The Design and Implementation of Hyperupcalls

Nadav Amit
VMware Research

Michael Wei
VMware Research

Abstract

The virtual machine abstraction provides a wide variety of benefits which have undeniably enabled cloud computing. Virtual machines, however, are a double-edged sword as hypervisors they run on top of must treat them as a black box, limiting the information which the hypervisor and virtual machine may exchange, a problem known as the *semantic gap*. In this paper, we present the design and implementation of a new mechanism, hyperupcalls, which enables a hypervisor to safely execute verified code provided by a guest virtual machine in order to transfer information. Hyperupcalls are written in C and have complete access to guest data structures such as page tables. We provide a complete framework which makes it easy to access familiar kernel functions from within a hyperupcall. Compared to state-of-the-art paravirtualization techniques and virtual machine introspection, Hyperupcalls are much more flexible and less intrusive. We demonstrate that hyperupcalls can not only be used to improve guest performance for certain operations by up to 2× but hyperupcalls can also serve as a powerful debugging and security tool.

1 Introduction

Hardware virtualization introduced the abstraction of a virtual machine (VM), enabling hosts known as *hypervisors* to run multiple operating systems (OSs) known as *guests* simultaneously, each under the illusion that they are running in their own physical machine. This is achieved by exposing a hardware interface which mimics that of true, physical hardware. The introduction of this simple abstraction has led to the rise of the modern data center and the cloud as we know it today. Unfortunately, virtualization is not without drawbacks. Although the goal of virtualization is for VMs and hypervisors to be oblivious from each other, this separation renders both sides unable to understand decisions made on the other side, a problem known as the *semantic gap*.

Requestor	Paravirtual, Executed by:		Uncoordinated
	Hypervisor	Guest	Introspection
Guest	Hypercalls	Pre-Virt [42]	HVI [72]
HV	**Hyperupcalls**	Upcalls	VMI [25]

Table 1: *Hypervisor-Guest Communication Mechanisms.* Hypervisors (HV) and guests may communicate through a variety of mechanisms, which are characterized by who initiates the communication, who executes and whether the channel for communication is coordinated (paravirtual). Shaded cells represent channels which require context switches.

Addressing the semantic gap is critical for performance. Without information about decisions made in guests, hypervisors may suboptimally allocate resources. For example, the hypervisor cannot know what memory is free in guests without understanding their internal OS state, breaking the VM abstraction. State-of-the-art hypervisors today typically bridge the semantic gap with *paravirtualization* [11, 58], which makes the guest aware of the hypervisor. Paravirtualization alleviates the guest from the limitations of the physical hardware interface and allows direct information exchange with the hypervisor, improving overall performance by enabling the hypervisor to make better resource allocation decisions.

Paravirtualization, however, involves the execution of code both in the context of the hypervisor and the guest. *Hypercalls* require that the guest make a request to be executed in the hypervisor, much like a system call, and *upcalls* require that the hypervisor make a request to be executed in the guest. This design introduces a number of drawbacks. First, paravirtual mechanisms introduce context switches between hypervisors and guests, which may be substantial if frequent interactions between guests and the hypervisor are needed [7]. Second, the requestor of a paravirtual mechanism must wait for it to be serviced in another context which may be busy, or waking the guest if it is idle. Finally, par-

avirtual mechanisms couple the design of the hypervisor and guest: paravirtual mechanisms need to be implemented for each guest and hypervisor, increasing complexity [46] and hampering maintainability [77]. Adding paravirtual features requires updating both the guest and hypervisor with a new interface [69] and has the potential to introduce bugs and the attack surface [47, 75].

A different class of techniques, VM introspection (VMI) [25] and the reverse, hypervisor introspection (HVI) [72] aim to address some of the shortcomings of paravirtualization by *introspecting* the other context, enabling communication transfer without context switching or prior coordination. These techniques however, are fragile: small changes in data structures, behavior or even security hardening [31] can break introspective mechanisms, or worse, introduce security vulnerabilities. As a result, introspection is usually relegated to the area of intrusion detection systems (IDSs) which detect malware or misbehaving applications.

In this paper, we describe the design and implementation of hyperupcalls [1], a technique which enables a hypervisor to communicate with a guest, like an upcall, but without a context switch, like VMI. This is achieved through the use of verified code, which enables a guest to communicate to the hypervisor in a flexible manner while ensuring that the guest cannot provide misbehaving or malicious code. Once a guest registers a hyperupcall, the hypervisor can execute it to perform actions such as locating free guest pages or running guest interrupt handlers without switching into the guest.

Hyperupcalls are easy to build: they are written in a high level language such as C, and we provide a framework which allows hyperupcalls to share the same codebase and build system as the Linux kernel that may be generalized to other operating systems. When the kernel is compiled, a toolchain translates the hyperupcall into verifiable bytecode. This enables hyperupcalls to be easily maintained. Upon boot, the guest registers the hyperupcalls with the hypervisor, which verifies the bytecode and compiles it back into native code for performance. Once recompiled, the hypervisor may invoke the hyperupcall at any time.

We show that using a hyperupcalls can significantly improve performance by allowing a hypervisor to be proactive about resource allocation, rather than waiting for guests to react through existing mechanisms. We build hyperupcalls for memory reclamation and dealing with interprocessor interrupts (IPIs) and show a performance improvement of up to 2×. In addition to improving performance, hyperupcalls can also enhance both the security and debuggability of systems in virtual environments. We develop a hyperupcall to enables guests to write-protect memory pages without the use of specialized hardware, and another which enables `ftrace` [57] to capture both guest and hypervisor events in a unified trace, allowing us to gain new insights on performance in virtualized environments.

This paper makes the following contributions:

- We build a taxonomy of mechanisms for bridging the semantic gap between hypervisor and guests and place hyperupcalls within that taxonomy (§2).
- We describe and implement hyperupcalls (§3) with:
  - An environment for writing hyperupcalls and a framework for using guest code (§3.1)
  - A compiler (§3.2) and verifier (§3.4) for hyperupcalls which addresses the complexities and limitations of verified code.
  - Registration (§3.3) and execution (§3.5) mechanisms for hyperupcalls.
- We prototype and evaluate hyperupcalls and show that hyperupcalls can improve performance (§4.3, §4.2), security (§4.5) and debuggability (§4.4).

2 Communication Mechanisms

It is now widely accepted that in order to extract the most performance and utility from virtualization, hypervisors and their guests need to be aware of one another. To that end, a number of mechanisms exist to facilitate communication between hypervisors and guests. Table 1 summarizes these mechanisms, which can be broadly characterized by the requestor, the executor, and whether the mechanism requires that the hypervisor and the guest coordinate ahead of time.

In the next section, we discuss these mechanisms and describe how hyperupcalls fulfill a need for a communication mechanism where the hypervisor makes and executes its own requests without context switching. We begin by introducing state-of-the-art paravirtual mechanisms in use today.

2.1 Paravirtualization

Hypercalls and upcalls. Most hypervisors today leverage paravirtualization to communicate across the semantic gap. Two mechanisms in widespread use today are *hypercalls*, which allow guests to invoke services provided by the hypervisor, and *upcalls*, which enable the hypervisor to make requests to guests. Paravirtualization means that the interface for these mechanisms are coordinated ahead of time between hypervisor and guest [11].

One of the main drawbacks of upcalls and hypercalls is that they require a context switch as both mechanisms are executed on the opposite side of the request. As a result, these mechanisms must be invoked with care. Invoking

[1] Hyperupcalls were previously published as "hypercallbacks" [5].

a hypercall or upcall too frequently can result in high latencies and computing resource waste [3].

