Experience Report: An Analysis of Hypercall Handler Vulnerabilities

Aleksandar Milenkoski*, Bryan D. Payne†, Nuno Antunes‡, Marco Vieira‡ and Samuel Kounev§
*Karlsruhe Instutute of Technology, Karlsruhe, Germany
Email: milenkoski@kit.edu
†Nebula Inc., Mountain View, California, USA
Email: bdpayne@acm.org
‡University of Coimbra, Coimbra, Portugal
Email: {nmsa, mvieira}@dei.uc.pt
§University of Würzburg, Würzburg, Germany
Email: samuel.kounev@uni-wuerzburg.de

***Abstract*—Hypervisors are becoming increasingly ubiquitous with the growing proliferation of virtualized data centers. As a result, attackers are exploring vectors to attack hypervisors, against which an attack may be executed via several attack vectors such as device drivers, virtual machine exit events, or hypercalls. Hypercalls enable intrusions in hypervisors through their hypercall interfaces. Despite the importance, there is very limited publicly available information on vulnerabilities of hypercall handlers and attacks triggering them, which significantly hinders advances towards monitoring and securing these interfaces. In this paper, we characterize the hypercall attack surface based on analyzing a set of vulnerabilities of hypercall handlers. We systematize and discuss the errors that caused the considered vulnerabilities, and activities for executing attacks triggering them. We also demonstrate attacks triggering the considered vulnerabilities and analyze their effects. Finally, we suggest an action plan for improving the security of hypercall interfaces.**

***Keywords*-vulnerability analysis; hypercalls; hypervisor security;**

I. Introduction

In recent years, virtualization has received increasing interest, both from industry and academia, as a way to reduce costs through server consolidation and to enhance the flexibility of physical infrastructures. In a virtualized system, which is governed by a hypervisor, each virtual machine (VM) accesses the physical resources of the infrastructure through the hypervisor and is entitled to a predefined fraction of its capacity. While server consolidation through virtualization provides many benefits it also introduces some new challenges. For instance, Gens et al. [1] report that security is a major concern for users of modern virtualized service infrastructures. The introduction of a hypervisor and the allocation of potentially multiple VMs on a single physical server are additional critical aspects introducing new potential threats and vulnerabilities [2]. Attackers are actively exploring virtualization-specific attack surfaces such as hypervisors.

Hypercalls are software traps from a kernel of a fully or partially paravirtualized guest VM to the hypervisor. They can enable intrusion into vulnerable hypervisors, initiated from a malicious guest VM kernel, in a procedural manner through hypervisors' hypercall interfaces. The exploitation of a vulnerability of a hypercall handler may lead to altering the hypervisor's memory enabling, for example, the execution of malicious code with hypervisor privilege, as demonstrated by Rutkowska and Wojtczuk [3].

The number of vulnerabilities of hypercall handlers (i.e., hypercall vulnerabilities) is expected to increase over time as the size and complexity of hypervisors' hypercall interfaces increases. For instance, the introduction of the transcendent memory (TMEM) hypercalls of the Xen hypervisor lead to the introduction of 5 critical vulnerabilities [4]. In addition, given the widespread use of hypervisors, for example, in the context of cloud computing, the incentive of attackers to discover vulnerabilities of hypervisors, including vulnerabilities of hypercall handlers, is high.

Despite the importance of hypercall vulnerabilities, there is not much publicly available information on them. Publicly disclosed vulnerability reports describing hypercall vulnerabilities (e.g., CVE-2013-4494, CVE-2013-3898) are typically the sole source of information and provide only high-level descriptions. In addition, there is no publicly available information on attacks triggering hypercall vulnerabilities (i.e., hypercall attacks) performed in practice and there are only a few publicly available attack scripts that demonstrate hypercall attacks. Finally, to the best of our knowledge, there is no previous work examining the hypercall attack surface in detail. As a result, the characterization of the hypercall attack surface is very challenging. Yet this is crucial for better understanding of the security threats that hypercall interfaces pose, which will help to focus approaches for improving the security of hypervisors.

The aim of this paper is to shed more light on the hypercall attack surface. To this end, we present an in-depth analysis of 35 hypercall vulnerabilities, which we found by searching major vulnerability report databases (e.g., cvedetails [5]) based on relevant keywords. We discuss issues, challenges, and gaps that apply specifically to securing hypercall interfaces and how they differ from security concerns related to system calls. Our analysis is based on information obtained by reverse engineering released patches fixing the considered vulnerabilities. In summary, the contributions of this paper are

- ◦ systematization and analysis of the origins of the considered hypercall vulnerabilities;
- ◦ demonstration of hypercall attacks and analysis of their effects;
- ◦ hypercall attack models based on systematization of activities for executing hypercall attacks; and
- ◦ discussion on future research directions focusing on both proactive and reactive approaches for securing hypercall interfaces.

This paper is organized as follows: in Section II, we present the set of hypercall vulnerabilities that we analyze and the applied analysis method; in Section III, we systematize and discuss the origins of the analyzed vulnerabilities and attackers' activities for executing hypercall attacks; in Section IV, we identify open issues and we propose future research directions; in Section V, we summarize this paper.

