICE TRANSACTIONS on Information and Systems* 99.2 (2016), pp. 360–369.

[49] Sangho Lee et al. "Inferring fine-grained control flow inside SGX enclaves with branch shadowing". In: *26th USENIX Security Symposium (USENIX Security 17)*. 2017, pp. 557–574.

[50] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[51] Moritz Lipp et al. "PLATYPUS: Software-based Power Side-Channel Attacks on x86". In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2021.

[52] Moritz Lipp et al. "Take a way: Exploring the security implications of AMD's cache way predictors". In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 2020, pp. 813–825.

[53] Fangfei Liu et al. "Catalyst: Defeating last-level cache side channel attacks in cloud computing". In: *HPCA*. 2016, pp. 406–418.

[54] Jiuxing Liu and Bulent Abali. "Virtualization polling engine (VPE) using dedicated CPU cores to accelerate I/O virtualization". In: *Proceedings of the 23rd international conference on Supercomputing*. 2009, pp. 225–234.

[55] Alexander Lobakin and Kristen Accardi. *Function Granular KASLR*. `https://lore.kernel.org/lkml/20211202223214.72888-1-alexandr.lobakin@intel.com/`.

[56] Giorgi Maisuradze and Christian Rossow. "ret2spec: Speculative Execution using Return Stack Buffers". In: (2018).

[57] Soo-Jin Moon, Vyas Sekar, and Michael K Reiter. "Nomad: Mitigating arbitrary cloud side channels via provider-assisted migration". In: *CCS*. 2015.

[58] Onur Mutlu et al. "Runahead execution: An effective alternative to large instruction windows". In: *IEEE Micro* 23.6 (2003), pp. 20–25.

[59] 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.

[60] Colin Percival. *Cache missing for fun and profit (2005)*. 2005.

[61] Peter Pessl et al. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks". In: *25th USENIX security symposium (USENIX security 16)*. 2016, pp. 565–581.

[62] Hany Ragab et al. "CrossTalk: Speculative Data Leaks Across Cores Are Real". In: *S&P*. Intel Bounty Reward. May 2021. URL: `Paper=https://download.vusec.net/papers/crosstalk_sp21.pdf%20Web=https://www.vusec.net/projects/crosstalk%20Code=https://github.com/vusec/ridl%20Press=https://bit.ly/3frdRuV`.

[63] Hany Ragab et al. "Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks". In: *USENIX Security*. Aug. 2021. URL: `Paper=https://download.vusec.net/papers/fpvi-scsb_sec21.pdf%20Web=https://www.vusec.net/projects/fpvi-scsb%20Code=https://github.com/vusec/fpvi-scsb`.

[64] Charles Reis, Alexander Moshchuk, and Nasko Oskov. "Site isolation: Process separation for web sites within the browser". In: *USENIX Security*. 2019.

[65] Stephan van Schaik et al. "Addendum 1 to RIDL: Rogue In-flight Data Load". In: *S&P*. Nov. 2019.

[66] Stephan van Schaik et al. "Addendum 2 to RIDL: Rogue In-flight Data Load". In: *S&P*. Jan. 2020.

[67] Stephan van Schaik et al. "RIDL: Rogue In-flight Data Load". In: *S&P*. May 2019.

[68] Michael Schwarz et al. "Netspectre: Read arbitrary memory over network". In: *European Symposium on Research in Computer Security*. Springer. 2019, pp. 279–299.

[69] Michael Schwarz et al. "ZombieLoad: Cross-Privilege-Boundary Data Sampling". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 753–768. ISBN: 9781450367479. DOI: `10.1145/3319535.3354252`. URL: `https://doi.org/10.1145/3319535.3354252`.

[70] Julian Stecklina and Thomas Prescher. "LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels". In: (2018).

[71] Jakub Szefer et al. "Eliminating the hypervisor attack surface for a more secure cloud". In: *Proceedings of the 18th ACM conference on Computer and communications security*. 2011, pp. 401–412.

[72] Jo Van Bulck et al. "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection". In: *41th IEEE Symposium on Security and Privacy (S&P'20)*. 2020.

[73] Zhenghong Wang and Ruby B Lee. "Covert and side channels due to processor architecture". In: *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE. 2006, pp. 473–482.

[74] Ofir Weisse et al. "Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution". In: *Technical report* (2018).

[75] Zhenyu Wu, Zhang Xu, and Haining Wang. "Whispers in the hyper-space: high-bandwidth and reliable covert channel attacks inside the cloud". In: *IEEE/ACM Transactions on Networking* 23.2 (2014), pp. 603–615.

[76] Hongyan Xia et al. "A Secret-Free Hypervisor: Rethinking Isolation in the Age of Speculative Vulnerabilities". In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society. 2022, pp. 1544–1544.

[77] Wenjie Xiong and Jakub Szefer. “Survey of transient execution attacks and their mitigations”. In: *ACM Computing Surveys (CSUR)* 54.3 (2021), pp. 1–36.

[78] Cong Xu et al. “vTurbo: Accelerating Virtual Machine I/O Processing Using Designated Turbo-Sliced Core”. In: *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 2013, pp. 243–254.

[79] Yuval Yarom and Katrina Falkner. “FLUSH + RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack.” In: *USENIX Security'14*.