Another drawback of upcalls in particular that the requests are handled by the guest, which could be busy handling other tasks. If the guest is busy or if a guest is idle, upcalls incur the additional penalty of waiting for the guest to be free or for the guest or woken up. This can take an unbounded amount of time, and hypervisors may have to rely on a penalty system to ensure guests respond in a reasonable amount of time.

Finally, by increasing the coupling between the hypervisor and its guests, paravirtual mechanisms can be difficult to maintain over time. Each hypervisor have their own paravirtual interfaces, and each guest must implement the interface of each hypervisor. The paravirtual interface is not thin: Microsoft's paravirtual interface specification is almost 300 pages long [46]. Linux provides a variety of paravirtual hooks, which hypervisors can use to communicate with the VM [78]. Despite the effort to standardize the paravirtualization interfaces they are incompatible with each other, and evolve over time, adding features or even removing some (e.g., Microsoft hypervisor event tracing). As a result, most hypervisors do not fully support efforts to standardize interfaces and specialized OSs look for alternative solutions [45, 54].

Pre-virtualization. Pre-virtualization [42] is another mechanism in which the guest requests services from the hypervisor, but the requests are served in the context of the guest itself. This is achieved by code injection: the guest leaves stubs, which the hypervisor fills with hypervisor code. Pre-virtualization offers an improvement over hypercalls, as they provide more flexible interface between the guest and the hypervisor. Arguably, pre-virtualization suffers from a fundamental limitation: code that runs in the guest is deprivileged and cannot perform sensitive operations, for example, accessing shared I/O devices. As a result, in pre-virtualization, the hypervisor code that runs in the guest still needs to communicate with the privileged hypervisor code using hypercalls.

2.2 Introspection

Introspection occurs when a hypervisor or guest attempts to infer information from the other context without directly communicating with it. With introspection, no interface or coordination is required. For instance, a hypervisor may attempt to infer the state of completely unknown guests simply by their memory access patterns. Another difference between introspection and paravirtualization is that no context switch occurs: all the code to perform introspection is executed in the requestor.

Virtual machine introspection (VMI). When a hypervisor introspects a guest, it is known as VMI [25]. VMI was first introduced to enhance VM security by providing intrusion detection (IDS) and kernel integrity checks from a privileged host [10, 24, 25]. VMI has also been applied to checkpointing and deduplicating VM state [1], as well as monitoring and enforcing hypervisor policies [55]. These mechanisms range from simply observing a VM's memory and I/O access patterns [36] to accessing VM OS data structures [16], and at the extreme end they may modify VM state and even directly inject processes into it [26, 19]. The primary benefits of VMI are that the hypervisor can directly invoke VMI without a context switch, and the guest does not need to be "aware" that it is inspected for VMI to function. However, VMI is fragile: an innocuous change in the VM OS, such as a hotfix which adds an additional field to a data structure could render VMI non-functional [8]. As a result, VMI tends to be a "best effort" mechanism.

HVI. Used to a lesser extent, a guest may introspect the hypervisor it is running on, known as hypervisor introspection (HVI) [72, 61]. HVI is typically employed either to secure a VM from untrusted hypervisors [62] or by malware to circumvent hypervisor security [59, 48].

2.3 Extensible OSes

While hypervisors provide a fixed interface, OS research suggested along the years that flexible OS interfaces can improve performance without sacrificing security. The Exokernel provided low level primitives, and allowed applications to implement high-level abstractions, for example for memory management [22]. SPIN allowed to extend kernel functionality to provide application-specific services, such as specialized interprocess communication [13]. The key feature that enables these extensions to perform well without compromising security, is the use of a simple byte-code to express application needs, and running this code at the same protection ring as the kernel. Our work is inspired by these studies, and we aim to design a flexible interface between the hypervisor and guests to bridge the semantic gap.

2.4 Hyperupcalls

This paper introduces hyperupcalls, which fulfill a need for a mechanism for the hypervisor to communicate to the guest which is coordinated (unlike VMI), executed by the hypervisor itself (unlike upcalls) and does not require context switches (unlike hypercalls). With hyperupcalls, the VM coordinates with the hypervisor by registering verifiable code. This code is then executed by the hypervisor in response to events (such as memory pressure, or

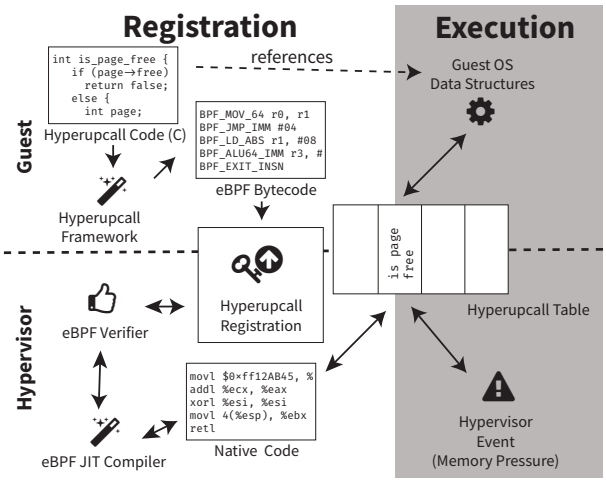


Figure 1: *System Architecture.* Hyperupcall registration (left) consists of compiling C code, which may reference guest data structures, into verifiable bytecode. The guest registers the generated bytecode with the hypervisor, which verifies its safety, compiles it into native code and sets it in the VM hyperupcall table. When the hypervisor encounters an event (right), such as a memory pressure, it executes the respective hyperupcall, which can access and update data structures of the guest.

VM entry/exit). In a way, hyperupcalls can be thought of as upcalls executed by the hypervisor.

In contrast to VMI, the code to access VM state is provided by the guest so the hyperupcalls are fully aware of guest internal data structures— in fact, hyperupcalls are built with the guest OS codebase and share the same code, thereby simplifying maintenance while providing the OS with an expressive mechanism to describe its state to underlying hypervisors.

Compared to upcalls, where the hypervisor makes asynchronous requests to the guest, the hypervisor can execute a hyperupcall at any time, even when the guest is not running. With an upcall, the hypervisor is at the mercy of the guest, which may delay the upcall [6]. Furthermore, because upcalls operate like remote requests, upcalls may be forced to implement OS functionality in a different manner. For example, when flushing remote pages in memory ballooning [71], the canonical technique for identifying free guest memory, the guest increases memory pressure using a dummy process to free pages. With a hyperupcall, the hypervisor can act as if it were a guest kernel thread and scan the guest for free pages directly.

Hyperupcalls resemble pre-virtualization, in that code is transferred across the semantic gap. Transferring code not only allows for more expressive communication, but it also moves the execution of the request to the other side of the gap, enhancing performance and functionality. Unlike pre-virtualization, the hypervisor cannot trust the code being provided by the virtual machine, and the hypervisor must ensure that execution environment for the hyperupcall is consistent across invocations.

	Local Hyperupcalls	**Global Hyperupcalls**
event examples	VM-exit VM-entry interrupt injection page mapping	memory reclaim memory aging VCPU preemption
use	notifications and local policy decisions	global policy decisions
preemptable	yes	no
memory mappings	host user-space	host kernel-space
memory limit	high	low
memory pinning	no	yes
callback chaining	yes	no
hyperupcall examples	IPI handling security agent tracing	scheduler activation memory discard hints

Table 2: *Hyperupcall event types.* The hypervisor enforces certain limitations on global hyperupcalls, which are used to make policy decisions.