II. Sample Set of Hypercall Vulnerabilities

Table I lists the hypercall vulnerabilities analyzed for this study. We analyzed 35 hypercall vulnerabilities, which we found by searching major CVE (Common Vulnerabilities and Exposures) report databases (e.g., cvedetails [5]) based on relevant keywords, such as names of operations of hypercalls. In order to ensure representativeness of the considered set of vulnerabilities, we analyzed *all* publicly disclosed hypercall vulnerabilities that we found as previously described, expecting that the number of those that we did not find is negligibly small.

The majority of the vulnerabilities listed in Table I are from the Xen hypervisor [6]. This is because Xen has the most extensive hypercall interface, which enables full paravirtualization of guest VMs, as opposed to other hypervisors, such as KVM (Kernel-based Virtual Machine) [7], which enables only partial paravirtualization of guest VMs. The ioctl (input/output control) calls that the KVM hypervisor supports are functionally and conceptionally very similar to hypercalls of the Xen hypervisor. For the purpose of this study, in addition to its standard hypercall interface, we consider the ioctl interface of the KVM hypervisor, and therefore vulnerabilities in handlers of ioctl calls, as hypercall interface and hypercall vulnerabilities, respectively.

At the time of writing, the total number of publicly disclosed hypercall vulnerabilities is small. Based on what we observed when analyzing the vulnerabilities, we argue that this is mainly because the assessment of hypercall handlers for vulnerabilities is a challenging task due to the complexity of the operations that hypercall handlers perform. In addition, the lack of instructive documentation of the program code of hypercall handlers makes the assessment for vulnerabilities by an analyst, who has not been involved in the design and/or development of the handlers, an especially challenging task. As a result, as one can conclude from the range of releases of hypervisors affected by hypercall vulnerabilities (Table I, column 'release' of 'affected hypervisor'), it took a significant amount of time for some hypercall vulnerabilities to be discovered (e.g., CVE-2012-5514, CVE-2013-4494).

Our approach for analyzing a hypercall vulnerability consisted of the following steps: *i)* analysis of the CVE report describing the vulnerability and of other relevant information sources, for example, security advisories; *ii)* reverse-engineering of the released patch fixing the vulnerability; and *iii)* developing proof-of-concept code for triggering the vulnerability in a testbed environment.

The execution of proof-of-concept code enabled us to closely observe all stages of an attack life cycle — pre-attack activities, followed by execution of a hypercall, or a series of hypercalls, triggering a given vulnerability, and finally, the post-attack state of the targeted hypervisor.

We analyzed vulnerabilities of closed-source hypervisors (i.e., CVE-2013-3898) and vulnerabilities that are not likely to be triggered in practice to a lesser extent than what is described above (i.e., we did not develop proof-of-concept code). This analysis, although not as extensive as the one described above, still provided us with enough information for making relevant observations. We classified a few vulnerabilities as not likely to be triggered in practice:

- ◦ vulnerabilities of hypervisors not likely to be deployed in production environments, for example, an open-source hypervisor compiled with its debugging feature enabled — CVE-2012-3516 and CVE-2013-0154;
- ◦ vulnerabilities that can be triggered only from guest VMs that are not likely to be seen in production environments, for example, a guest VM with a large number of virtual CPUs (vCPUs) — CVE-2013-0151 and CVE-2013-4554;
- ◦ vulnerabilities in handlers of deprecated or experimental hypercalls — CVE-2012-3497 and CVE-2013-4553.

We provide information related to the criteria mentioned above in the column 'notes' of Table I.

III. Analysis of the Hypercall Attack Surface

We analyze the hypercall attack surface from two perspectives: *i)* a targeted hypervisor; and *ii)* an attacker triggering a hypercall vulnerability. Assuming the perspective of a targeted hypervisor, we provide in-depth technical information about hypercall vulnerabilities, and systematize and discuss the errors causing the hypercall vulnerabilities analyzed for this study (Section III-A). We also demonstrate attacks triggering hypercall vulnerabilities, performed by executing the proof-of-concept code that we developed (Section III-A), and discuss their effects (Section III-B).