3 Architecture

Hyperupcalls are short verifiable programs provided by guests to the hypervisor to improve performance or provide additional functionality. Guests provide hyperupcalls to the hypervisor through a *registration* process at boot, allowing the hypervisor to access the guest OS state and provide services by *executing* them after verification. The hypervisor runs hyperupcalls in response to events or when it needs to query guest state. The architecture of hyperupcalls and the system we have built for utilizing them is depicted in Figure 1.

We aim to make hyperupcalls as simple as possible to build. To that end, we provide a complete *framework* which allows a programmer to write hyperupcalls using the guest OS codebase. This greatly simplifies the development and maintenance of hyperupcalls. The framework compiles this code into verifiable code which the guest registers with the hypervisor. In the next section, we describe how an OS developer writes a hyperupcall using our framework.

3.1 Building Hyperupcalls

Guest OS developers write hyperupcalls for each hypervisor event they wish to handle. Hypervisors and guests agree on these events, for example VM entry/exit, page mapping or virtual CPU (VCPU) preemption. Each hyperupcall is identified by a predefined identifier, much like the UNIX system call interface [56]. Table 2 gives examples of events a hyperupcall may handle.

3.1.1 Providing Safe Code

One of the key properties of hyperupcalls is that the code must be guaranteed to not compromise the hypervisor. In order for a hyperupcall to be safe, it must only be able to access a restricted memory region dictated by the hypervisor, run for a limited period of time without blocking, sleeping or taking locks, and only use hypervisor services that are explicitly permitted.

Since the guest is untrusted, hypervisors must rely on a security mechanism which guarantees these safety properties. There are many solutions that we could have chosen: software fault isolation (SFI) [70], proof-carrying code [51] or safe languages such as Rust. To implement hyperupcalls, we chose the enhanced Berkeley Packet Filter (eBPF) VM.

We chose eBPF for several reasons. First, eBPF is relatively mature: BPF was introduced over 20 years ago and is used extensively throughout the Linux kernel, originally for packet filtering but extended to support additional use cases such as sandboxing system calls (`seccomp`) and tracing of kernel events [34]. eBPF enjoys wide adoption and is supported by various runtimes [14, 49]. Second, eBPF can be provably verified to have the safety properties we require, and Linux ships with a verifier and JIT which verifies and efficiently executes eBPF code [74]. Finally, eBPF has a LLVM compiler backend, which enables eBPF bytecode to be generated from a high level language using a compiler frontend (Clang). Since OSes are typically written in C, the eBPF LLVM backend provides us with a straightforward mechanism to convert unsafe guest OS source code into verifiably safe eBPF bytecode.

3.1.2 From C to eBPF — the *Framework*

Unfortunately, writing a hyperupcall is not as simple recompiling OS code into eBPF bytecode. However, our framework aims to make the process of writing a hyperupcalls simple and maintainable as possible. The framework provides three key features that simplify the writing of hyperupcalls. First, the framework takes care of dealing with guest address translation issues so guest OS symbols are available to the hyperupcall. Second, the framework addresses limitations of eBPF, which places significant constraints on C code. Finally, the framework defines a simple interface which provides the hyperupcall with data so it can execute efficiently and safely.

Guest OS symbols and memory. Even though hyperupcalls have access to the entire physical memory of the guest, accessing guest OS data structures requires knowing where they reside. OSes commonly use kernel address space layout randomization (KASLR) to randomize the virtual offsets for OS symbols, rendering them unknown during compilation time. Our framework enables OS symbol offsets to be resolved at runtime by associating pointers using address space attributes and injecting code to adjust the pointers. When a hyperupcall is registered, the guest provides the actual symbol offsets enabling a hyperupcall developer to reference OS symbols (variables and data structures) in C code as if they were accessed by a kernel thread.

Global / Local Hyperupcalls. Not all hyperupcalls need to be executed in a timely manner. For example, notifications informing the guest of hypervisor events such as a VM-entry/exit or interrupt injection only affect the guest and not the hypervisor. We refer to hyperupcalls that only affect the guest that registered it as local, and hyperupcalls that affect the hypervisor as a whole as global. If a hyperupcall is registered as local, we relax the timing requirement and allow the hyperupcall to block and sleep. Local hyperupcalls are accounted in the VCPU time of the guest similar to a trap, so a misbehaving hyperupcall penalizes itself.

Global hyperupcalls, however, must complete their execution in a timely manner. We ensure that for the guest OS pages requested by global hyperupcalls are pinned during the hyperupcall, and restrict the memory that can be accessed to 2% (configurable) of the guest's total physical memory. Since local hyperupcalls may block, the memory they use does not need to be pinned, allowing local hyperupcalls to address all of guest memory.

Addressing eBPF limitations. While eBPF is expressive, the safety guarantees of eBPF bytecode mean that it is not Turing-complete and limited, so only a subset of C code can be compiled into eBPF. The major limitations of eBPF are that it does not support loops, the ISA does not contain atomics, cannot use self-modifying code, function pointers, static variables, native assembly code, and cannot be too long and complex to be verified.

One of the consequences of these limitations is that hyperupcall developers must be aware of the code complexity of the hyperupcall, as complex code will fail the verifier. While this may appear to be an unintuitive restriction, other Linux developers using BPF face the same restriction, and we provide a helper functions in our framework to reduce complexity, such as `memset` and `memcpy`, as well as functions that perform native atomic

Helper Name	Function
send_vcpu_ipi	Send an interrupt to VCPU
get_vcpu_register	Read a VCPU register
set_vcpu_register	Read a VCPU register
memcpy	memcpy helper function
memset	memset helper function
cmpxchg	compare-and-swap
flush_tlb_vcpu	Flush VCPU's TLB
get_exit_info	Get info on an VM_EXIT event

Table 3: *Selected hyperupcall helper functions.* The hyperupcall may call these functions implemented in the hypervisor, as they cannot be verified using eBPF.

operations such as cmpxchg. A selection of these helper functions is shown in Table 3. In addition, our framework masks memory accesses (§3.4), which greatly reduces the complexity of verification. In practice, as long as we were careful to unroll loops, we did not encounter verifier issues while developing the use cases in (§4) using a setting of 4096 instructions and a stack depth of 1024.

Hyperupcall interface. When a hypervisor invokes a hyperupcall, it populates a context data structure, shown in Table 4. The hyperupcall receives an event data structure which indicates the reason the callback was called, and a pointer to the guest (in the address space of the hypervisor, which is executing the hyperupcall). When the hyperupcall completes, it may return a value, which can be used by the hypervisor.

Writing the hyperupcall. With our framework, OS developers write C code which can access OS variables and data structures, assisted by the helper functions of the framework. A typical hyperupcall will read the event field, read or update OS data structures and potentially return data to the hypervisor. Since the hyperupcall is part of the OS, the developers can reference the same data structures used by the OS itself—for example, through header files. This greatly increases the maintainability of hyperupcalls, since data layout changes are synchronized between the OS source and the hyperupcall source.

It is important to note that a hyperupcall cannot invoke guest OS functions directly, since that code has not been secured by the framework. However, OS functions can be compiled into hyperupcalls and be integrated in the verified code.