Table I: Analyzed hypercall vulnerabilities

CVE ID	Hypercall	Affected hypervisor		Notes
		Name	Release	
CVE-2008-3687	flask_op	Xen	3.3	
CVE-2009-2287	KVM_SET_SREGS	KVM	Linux kernel 2.6.x(<2.6.30)	
CVE-2009-3290	kvm_emulate_hypercall	KVM	Linux kernel 2.6.25-rc1–2.6.31	
CVE-2009-3638	KVM_GET_SUPPORTED_CPUID	KVM	Linux kernel 2.6.25-rc1–2.6.32-rc4	
CVE-2009-4004	KVM_X86_SETUP_MCE	KVM	Linux kernel <2.6.32-rc7	
CVE-2010-3698	KVM_RUN	KVM	Linux kernel <2.6.36	
CVE-2011-4347	KVM_ASSIGN_PCI_DEVICE	KVM	Linux kernel 3.1.7–3.1.9	
CVE-2012-1601	KVM_CREATE_IRQCHIP	KVM	Linux kernel <3.3.6	
CVE-2012-3494	set_debugreg	Xen	4.0.x–4.1.x(<4.1.4)	
CVE-2012-3495	physdev_op	Xen	4.1.x(<4.1.4)	
CVE-2012-3496	memory_op	Xen	3.9.x–4.1.x(<4.1.4)	
CVE-2012-3497	tmem_op	Xen	>4.0.x	Only hypervisors with the TMEM feature enabled are vulnerable
CVE-2012-3516	grant_table_op	Xen	4.2.0	Only unstable hypervisor releases are vulnerable
CVE-2012-4461	KVM_SET_SREGS	KVM	Linux kernel <3.6.9	
CVE-2012-4538	hvm_op	Xen	4.0.x–4.1.x(<4.1.4), 4.2.x(<4.2.1)	
CVE-2012-4539	grant_table_op	Xen	4.0.x–4.1.x(<4.1.4)	
CVE-2012-5510	grant_table_op	Xen	4.x(<4.1.4)	
CVE-2012-5513	memory_op	Xen	<4.1.4	
CVE-2012-5514	memory_op	Xen	3.4.x–4.1.x(<4.1.4)	
CVE-2012-5515	memory_op	Xen	3.4.x–4.1.x(<4.1.4)	
CVE-2012-5525	mmuext_op	Xen	4.2.0	
CVE-2013-0151	hvm_op	Xen	4.2.x(<4.2.2)	Can be triggered only from a guest VM with a large number of vCPUs
CVE-2013-0154	mmuext_op	Xen	4.2.x(<4.2.2)	Only hypervisors with the debugging feature enabled are vulnerable
CVE-2013-1964	grant_table_op	Xen	4.0.x–4.1.x(<4.1.5)	
CVE-2013-3898	unknown	Hyper-V	Windows 8 and Server 2012	Closed-source hypervisor
CVE-2013-4494	grant_table_op	Xen	>3.2.x	
CVE-2013-4553	domctl	Xen	>3.4.x	Deprecated hypercall
CVE-2013-4554	hvm_do_hypercall	Xen	3.0.x–4.3.2	Can be triggered only from a guest VM with a non-mainstream operating system
CVE-2013-5634	KVM_GET_REG_LIST	KVM	Linux kernel 3.9.x	
CVE-2014-1891	flask_op	Xen	3.x.x–4.2.x(<4.2.4), 4.3.x(<4.3.2)	
CVE-2014-1892	flask_op	Xen	3.3.x–4.1.x	
CVE-2014-1893	flask_op	Xen	3.2.x–4.1.x	
CVE-2014-1894	flask_op	Xen	3.2.x	
CVE-2014-1895	flask_op	Xen	4.2.x(<4.2.4), 4.3.x(<4.3.2)	
CVE-2014-3124	hvm_op	Xen	>4.1.x	

Assuming the perspective of an attacker, we construct attack models based on systematizing activities for executing hypercall attacks (Section III-C).

Our two-perspective approach enables the comprehensive analysis of the hypercall attack surface, which, in turn, enables the development of an action plan for improving the security of hypercall interfaces.

A. Hypervisor's Perspective: Origins of Hypercall Vulnerabilities

In Table II, for each hypercall vulnerability, we present the type of error that caused it and the effects of a hypercall attack triggering it. We stress that the error categories presented in Table II are not intended for general use and are defined for the convenience of discussion. We defined error categories as opposed to using existing taxonomies for classifying software errors (e.g., [8]), since none of them fit well to our purpose; that is, we could not classify in the same category errors that share characteristics on which we focus in this paper.

We primarily distinguish between *implementation* errors (i.e., errors that are obviously due to programmer error) and *non-implementation* errors (i.e., errors in design, configuration, and so on).

1) Implementation Errors: We found several forms of implementation errors: *value validation errors* (i.e., missing value validation and incorrect value validation) and *incorrect implementation of inverse procedures*. It is obvious that the previously mentioned implementation errors are not

Table II: Origins of the considered hypercall vulnerabilities and effects of attacks triggering the vulnerabilities