3.2 Compilation

Once the hyperupcall has been written, it needs to be compiled into eBPF bytecode before the guest can register it with the hypervisor. Our framework generates this bytecode as part of the guest OS build process by running the hyperupcall C code through Clang and the eBPF LLVM backend, with some modifications to assist with address translation and verification:

Input field	Function
event	Event specific data including event ID.
hva	Host virtual address (HVA) in which the guest memory is mapped.
guest_mask	Guest address mask to mask bits which are higher than the guest memory address-width. Used for verification (§ 3.4).
vcpus	Pointers to the hypervisor VCPU data structure, if the event is associated with a certain VCPU, or a pointer to the guest OS data structure. Inaccessible to the hyperupcall, but used by helper functions.
vcpu_reg	Frequently accessed VCPU registers: instruction pointer and VCPU ID.
env	Environment variables, provided by the guest during hyperupcallregistration. Used to set address randomization offsets.

Table 4: *Hyperupcall context data*. These fields are populated by the hypervisor when a hyperupcall is called.

Guest memory access. To access guest memory, we use eBPF's direct packet access (DPA) feature, which was designed to allow programs to access network packets safely and efficiently without the use of helper functions. Instead of passing network packets, we utilize this feature by treating the guest as a "packet". Using DPA in this manner required a bug fix [2] to the eBPF LLVM backend, as it was written with the assumption that packet sizes are $\leq$64KB.

Address translations. Hyperupcalls allow the hypervisor to seamlessly use guest virtual addresses (GVAs), which makes it appear as if the hyperupcall was running in the guest. However, the code is actually executed by the hypervisor, where host virtual address (HVAs) are used, rendering guest pointers invalid. To allow the use of guest pointers transparently in the host context, these pointers therefore need to be translated from GVAs into HVAs. We use the compiler to make these translations.

To make this translation simple, the hypervisor maps the GVA range contiguously in the HVA space, so address translations can easily be done by adjusting the base address. As the guest might need the hyperupcall to access multiple contiguous GVA ranges—for example, one for the guest 1:1 direct mapping and of the OS text section [37]—our framework annotates each pointer with its respective "address space" attribute. We extend the LLVM compiler to use this information to inject eBPF code that converts each of the pointer from GVA to HVA by a simple subtraction operation. It should be noted that the generated code safety is not assumed by the hypervisor and is verified when the hyperupcall is registered.

Bound Checks. The verifier rejects code with direct memory accesses unless it can ensure the memory accesses are within the "packet" (in our case, guest memory) bounds. We cannot expect the hyperupcall programmer to perform the required checks, as the burden of adding them is substantial. We therefore enhance the compiler to automatically add code that performs bound checks prior to each memory access, allowing verification to pass. As we note in Section 3.4, the bounds checking is done using masking and not branches to ease verification.

Context caching. Our compiler extension introduces intrinsics to get a pointer to the context or to read its data. The context is frequently needed along the callback for calling helper functions and for translating GVAs. Delivering the context as a function parameter requires intrusive changes and can prevent sharing code between the guest and its hyperupcall. Instead, we use the compiler to cache the context pointer in one of the registers and retrieve it when needed.

3.3 Registration

After a hyperupcall is compiled into eBPF bytecode, it is ready to be registered. Guests can register hyperupcalls at any time, but most hyperupcalls are registered when the guest boots. The guest provides the hyperupcall event ID, hyperupcall bytecode and the virtual memory the hyperupcall will use. Each parameter is described below:

Hyperupcall event ID. ID of the event to handle.

Memory registration. The guest registers the virtual contiguous memory regions used by the hyperupcall. For global hyperupcalls, this memory is restricted to a maximum of 2% of the guest's total physical memory (configurable and enforced by the hypervisor).

Hyperupcall bytecode. The guest provides a pointer to the hyperupcall bytecode with its size.

3.4 Verification

The hypervisor verifies that each hyperupcall is safe to execute at registration time. Our verifier is based on the Linux eBPF verifier and checks three properties of the hyperupcall: memory accesses, number of runtime instructions, and helper functions used.

Ideally, verification is *sound*, ensuring only safe code passes verification, and *complete*, successfully verifying any safe program. While soundness cannot be compromised as it might jeopardize the system safety, many verification systems, including eBPF, sacrifice completeness to keep the verifier simple. In practice, the verifier requires programs to be written in a certain way to pass verification [66], and even then verification can fail due to path explosion. These limitations are at odds of our goal of making hyperupcalls simple to build.

We discuss the properties our verifier checks below, and how we simplify these checks to make verification as straightforward as possible.

Bounded runtime instructions. For global hyperupcalls, the eBPF verifier ensures that any possible execution of the hyperupcall contains a limited number of instructions, which is set by the hypervisor (defaulted to 4096). This ensures that the hypervisor can execute the hyperupcall in a timely manner, and that there are no infinite loops which can cause the hyperupcall not to exit.

Memory access verification. The verifier ensures that memory accesses only occur in the region bounded by the "packet", which in a hyperupcall is the virtual memory region provided during registration. As noted before, we enhance the compiler to automatically add code that proves to the verifier that each memory access is safe.

However, adding such code naively results in frequent verification failures. The current Linux eBPF verifier is quite limited in its ability to verify the safety of memory accesses, as it requires that they will be preceded by compare and branch instructions to prevent out of bound accesses. The verifier explores the possible execution paths and ensures their safety. Although the verifier employs various optimizations to prune branches and avoid walking every possible branch, verification often exhausts available resources and fails as we and others have experienced [65].

Therefore, instead of using compare and branch to ensure memory access safety, our enhanced compiler adds code that masks memory accesses offset within each range, preventing out-of-bounds memory accesses. We enhance the verifier to recognize this masking as safe. After applying this enhancement, all the programs we wrote passed verification.

Helper function safety. Hyperupcalls may call helper functions to both improve performance and to help limit the number of runtime instructions. Helper functions are a standard eBPF feature and the verifier enforces the helper functions which can be called, which may vary from event to event depending on hypervisor policy. For example, the hypervisor may disallow the use of `flush_tlb_vcpu` during memory reclamation, as it may block the system for an extended amount of time.

The verifier checks to ensure that the inputs to the helper function are safe, ensuring that the helper function only accesses memory which it is permitted to access. While these checks could be done in the helper function, new eBPF extensions allow the verifier to statically verify the helper function inputs. Furthermore, the hypervisor can also set a policy for inputs on a per-event basis (e.g, `memcpy` size for global hyperupcalls).

The number and complexity of helper functions should be limited as well, as they become part of the trusted computing base. We therefore only introduce simple helper functions, which mostly rely on code that the guest can already trigger today directly or indirectly, for example interrupt injection.

eBPF security. Two of the proof-of-concept exploits of the recently discovered "Spectre" hardware vulnerabilities [38, 30] targeted eBPF, which might raise concerns about eBPF and hyperupcall safety. While exploiting these vulnerabilities is simpler if an attacker can run unprivileged code in privileged context, just as hyperupcalls do, discovered attacks can be prevented [63]. In fact, these security vulnerabilities can make hyperupcalls more compelling as their mitigation techniques (e.g, return stack buffer stuffing [33]) induce extra overheads when context switches take place using traditional paravirtual mechanisms such as upcalls and hypercalls.

3.5 Execution

Verified hyperupcalls are installed into a per guest hyperupcall table. Once the hyperupcall has been registered and verified, the hypervisor executes hyperupcalls in response to events.

Hyperupcall patching. To avoid the overhead of testing whether hyperupcall is registered, the hypervisor uses a code patching technique, known in Linux as "static keys" [12]: a no-op instruction is set on each of the hypervisor hyperupcall invocation code only when hyperupcalls are registered.

Accessing remote VCPU state. Some hyperupcalls read or modify the state of remote VCPUs. These VCPUs may not be running or their state may be accessed by a different thread of the hypervisor. Even if the remote VCPU is preempted, the hypervisor may have already read some registers and not expect them to change until the VCPU resumes execution. If the hyperupcall writes to remote VCPU registers, it may break the hypervisor invariants and even introduce security issues.

Furthermore, reading remote VCPU registers can induce high overheads, as part of the VCPU state may be cached in another CPU, and must be written back to memory first if the VCPU state is to be read. More importantly, in Intel CPUs the VCPU state cannot be accessed by common instructions, and the VCPU must be "loaded" first before its state can be accessed by using special instructions (`VMREAD` and `VMWRITE`). Switching the loaded VCPU incurs significant overhead, which roughly 1800 cycles on our system.

For performance, we define synchronization points where the hypervisor is commonly preempted, and accessing the VCPU state is known to be safe. At these points we "decache" VCPU registers from the VMCS and write them to the memory so the hyperupcall can read them. The hyperupcall writes to remote VCPU registers and updates the decached value to flag the hypervisor to reload the register values into the VMCS before resuming that VCPU. Hyperupcalls that access remote VCPUs are executed on a best-effort basis, running only if the VCPU is in a synchronization point. The remote VCPU is prevented from resuming execution while the hyperupcall is running.

Using guest OS locks. Some of the OS data-structures are protected by locks. Hyperupcalls that require consistent guest OS data structure view should abide the synchronization scheme that the guest OS dictates. Hyperupcall, however, can only acquire locks opportunistically, since a VCPU might be preempted while holding a lock. The lock implementation might need to be adapted to support locking by an external entity, different than any VCPU. Releasing a lock can require relatively large code to handle slow-paths, which might prevent the timely verification of the hyperupcall.

While various ad-hoc solutions may be proposed, it seems a complete solution requires the guest OS locks to be hyperupcall-aware. It also necessitates support for calling eBPF function from eBPF code to avoid inflated code size that might cause verification failures. Since this support has been added very recently, our implementation does not include lock support.

4 Use Cases and Evaluation

Our evaluation is guided by the following questions:

- What are the overheads of using verified code (eBPF) versus native code? (§4.1).
- How do hyperupcalls compare to other paravirtual mechanisms (§4.3, 4.2, 4.5)?
- How can hyperupcalls enhance not only the performance (§4.3, 4.2) but also the security (§4.5) and debuggability (§4.4) of virtualized environments?

Testbed. Our testbed consists of a 48 core dual-socket Dell PowerEdge R630 server with Intel E5-2670 CPUs, a Seagate ST1200 disk, which runs Ubuntu 17.04 with Linux kernel v4.8. The benchmarks are run on guests with 16 VCPUs and 8GB of RAM. Each measurement was performed 5 times and the average result is reported.

Hyperupcall prototype. We implemented a prototype for hyperupcall support on Linux v4.8 and KVM, the hypervisor which is integrated in Linux. Hyperupcalls are compiled through a patched LLVM 4, and are verified through the Linux kernel eBPF verifier with the patches we described in §3. We enable the Linux eBPF "JIT" engine , which compiles the eBPF code to native machine code after verification. The correctness of the BPF JIT

use case	h-visor event	runtime (cycles) h-upcall	native	eBPF instr.	C SLoC
discard § 4.3	reclaim	185	147	357	32
tracing § 4.4	exit	568	336	3308	889
TLB § 4.2	interrupt	395	530	111	112
protect § 4.5	exit	43	25	119	74
	map	108	92	170	52

Table 5: Evaluated hyperupcall use cases, comparison of runtime, eBPF instructions and number of lines of code.

engine has been studied and can be verified [74].

Use cases. We evaluate four hyperupcall use cases as listed in Table 5. Each use case demonstrates the use of hyperupcalls on different hypervisor events, and uses hyperupcalls of varying complexity.

4.1 Hyperupcall overheads

We evaluate the overheads of using verified code to service hypervisor requests by comparing the runtime of a hyperupcall versus native code with the same function (Table 5). Overall, we find that the absolute overhead of the verified code relative to native is small ($<$ 250 cycles). For the TLB use case which handles TLB shootdown to inactive cores, our hyperupcall runs faster than native code since the TLB flush is deferred. The overhead of verifying a hyperupcall is minimal. For the longest hyperupcall (tracing), verification took 67ms.

4.2 TLB Shootdown

While interrupt delivery to VCPUs can usually be done efficiently, there is a significant penalty if the target VCPU is not running. This can occur if CPUs are overcommitted and scheduling the target VCPU requires preempting another VCPU. With synchronous interprocessor interrupts (IPIs), the sender resumes execution only after the receiver indicates the IPI was delivered and handled, resulting in prohibitive overheads.

The overhead of IPI delivery is most notable in the case of translation lookaside buffer (TLB) shootdowns, a software protocol that OSs use to keep TLBs—caches of virtual to physical address mapping—coherent. As common CPU architectures (e.g., x86) do not keep TLBs coherent in hardware, an OS thread that modifies a mapping sends an IPI to other CPUs that may cache the mapping, and these CPUs then flush their TLBs.

We use hyperupcalls to handle this scenario by registering a hyperupcall which handles TLB shootdowns when interrupts are delivered to a VCPU. The hypervisor provides that hyperupcall with the interrupt vector and the target VCPU after ensuring it is in quiescent state. Our hyperupcall checks whether this vector is the "remote function invocation" vector and whether the function pointer equals to the OS TLB flush function. If it does, it runs this function with few minor modifications: (1) instead of flushing the TLB using native instruction, the TLB flush is performed using a helper function, which defers it to the next VCPU re-entry; (2) TLB flush is performed even when the VCPU interrupts are disabled, as experimentally it improves performance.

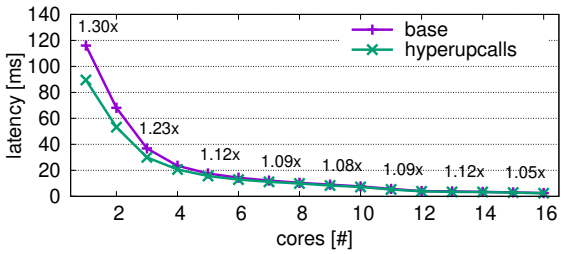


Figure 2: The latency of Apache when CPUs are overcommitted, with and without a hyperupcall that handle interrupts to preempted VCPUs. Numbers above data points indicate speedup over base.

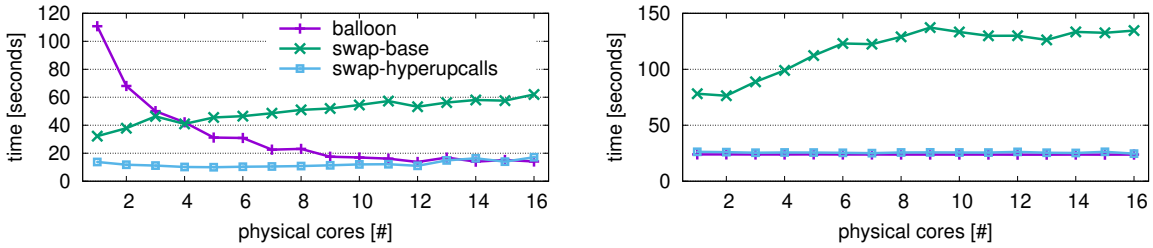
Admittedly, an alternative solution is available: introducing a hypercall that delegates TLB flushes to the hypervisor [52]. Although this solution can prevent TLB flushes, it requires a different code path, which may introduce hidden bugs [43], complicate the integration with OS code or introduce additional overheads [44]. This solution is also limited to TLB flushes, and cannot deal with other interrupts, for example, rescheduling IPIs.