CVE ID	Error	Effect
CVE-2008-3687	Implementation — value validation — missing value validation	Corrupted state
CVE-2009-2287	Non-implementation	Hang/crash
CVE-2009-3290	Non-implementation	Corrupted state
CVE-2009-3638	Implementation — value validation — incorrect value validation	Crash/corrupted state
CVE-2009-4004	Implementation — value validation — incorrect value validation	Crash/corrupted state
CVE-2010-3698	Non-implementation	Crash
CVE-2011-4347	Non-implementation	Crash
CVE-2012-1601	Non-implementation	Corrupted state
CVE-2012-3494	Implementation — value validation — incorrect value validation	Crash
CVE-2012-3495	Implementation — value validation — missing value validation	Crash/corrupted state
CVE-2012-3496	Non-implementation	Crash
CVE-2012-3497	Implementation — value validation — missing value validation	Crash/corrupted state
CVE-2012-3516	Implementation — value validation — missing value validation	Crash/corrupted state
CVE-2012-4461	Non-implementation	Crash
CVE-2012-4538	Non-implementation	Crash
CVE-2012-4539	Implementation — value validation — missing value validation	Hang/crash
CVE-2012-5510	Implementation — incorrect implementation of inverse procedures	Crash/corrupted state
CVE-2012-5513	Implementation — value validation — missing value validation	Crash/corrupted state
CVE-2012-5514	Non-implementation	Hang
CVE-2012-5515	Implementation — value validation — missing value validation	Hang
CVE-2012-5525	Implementation — value validation — missing value validation	Crash
CVE-2013-0151	Non-implementation	Crash
CVE-2013-0154	Non-implementation	Crash
CVE-2013-1964	Non-implementation	Crash/corrupted state/information leakage
CVE-2013-3898	Implementation — value validation — missing value validation	Crash/corrupted state
CVE-2013-4494	Non-implementation	Hang
CVE-2013-4553	Non-implementation	Hang
CVE-2013-4554	Non-implementation	Corrupted state
CVE-2013-5634	Non-implementation	Crash
CVE-2014-1891	Implementation — value validation — missing value validation	Crash/corrupted state
CVE-2014-1892	Non-implementation	Crash
CVE-2014-1893	Implementation — value validation — incorrect value validation	Crash/corrupted state
CVE-2014-1894	Implementation — value validation — incorrect value validation	Crash/corrupted state
CVE-2014-1895	Implementation — value validation — incorrect value validation	Crash/information leakage
CVE-2014-3124	Non-implementation	Crash/corrupted state

exclusive to hypercall interfaces. However, in this section, we discuss issues related to these errors that are exclusive to hypercall interfaces.

Value validation errors. We observed that most of the implementation errors causing hypercall vulnerabilities are missing value validations, followed by incorrect value validations, either of input parameters or of internal variables. Under internal variables, we understand variables that are created, and to which values are assigned, within a hypercall handler (e.g., return values of functions invoked within a hypercall handler).

An example vulnerability due to missing value validation of an input parameter is CVE-2012-5525 [9] of the Xen hypervisor, which we discuss next.

Hypercall attack 1: The *get_page_from_gfn* function, which is invoked within multiple hypercall handlers, provides information about a memory page specified with its machine frame number (MFN), whose value can be fully manipulated by a guest VM user as a hypercall input parameter. In case a malicious guest VM user provides an invalid MFN, *get_page_from_gfn* will return an invalid page information, which may cause the hypervisor to crash. An invalid MFN is a MFN that is larger than the largest MFN mapped to the guest VM invoking *get_page_from_gfn*. This is because *get_page_from_gfn* uses the user-provided MFN as an offset for reading from an array where each element contains information about a single page. We triggered CVE-2012-5525 by invoking the hypercall *HYPERVISOR_mmuext_op* (operation *MMUEXT_CLEAR_PAGE*) and providing a MFN of *0x0EEEEE*, which caused the hypervisor to crash.[1]

An example vulnerability due to missing value validation of an internal variable is CVE-2012-3495 [10] of the Xen hypervisor, which we discuss next.

Hypercall attack 2: PHYSDEVOP_get_free_pirq, an operation

[1]We executed a proof-of-concept code triggering CVE-2012-5525 in the following environment: *guest VM* — OS: Ubuntu Precise Pangolin (32 bit), kernel: 3.8.0-29-generic; *host VM* — OS: Ubuntu Precise Pangolin (32 bit), kernel 3.8.0-29-generic; *hypervisor* — Xen 4.2.0 (32 bit).

of the *physdev_op* hypercall, is used for allocating a PIRQ (PCI IRQ — Peripheral Component Interconnect Interrupt Request) to the guest VM from where it is invoked. In the handler of *PHYSDEVOP_get_free_pirq*, the return value of the function *get_free_pirq*, which corresponds to a free PIRQ if there is one, is used as an index for accessing an element of the array *pirq_irq* in order to mark a free PIRQ as allocated by writing *-1*. However, the return value of *get_free_pirq* is not validated to be a valid PIRQ and not an error code (i.e., *-28*), which *get_free_pirq* returns if there is no free PIRQ. In case *get_free_pirq* returns an error code, the error code is used as an index for accessing *pirq_irq* and as a result, *-1* is written at the memory address **(pirq_irq - 28)*, which is mapped to the hypervisor. In order to trigger CVE-2012-3495, a request for allocating a PIRQ should be made when there are no free PIRQs. It can be concluded that a repetitive execution of *PHYSDEVOP_get_free_pirq* will eventually result in triggering of CVE-2012-3495. Depending on the exact memory layout of the hypervisor, it may crash. In case the hypervisor does not crash, it may be possible to carefully craft an exploit to achieve privilege escalation. We triggered CVE-2012-3495 by executing *PHYSDEVOP_get_free_pirq* 17 times, which caused the hypervisor to crash.[2]

Missing and incorrect value validation errors are obviously due to programmer error. Missing value validation errors can be addressed by adding program code verifying values of variables. However, when analyzing the hypercall vulnerabilities of the Xen hypervisor, we observed that performing frequent value validations may reduce the execution speed of hypercalls and increase the performance overhead inccured by them in a way which we discuss next.