Evaluation We run Apache Web server [23] in a guest using the default `mpm_event` module, which runs multithreaded workers to handle incoming requests. To measure performance, we use ApacheBench, an Apache HTTP server benchmarking tool, generating 10k requests using 16 connections, and measuring the request latency. The results, which are shown in Figure 2, show hyperupcalls reduce the latency by up to 1.3×. It might appear surprising that performance improves even when the physical CPUs are not oversubscribed. However, as VCPUs are often momentarily idle in this benchmark, they can also trigger an exit to the hypervisor.

4.3 Discarding Free Memory

Free memory, by definition, holds no needed data and can be discarded. If the hypervisor knows what memory is free in the guest, it can discard it during memory reclamation, snapshotting, live migration or lock-step execution [20] and avoid I/O operations for saving and restoring their content. Information on which memory pages are free, however, is held by the guest and unavailable to the hypervisor due to the semantic gap.

Throughout the years several mechanisms have been proposed to inform the hypervisor which memory pages



(a) reclaim (b) refault

Figure 3: Time of guest memory reclaim of 7GB and refault, when reading a 4GB file, when CPUs are overcommitted. The x-axis shows the number of physical cores available. (a) As the number of physical cores decrease (and overcommitment increases), the time to reclaim memory increases. (b) refaulting free memory incurs a significant penalty for uncooperative swapping (swap-base) on its own because it swaps out active and free pages.

are free using paravirtualization. These solutions, however, either couple the guest and hypervisor [60]; induce overheads due to frequent hypercalls [41] or are limited to live migration [73]. All of these mechanisms suffer for an inherent limitation: without coupling the guest and the hypervisor, the guest needs to communicate to the hypervisor which pages are free.

In contrast, a hypervisor that supports hyperupcalls does not need to be notified about free pages. Instead, the guest sets a hyperupcall that describes whether a page is discardable based on the page metadata (Linux's `struct page`) and is based in Linux on the `is_free_buddy_page` function. When the hypervisor performs an operation that can benefit from discarding a free guest memory page such as reclaiming a page, the hypervisor invokes this hyperupcall to check whether the page is discardable. The hyperupcall is also called when the page is already unmapped, preventing a race in which it is discarded when it is no longer free.

Checking whether a page can be discarded must be done through a global hyperupcall, since the answer must be provided in a bounded and short time. As a result, the guest can only register part of its memory to be used by the hyperupcall, since this memory is never paged out to ensure timely execution of the hyperupcall. Our Linux guest registers the memory of the pages' metadata, which accounts to about 1.6% of the guest's physical memory.

Evaluation. To evaluate the performance of the "memory discard" hyperupcall, we measure its impact on a guest whose memory is reclaimed due to memory pressure. When memory is scarce, hypervisors can perform "uncooperative swapping"—directly reclaim guest memory and swap it out to disk. This approach, however, often leads to suboptimal reclamation decisions. Alternatively, hypervisors can use *memory ballooning*, a paravirtual mechanism in which a guest module is informed on host memory pressures and causes the guest to reclaim memory directly [71]. The guest can then make knowledgeable reclamation decisions and discard free pages. Although memory ballooning usually performs well, performance suffers when memory needs to be abruptly reclaimed [4, 6] or when the guest disk is set on a network attached storage [68], and it is therefore not used under high memory pressure [21].

To evaluate memory ballooning, uncooperative swapping and swapping with hyperupcalls we run a scenario in which memory and physical CPU need to be abruptly reclaimed, such as to accommodate a new guest. In the guest, we start and exit "memhog", making 4GB available to be reclaimed in the guest. Next, we make the guest busy by running a CPU intensive task with low memory footprint - the SysBench CPU benchmark, which computes primes using all VCPUs [39].

Now, with the the system busy, we simulate the need to reclaim resources to start a new guest by increasing memory and CPU overcommitment. We lower the number of physical CPUs available to the guest and restrict it to only 1GB of memory. We measure the time it takes to reclaim memory against the number of physical CPUs that were allocated for the guest (Figure 3a). This simulates a new guest starting up. Then, we stop increasing memory pressure and measure the time to run a guest application with a large memory footprint using the SysBench file read benchmark on 4GB (Figure 3b). This simulates the guest reusing pages that have been reclaimed by the hypervisor.

Ballooning reclaims memory slowly (up to 110 seconds) when physical CPUs are overcommitted, as the memory reclamation operations compete with the CPU intensive tasks on CPU time. Uncooperative swapping (swap-base) can reclaim faster (32 seconds), but as it is oblivious to whether memory pages are free, it incurs higher overhead in refaulting guest free pages. In contrast, when hyperupcalls are used, the hypervisor can promote free pages' reclamation and discard them, thereby reclaiming memory up to 8 times faster than bal-

loon, with only 10% slowdown in refaulting the memory.

CPU overcommitment, of course, is not the only scenario where ballooning is non-responsive or unusable. Hypervisors refrain from ballooning when memory pressure is very high, and use host-level swapping instead [67]. It is possible for hyperupcalls to operate synergistically with ballooning: the hypervisor may use the balloon normally and use hyperupcalls when resource pressures are high or the balloon is not responding.

4.4 Tracing

Event tracing is an important tool for debugging correctness and performance issues. However, collecting traces for virtualized workloads is somewhat limited. Traces collected inside a guest do not show hypervisor events, such as when a VM-exit is forced, which can have significant effect on performance. For traces that are collected in the hypervisor to be informative, they require knowledge about guest OS symbols [15]. Such traces cannot be collected in cloud environments. In addition, each trace collects only part of the events and does not show how guest and hypervisor events interleave.

To address this issue, we run the Linux kernel tracing tool, `ftrace` [57], inside a hyperupcall. `Ftrace` is well suited to run in a hyperupcall. It is simple, lockless, and built to enable concurrent tracing in multiple contexts: non-maskable interrupt (NMI), hard and soft interrupt handlers and user processes. As a result, it was easily be adapted to trace hypervisor events concurrently with guest events. Using the `ftrace` hyperupcall, the guest can trace both hypervisor and guest events in one unified log, easing debugging. Since tracing all events use only guest logic, new OS versions can change the tracing logic, without requiring hypervisor changes.

Evaluation. Tracing is efficient, despite the hyperupcallcomplexity (3308 eBPF instructions), as most of the code deals with infrequent events that handles situations in which trace pages fill up. Tracing using hyperupcalls is slower than using native code by 232 cycles, which is still considerably shorter time than the time a context switch between the hypervisor and the guest takes.

Tracing is a useful tool for performance debugging, which can expose various overheads [79]. For example, by registering the `ftrace` on the VM-exit event, we see that many processes, including short-lived ones, trigger multiple VM exits due to the execution of the `CPUID` instruction, which enumerates the CPU features and must be emulated by the hypervisor. We find that the GNU C Library, which is used by most Linux applications, uses `CPUID` to determine the supported CPU features. This overhead could be prevented by extending Linux virtual dynamic shared object (vDSO) for applications to query the supported CPU features without triggering an exit.

4.5 Kernel Self-Protection

One common security hardening mechanisms that OSs employ is "self-protection": OS code and immutable data write protection. However, this protection is done using page tables, allowing malware to circumvent it by modifying page table entries. To prevent such attacks, the use of nested page tables has been suggested, as these tables are inaccessible from the guest [50].

However, nesting can only provide a limited number of policies and for example, cannot whitelist guest code that is allowed to access protected memory. Hyperupcalls are much more expressive, allowing the guest to specify memory protection in a flexible manner.