Given that hypercalls to hypervisors are complex instructions that take many CPU cycles to execute, hypercalls have only a limited amount of time for execution.[3] Although some hypercalls are sufficiently simple so a given time limit is enough for completing their tasks, many hypercalls are complex and cannot complete their tasks in a set time limit. Therefore, hypervisors employ a *hypercall continuation* mechanism. This mechanism saves the execution state of a hypercall and returns control to the guest VM from where the hypercall had been invoked so that the hypercall can be resumed at a later time. Given that this context switching takes time, it can be concluded that hypercall continuations increase the performance overhead incurred by hypercalls. We observed that performing frequent value validations (e.g., of both input parameters and internal variables) may cause the frequency of hypercall continuations to increase, which incurs performance overhead and reduces the execution speed of hypercalls. Note that partially and fully paravirtualized guest VMs rely heavily on hypercalls, and therefore the execution speed of hypercalls is crucial for their performance.

[2]We executed a proof-of-concept code triggering CVE-2012-3495, CVE-2012-5510, CVE-2013-4494, and CVE-2013-1964 in the following environment: *guest VM* — OS: Ubuntu Precise Pangolin (32 bit), kernel: 3.8.0-29-generic; *host VM* — OS: Ubuntu Precise Pangolin (32 bit), kernel: 3.8.0-29-generic; *hypervisor* — Xen 4.1.2 (32 bit).

[3]For instance, the hypercalls of the Microsoft hypervisor have only 50 microseconds to execute [11].

We observed that the previously discussed impact of variable value validations on the execution speed of the hypercalls of the Xen hypervisor has been a key factor for the use of specific hypercall programming practices for boosting the latter, which, however, led to introducing vulnerabilities. An evidence supporting this statement is the vulnerability CVE-2012-5513 [12], which we discuss next.

Hypercall attack 3: CVE-2012-5513 is a vulnerability of *XENMEM_exchange*, an operation of the *memory_op* hypercall of the Xen hypervisor. In the handler of *XENMEM_exchange*, the function *__copy_to_guest_offset* is used for fast data copy, from virtual memory addresses mapped to a guest VM to virtual memory addresses mapped to the hypervisor. *__copy_to_guest_offset* is fast since it does not verify memory addresses for validity. The memory addresses used by *__copy_to_guest_offset* should be validated before *__copy_to_guest_offset* is invoked, which increases the risk of an implementation error. A guest VM user can fully manipulate the values of the virtual memory addresses used by *__copy_to_guest_offset* since they are input parameters of *XENMEM_exchange*. The latter enables a malicious guest VM user to overwrite hypervisor's memory by setting the memory addresses, to which data will be copied by *__copy_to_guest_offset*, to addresses mapped to the hypervisor. We triggered CVE-2012-5513 by providing a copy destination address of *0xFFFF808000000000*, which caused the hypervisor to crash. Since an area of the hypervisor's memory is overwritten with memory addresses accessible by the guest VM from where CVE-2012-5513 is triggered, malicious code execution with hypervisor privileges may be possible for certain memory layouts of the hypervisor (e.g., by trampolining).[4]

Our observations presented above indicate that the trade-off between the two crucial properties of hypercall interfaces (i.e., performance and security) is currently an issue that should be addressed with great care. We discuss more on this issue in Section IV.

Incorrect implementation of inverse procedures. We observed that some of the hypercall vulnerabilities that we analyzed are associated with inverse procedures, out of which one vulnerability is due to incorrect implementation of such procedures. Under inverse procedures, we understand two procedures, one undoing the effects of the other, which have to be executed in a given order, such as locking and unlocking, and memory allocating and deallocating procedures. An example vulnerability due to incorrect implementation of inverse procedures is CVE-2012-5510 [13], which we discuss next.

Hypercall attack 4: CVE-2012-5510 is a vulnerability of *GNTTABOP_set_version*, an operation of the *grant_table_op* hypercall of the Xen hypervisor. *GNTTABOP_set_version* is used for downgrading, from version 2 to version 1, and upgrading, from version 1 to version 2, the grant table of a guest VM. When a grant table is downgraded, the function *gnttab_unpopulate_status_frames*, inverse to *gnttab_populate_status_frames*, is used to deallocate page frames used only by a grant table of version 2. However, *gnttab_unpopulate_status_frames* does not properly perform the

[4]We executed a proof-of-concept code triggering CVE-2012-5513 and CVE-2012-3496 in the following environment: *guest VM* — OS: Debian Squeeze (64 bit), kernel: 2.6.32-5-amd64; *host VM* — OS: Debian Squeeze (64 bit), kernel: 2.6.32-5-amd64; *hypervisor* — Xen 4.1.0 (64 bit).

standard procedure for deallocating page frames — it does not remove the nodes that are associated with the frames being deallocated from the linked list *xenpage_list*, where the hypervisor stores frame information for memory management purposes. As a result, subsequent attempts to allocate the same frames by upgrading a grant table may lead to adding a node to *xenpage_list* that is a duplicate of the node that has not been removed by *gnttab_unpopulate_status_frames*. This causes corruption of *xenpage_list* and leads to undefined behavior of the hypervisor. We triggered CVE-2012-5510 by downgrading and upgrading the grant table of a guest VM 58 times in a row, which caused the hypervisor to crash due to memory corruption.[2]

An obvious reason for the incorrect implementation of inverse procedures is their complexity, which increases the risk of implementation errors. Most of the vulnerabilities associated with inverse procedures that we analyzed are due to *non-implementation errors*; that is, we observed that often vulnerabilities are introduced by not executing inverse procedures properly (e.g., not executing one of them or executing them in an irregular order) in certain scenarios that a vulnerable hypervisor cannot properly handle. An example is the vulnerability CVE-2013-4494 [14] of the Xen hypervisor, which we discuss next.

Hypercall attack 5: Two pairs of inverse procedures (i.e., *page_alloc_lock* and *page_alloc_unlock*, which are used for locking and unlocking the page allocation structure of a guest VM, and *grant_table_lock* and *grant_table_unlock*, which are used for locking and unlocking the grant table structure of a guest VM) are executed as part of the operations of several hypercalls, however, not always in the same order (e.g., in a reverse order). The latter is not an issue in scenarios where hypercalls executing *page_alloc_lock*, *page_alloc_unlock*, *grant_table_lock*, and *grant_table_unlock* in a reverse order are executed at different times, for example, sequentially. However, in the specific scenario where such hypercalls are executed from a guest VM at the same time at separate vCPUs, a deadlock is created. We triggered CVE-2013-4494 by executing the hypercalls *grant_table_op*, operation *GNTTABOP_SETUP*, and *grant_table_op*, operation *GNTTABOP_TRANSFER*, at the same time at two separate vCPUs, which caused the hypervisor to hang.[2]

The discussion above introduces a major issue that is one of the central topics of this paper and that we focus on next.

2) Non-implementation Errors: We demonstrate the complexity that vulnerabilities due to non-implementation errors may have through an example where we trigger the vulnerability CVE-2013-1964 [15] of the Xen hypervisor.

Hypercall attack 6: GNTTABOP_copy, an operation of the *grant_table_op* hypercall, is used for copying memory pages from a source VM (SVM) to a destination VM (DVM) with respect to data access permissions set by the SVM and the DVM using grant table entries (i.e., grants). Grant tables of version 2 support transitive grants, which are used for transitive assignment of permissions such that a transitive grant points to another grant. In order for the SVM to copy a page to the DVM, it must first acquire a grant from the hypervisor and, after the page is copied, it requests a release of the grant. However, vulnerable releases of the Xen hypervisor cannot properly handle the scenario where non-transitive grants of a version 2 grant table are used (i.e., a vulnerable hypervisor releases a non-transitive grant of a version 2 grant table as if it is a transitive grant). The culprit of this error is that the hypervisor has been designed to support the specific scenario where transitive grants that point to a grant of the VM acquiring a grant are used and as a result, the hypervisor wrongly treats non-transitive grants as transitive grants. The triggering of CVE-2013-1964 results in an unrequested release of the first grant of the grant table of the VM that has executed *GNTTABOP_copy*. This may cause the hypervisor to crash, or corrupt its state, which enables further malicious activities. We triggered CVE-2013-1964 by executing *GNTTABOP_copy* such that a release of a version 2 non-transitive grant was requested, which caused the hypervisor to crash.[2]

As *Hypercall attack 6* demonstrates, we argue that a typical hypercall interface can properly handle only a certain amount of hypercall execution scenarios. As a result, a malicious guest VM user can trigger a hypercall vulnerability by creating an "unexpected" hypercall execution scenario, which the targeted hypervisor is not able to properly handle.

As opposed to the majority of hypercall vulnerabilities due to implementation errors, all hypercall vulnerabilities due to non-implementation errors can be triggered by executing a regular hypercall, or a series of regular hypercalls, in a way such that the hypervisor cannot properly handle. By *regular hypercalls*, we mean hypercalls that are not specifically crafted for triggering vulnerabilities (i.e., with specifically crafted parameter values), which may be invoked as part of regular system operation. For example, we were able to crash the Xen hypervisor by simply executing *XENMEM_populate_physmap* (an operation of the *memory_op* hypercall) with valid parameter values from an auto-translated paravirtualized guest VM, only because *XENMEM_populate_physmap* is not intended for use by such a VM (we triggered the vulnerability CVE-2012-3496 [16]).[4]

The fact that non-implementation errors causing hypercall vulnerabilities (many of which can be triggered unintentionally as part of regular system operation) are common, raises the question — *are hypercall interfaces of hypervisors reliable*? The primary task of hypervisors is to manage the operation of multiple guest VMs, where each guest VM is of a given architecture (e.g., 32 or 64 bit), runs a given operating system, is virtualized in a specific way (e.g., fully paravirtualized, partially paravirtualized, hardware virtualized), may be in a given state at any point in time (e.g., running, rebooting, halted), has a certain amount of resources allocated to it (e.g., vCPUs), and so on. Given the complexity of the previously mentioned task, we argue that designing and developing the hypercall interfaces of hypervisors in a way such that they are able to reliably handle any hypercall execution scenario without exposing attack vectors is challenging.