We use hyperupcalls to provide hypervisor-level guest kernel self-protection, which can be easily modified to accommodate complex policies. In our implementation the guest sets a bitmap which marks protected pages, and registers hyperupcall on exit events, which checks the exit reason, whether a memory access occurred and if the guest attempted to write to protected memory according to the bitmap. If there is an attempt to access protected memory, a VM shutdown is triggered. The guest sets an additional hyperupcall on the "page map" event, which queries the required protection of the guest page frames. This hyperupcall prevents situations in which the hypervisor proactively prefaults guest memory.

Evaluation. This hyperupcall code is simple, yet incurs overhead of 43 cycles per exit. Arguably, only workloads which already experience very high number of context switches would be affected by the additional overheads. Modern CPUs prevent such frequent switches.

5 Conclusion

Bridging the semantic gap is critical performance and for the hypervisor to provide advanced services to guests. Hypercalls and upcalls are now used to bridge the gap, but they have several drawbacks: hypercalls cannot be initiated by the hypervisor, upcalls do not have a bounded runtime, and both incur the penalty of context switches. Introspection, an alternative which avoids context switches can be unreliable as it relies on observations instead of an explicit interface. Hyperupcalls overcome these limitations by allowing the guest to expose its *logic* to the hypervisor, avoiding a context switch by enabling the hyperupcall to safely execute guest logic directly.

We have built a complete infrastructure for developing hyperupcalls which allow developers to easily add new paravirtual features using the codebase of the OS. We have written and evaluated several hyperupcalls and show hyperupcalls improve virtualized performance by up to 2×, ease debugging of virtualized workloads and improve VM security.

References

[1] Ferrol Aderholdt, Fang Han, Stephen L Scott, and Thomas Naughton. Efficient checkpointing of virtual machines using virtual machine introspection. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 414–423, 2014.

[2] Nadav Amit. Patch: Wrong size-extension check in BPFDAGToDAGISel::SelectAddr. `https://lists.iovisor.org/pipermail/iovisor-dev/2017-April/000723.html`, 2017.

[3] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. vIOMMU: efficient IOMMU emulation. In *USENIX Annual Technical Conference (ATC)*, 2011.

[4] Nadav Amit, Dan Tsafrir, and Assaf Schuster. VSwapper: A memory swapper for virtualized environments. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 349–366, 2014.

[5] Nadav Amit, Michael Wei, and Cheng-Chun Tu. Hypercallbacks: Decoupling policy decisions and execution. In *ACM Workshop on Hot Topics in Operating Systems (HOTOS)*, pages 37–41, 2017.

[6] Kapil Arya, Yury Baskakov, and Alex Garthwaite. Tesseract: reconciling guest I/O and hypervisor swapping in a VM. In *ACM SIGPLAN Notices*, volume 49, pages 15–28, 2014.

[7] Anish Babu, MJ Hareesh, John Paul Martin, Sijo Cherian, and Yedhu Sastri. System performance evaluation of para virtualization, container virtualization, and full virtualization using Xen, OpenVX, and Xenserver. In *IEEE International Conference on Advances in Computing and Communications (ICACC)*, pages 247–250, 2014.

[8] Sina Bahram, Xuxian Jiang, Zhi Wang, Mike Grace, Jinku Li, Deepa Srinivasan, Junghwan Rhee, and Dongyan Xu. Dksm: Subverting virtual machine introspection for fun and profit. In *IEEE Symposium on Reliable Distributed Systems*, pages 82–91, 2010.

[9] Mirza Basim Baig, Connor Fitzsimons, Suryanarayanan Balasubramanian, Radu Sion, and Donald E. Porter. Cloudflow: Cloud-wide policy enforcement using fast vm introspection. In *IEEE International Conference on Cloud Engineering (IC2E)*, pages 159–164, 2014.

[10] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. Detecting kernel-level rootkits using data structure invariants. *IEEE Transactions on Dependable and Secure Computing*, 8(5):670–684, 2011.

[11] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[12] Jason Baron. Static keys. Linux-4.8:Documentation/static-keys.txt, 2012.

[13] Brian N Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility safety and performance in the spin operating system. *ACM SIGOPS Operating Systems Review (OSR)*, 29(5):267–283, 1995.

[14] Big Switch Networks. Userspace eBPF VM. `https://github.com/iovisor/ubpf`, 2015.

[15] Martim Carbone, Alok Kataria, Radu Rugina, and Vivek Thampi. VProbes: Deep observability into the ESXi hypervisor. VMware Technical Journal `https://labs.vmware.com/vmtj/vprobes-deep-observability-into-the-esxi-hypervisor`, 2014.

[16] Andrew Case, Lodovico Marziale, and Golden G Richard. Dynamic recreation of kernel data structures for live forensics. *Digital Investigation*, 7:S32–S40, 2010.

[17] Peter M Chen and Brian D Noble. When virtual is better than real. In *IEEE Workshop on Hot Topics in Operating Systems (HOTOS)*, pages 133–138, 2001.

[18] Jui-Hao Chiang, Han-Lin Li, and Tzi-cker Chiueh. Introspection-based memory de-duplication and migration. In *ACM SIGPLAN Notices*, volume 48, pages 51–62, 2013.

[19] Tzi-cker Chiueh, Matthew Conover, and Bruce Montague. Surreptitious deployment and execution of kernel agents in windows guests. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 507–514, 2012.

[20] Eddie Dong and Will Auld. COLO: COarse-grain LOckstepping virtual machine for non-stop service. Linux Plumbers Conference, 2012.

[21] Craig Thomas Ellrod. *Optimizing Citrix XenDesktop for High Performance*. Packt Publishing, 2015.

[22] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. *Exokernel: An operating system architecture for application-level resource management*, volume 29. 1995.

[23] Roy T. Fielding and Gail Kaiser. The Apache HTTP server project. *IEEE Internet Computing*, 1(4):88–90, 1997.

[24] Yangchun Fu and Zhiqiang Lin. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *IEEE Symposium on Security and Privacy (SP)*, pages 586–600, 2012.

[25] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *The Network and Distributed System Security Symposium (NDSS)*, volume 3, pages 191–206, 2003.

[26] Zhongshu Gu, Zhui Deng, Dongyan Xu, and Xuxian Jiang. Process implanting: A new active introspection framework for virtualization. In *IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 147–156, 2011.

[27] Nadav Har'El, Abel Gordon, Alex Landau, Muli Ben-Yehuda, Avishay Traeger, and Razya Ladelsky. Efficient and scalable paravirtual I/O system. *USENIX Annual Technical Conference (ATC)*, 2013.

[28] Jennia Hizver and Tzi-cker Chiueh. Real-time deep virtual machine introspection and its applications. In *ACM SIGPLAN Notices*, volume 49, pages 3–14, 2014.

[29] Owen S Hofmann, Alan M Dunn, Sangman Kim, Indrajit Roy, and Emmett Witchel. Ensuring operating system kernel integrity with OSck. In *ACM SIGARCH Computer Architecture News (CAN)*, volume 39, pages 279–290, 2011.

[30] Jann Horn. Speculative execution, variant 4: speculative store bypass. `https://bugs.chromium.org/p/project-zero/issues/detail?id=1528`, 2018.

[31] Nur Hussein. Randomizing structure layout. `https://lwn.net/Articles/722293/`, 2017.

[32] Amani S Ibrahim, James Hamlyn-Harris, John Grundy, and Mohamed Almorsy. Cloudsec: a security monitoring appliance for virtual machines in the iaas cloud model. In *IEEE International Conference on Network and System Security (NSS)*, pages 113–120, 2011.