An overlapping objective of the areas of hypervisor security and system reliability is the prevention of system failures, which is critical in mission- and business-critical virtualized environments. The system security and reliability communities advocate the need for advances that enable the computer system reliability community to better prevent failures to be adapted and shared with the former for possible implementation [17], [18]. Given the observations from our study, we recognize such a need when it comes to securing hypercall interfaces of hypervisors.

Our observation reveals that for the triggering of many of the current hypercall vulnerabilities (i.e., those due to non-implementation errors, see Table II), the way in which a hypercall, or a series of hypercalls, is executed plays a key role. This raises several issues related to the efficiency of existing mechanisms for securing hypercall interfaces, for example, intrusion detection and prevention systems. We discuss these issues in detail in Section IV.

B. Hypervisor's Perspective: Effects of Hypercall Attacks

We present in Table II the effects of the attacks triggering the considered vulnerabilities on the states of the targeted hypervisors. We observed that hypercall attacks are very efficient in obstructing the operation of a hypervisor, either by causing it to crash (effect *crash* in Table II, see for example *Hypercall attack 6*, Section III-A2) or to hang (effect *hang* in Table II, see for example *Hypercall attack 5*, Section III-A1). This is expected given that hypercalls perform system-critical operations. As a result, we argue that hypercall attacks can be very effective hypervisor denial-of-service (DoS) attacks, which are considered as very severe in mission- and business- critical virtualized environments where availability of hypervisors is of high importance.

Some hypercall attacks corrupt the state of the hypervisors they target without causing them to crash (effect *corrupted state* in Table II), part of which only if a given condition is satisfied (effect *crash/corrupted state* in Table II), for example, if the memory of the targeted hypervisor is laid out in a specific way (see for example *Hypercall attack 2*, Section III-A1). Our analysis of the post-attack states of hypervisors targeted by attacks corrupting their states revealed that severe intrusions leading to, for example, malicious code execution with hypervisor privilege, *are probable* (see for example *Hypercall attack 3*, Section III-A1). However, it *does not* seem likely that the execution of a single hypercall attack will lead to such an intrusion. On the contrary, our analysis showed that a hypercall attack is an effective mechanism for intruding hypervisors when executed as part of an elaborate multi-step attack, in which the task of the hypercall attack is to corrupt the state of the hypervisor and pave the way for further malicious activities. We found that this also holds for hypercall attacks resulting in an unauthorized retrieval of information (effect *information leakage* in Table II), read from memory allocated to the hypervisor or a guest VM collocated with the VM from where a hypercall attack is executed.

We note that, although not common, intrusions achieved by executing a single hypercall attack can happen and they are normally of the highest severity (i.e., they result in malicious code execution with hypervisor privilege). An example is the hypercall attack triggering the vulnerability CVE-2008-3687 demonstrated by Rutkowska and Wojtczuk [3] at the Black Hat 2008 conference.

C. Attacker's Perspective: Attack Models

Based on analyzing the attacks triggering the vulnerabilities listed in Table I, we identified patterns of activities comprising a successful attack campaign. We then categorized the identified patterns into attack models. Models of hypercall attacks facilitate the development of approaches for improving the security of hypercall interfaces where mimicking attackers targeting hypervisors via their hypercall interfaces is needed, for example, discovery of vulnerabilities by fuzzing. We discuss such approaches in detail in Section IV.

We distinguish two phases of a hypercall attack: *setup* and *attack execution* phase. A setup phase consists of execution of one or multiple regular hypercalls setting up the virtualized environment as necessary for triggering a given hypercall vulnerability and does not always take place. An attack execution phase consists of

- ○ execution of a *single* hypercall with
  - – *regular* parameter value(s) (i.e., regular hypercall), or
  - – parameter value(s) *specifically crafted* for triggering a given vulnerability, or
- ○ execution of a *series of regular hypercalls* in a given *order*, including
  - – *repetitive* execution of a *single* hypercall, or
  - – *repetitive* execution of *multiple* hypercalls.

The attack models involving execution of a single or multiple regular hypercalls assume that the hypercall(s) are executed in a way such that

- ○ the targeted hypervisor cannot properly handle the hypercalls, which is typical for triggering vulnerabilities due to non-implementation errors, or
- ○ an erroneous program code is executed, which is typical for triggering some vulnerabilities due to implementation errors.

As one can observe from the hypercall attack models presented above, the majority of the models involve execution only of regular hypercalls, as opposed to the intuitive assumption that most of these models would involve execution of hypercalls specifically crafted for triggering vulnerabilities. This raises a number of issues that we discuss in detail in Section IV.