[33] Intel. Retpoline, a branch target injection mitigation. `https://software.intel.com/sites/default/files/managed/1d/46/Retpoline-A-Branch-Target-Injection-Mitigation.pdf?source=techstories.org`, 2018.

[34] IO Visor Project. BCC - tools for BPF-based Linux IO analysis, networking, monitoring, and more. `https://github.com/iovisor/bcc`, 2015.

[35] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through VMM-based out-of-the-box semantic view reconstruction. In *ACM Conference on Computer and Communications Security (CCS)*, pages 128–138, 2007.

[36] Stephen T Jones, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *USENIX Annual Technical Conference (ATC)*, pages 1–14, 2006.

[37] Andi Kleen. Linux virtual memory map. Linux-4.8:Documentation/x86/x86_64/mm.txt, 2004.

[38] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, January 2018.

[39] Alexey Kopytov. SysBench: a system performance benchmark. `https://github.com/akopytov/sysbench`, 2017.

[40] Sanjay Kumar, Himanshu Raj, Karsten Schwan, and Ivan Ganev. Re-architecting VMMs for multicore systems: The sidecore approach. In *Workshop on Interaction between Opearting Systems & Computer Architecture (WIOSCA)*, 2007.

[41] Nitesh Narayan Lal. KVM guest page hinting RFC. KVM mailing list `http://www.spinics.net/lists/kvm/msg153666.html`, 2017.

[42] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. *Pre-virtualization: Slashing the cost of virtualization*. Universität Karlsruhe, Fakultät für Informatik, 2005.

[43] Andrew Lutomirski. Fix flush_tlb_page() on Xen. Linux Kernel Mailing List, `https://patchwork.kernel.org/patch/9693379/`, 2017.

[44] Andrew Lutomirski. Patch: x86/hyper-v: use hypercall for remote TLB flush. Linux Kernel Mailing List, `http://www.mail-archive.com/linux-`

kernel@vger.kernel.org/msg1402180.html, 2017.

[45] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *ACM SIGPLAN Notices*, volume 48, pages 461–472, 2013.

[46] Microsoft. Hypervisor top level functional specification v5.0b, 2017.

[47] Aleksandar Milenkoski, Bryan D Payne, Nuno Antunes, Marco Vieira, and Samuel Kounev. Experience report: an analysis of hypercall handler vulnerabilities. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 100–111, 2014.

[48] Preeti Mishra, Emmanuel S Pilli, Vijay Varadharajan, and Udaya Tupakula. Intrusion detection techniques in cloud environment: A survey. *Journal of Network and Computer Applications*, 77:18–47, 2017.

[49] Quentin Monnet. Rust virtual machine and JIT compiler for eBPF programs. https://github.com/qmonnet/rbpf, 2017.

[50] Jun Nakajima and Sainath Grandhi. Kernel protection using hardware based virtualization. KVM Forum, 2016.

[51] George C Necula. Proof-carrying code. In *ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages (POPL)*, pages 106–119, 1997.

[52] Jiannan Ouyang, John R Lange, and Haoqiang Zheng. Shoot4U: Using VMM assists to optimize TLB operations on preempted vCPUs. In *ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, 2016.

[53] Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *IEEE Symposium on Security and Privacy (SP)*, pages 233–247, 2008.

[54] Donald E Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C Hunt. Rethinking the library OS from the top down. In *ACM SIGPLAN Notices*, volume 46, pages 291–304, 2011.

[55] Adit Ranadive, Ada Gavrilovska, and Karsten Schwan. Ibmon: monitoring VMM-bypass capable infiniband devices using memory introspection. In *Workshop on System-level Virtualization for HPC (HPCVirt)*, pages 25–32, 2009.

[56] Dannies M. Ritchie and Ken Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 57(6):1905–1929, 1978.

[57] Steven Rostedt. Debugging the kernel using Ftrace. LWN.net, http://lwn.net/Articles/365835/, 2009.

[58] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review (OSR)*, 42(5):95–103, 2008.

[59] Joanna Rutkowska and Alexander Tereshkin. Bluepilling the Xen hypervisor. *BlackHat USA*, 2008.

[60] Martin Schwidefsky, Hubertus Franke, Ray Mansell, Himanshu Raj, Damian Osisek, and JongHyuk Choi. Collaborative memory management in hosted Linux environments. In *Ottawa Linux Symposium (OLS)*, volume 2, pages 313–330, 2006.

[61] Jiangyong Shi, Yuexiang Yang, and Chuan Tang. Hardware assisted hypervisor introspection. *SpringerPlus*, 5(1):647, 2016.

[62] Ming-Wei Shih, Mohan Kumar, Taesoo Kim, and Ada Gavrilovska. S-NFV: Securing NFV states by using SGX. In *ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization (SDN-NFV)*, pages 45–48, 2016.

[63] Alexei Starovoitov. Patch: BPF: introduce BPF_JIT_ALWAYS_ON config. Linux Kernel Mailing List, https://lkml.org/lkml/2018/1/9/858, 2018.

[64] Sahil Suneja, Canturk Isci, Vasanth Bala, Eyal De Lara, and Todd Mummert. Non-intrusive, out-of-band and out-of-the-box systems monitoring in the cloud. In *ACM SIGMETRICS Performance Evaluation Review (PER)*, volume 42, pages 249–261, 2014.

[65] William Tu. bpf invalid stack off=-528. IO Visor mailing list https://lists.iovisor.org/pipermail/iovisor-dev/2017-April/000724.html, 2017.

[66] William Tu. Direct packet access boundary check. IO-Visor mailing list https://lists.iovisor.org/pipermail/iovisor-dev/2017-January/000604.html, 2017.

[67] vSphere memory states. http://www.running-system.com/vsphere-6-esxi-memory-states-and-reclamation-techniques/.

[68] Best practices for running VMware vSphere on network attached storage. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/vmware-nfs-bestpractices-white-paper-en.pdf, 2009.

[69] VMware. open-vm-tools. https://github.com/vmware/open-vm-tools, 2017.

[70] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review (OSR)*, volume 27, pages 203–216, 1994.

[71] Carl A Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review (OSR)*, 36(SI):181–194, 2002.

[72] Gary Wang, Zachary John Estrada, Cuong Manh Pham, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. Hypervisor introspection: A technique for evading passive virtual machine monitoring. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2015.

[73] Wei Wang. Support reporting free page blocks patch. Linux Kernel Mailing List, https://lkml.org/lkml/2017/7/12/253, 2017.

[74] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. Jitk: A trustworthy in-kernel interpreter infrastructure. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pages 33–47, 2014.

[75] Rafal Wojtczuk. Analysis of the attack surface of Windows 10 virtualization-based security. Black-Hat USA, 2016.

[76] Timothy Wood, Prashant J Shenoy, Arun Venkataramani, and Mazin S Yousif. Black-box and gray-box strategies for virtual machine migration. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, volume 7, pages 17–17, 2007.

[77] Britta Wuelfing. Kroah-Hartman: Remove Hyper-V driver from kernel? Linux Magazine http://www.linux-magazine.com/Online/News/Kroah-Hartman-Remove-Hyper-V-Driver-from-Kernel, 2009.

[78] Xen Project. XenParavirtOps. https://wiki.xenproject.org/wiki/XenParavirtOps, 2016.

[79] Haoqiang Zheng and Jason Nieh. WARP: Enabling fast CPU scheduler development and evaluation. In *IEEE International Symposium On Performance Analysis of Systems and Software (ISPASS)*, pages 101–112, 2009.