In Figure 1(a)–(f), we depict the hypercall attacks described in Section III-A; that is, we provide examples of attacks that conform to the attack models that we define. In Figure 1(a)–(f), we depict only the hypercalls executed as part of a hypercall attack and relevant hypercall parameters (i.e., parameters identifying the executed hypercall, and, where applicable, parameters with values specifically crafted for triggering a vulnerability, which are marked in bold).

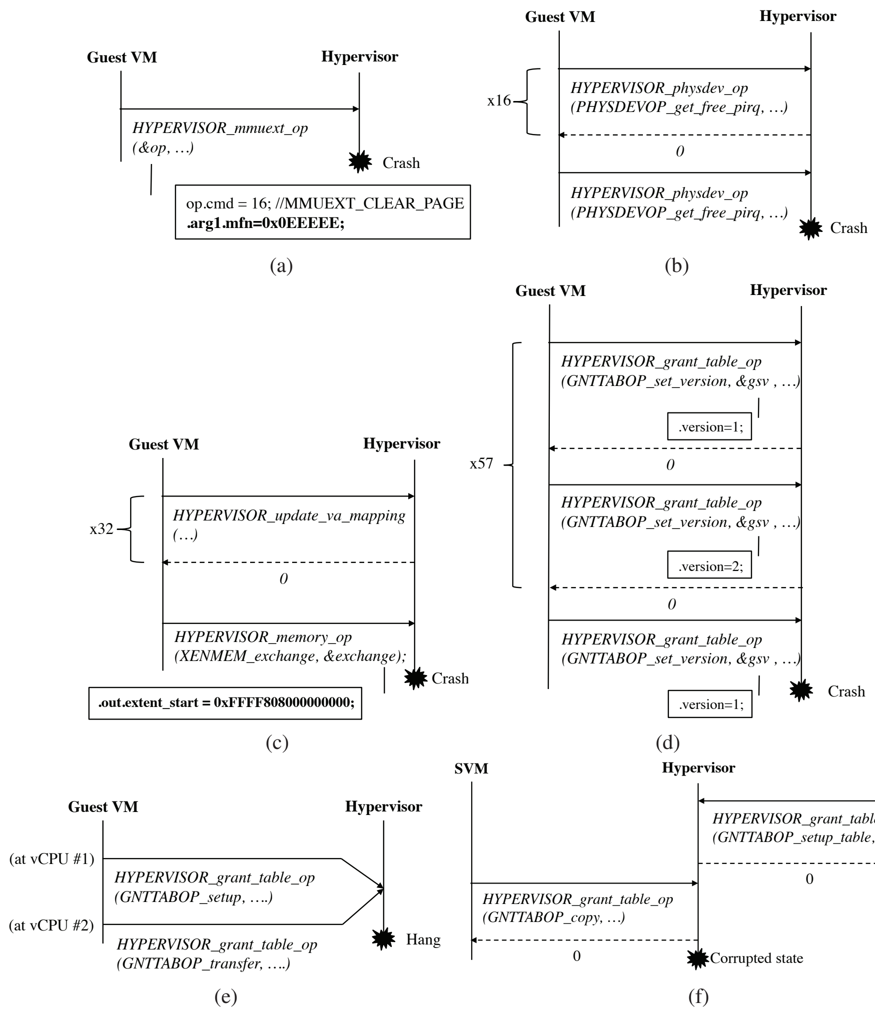
Guest VM
Hypervisor
HYPERVISOR_mmuext_op
(&op, ...)
Crash
op.cmd = 16; //MMUEXT_CLEAR_PAGE
.arg1.mfn=0x0EEEEE;
(a)
Guest VM
Hypervisor
x16
HYPERVISOR_physdev_op
(PHYSDEVOP_get_free_pirq, ...)
0
HYPERVISOR_physdev_op
(PHYSDEVOP_get_free_pirq, ...)
Crash
(b)
Guest VM
Hypervisor
x32
HYPERVISOR_update_va_mapping
(...)
0
HYPERVISOR_memory_op
(XENMEM_exchange, &exchange);
Crash
.out.extent_start = 0xFFFF808000000000;
(c)
Guest VM
Hypervisor
x57
HYPERVISOR_grant_table_op
(GNTTABOP_set_version, &gsv , ...)
.version=1;
0
HYPERVISOR_grant_table_op
(GNTTABOP_set_version, &gsv , ...)
.version=2;
0
HYPERVISOR_grant_table_op
(GNTTABOP_set_version, &gsv , ...)
Crash
.version=1;
(d)
Guest VM
Hypervisor
(at vCPU #1)
HYPERVISOR_grant_table_op
(GNTTABOP_setup, ...)
(at vCPU #2)
HYPERVISOR_grant_table_op
(GNTTABOP_transfer, ...)
Hang
(e)
SVM
Hypervisor
DVM
HYPERVISOR_grant_table_op
(GNTTABOP_setup_table, ...)
0
HYPERVISOR_grant_table_op
(GNTTABOP_copy, ...)
0
Corrupted state
(f)

Figure 1: (a) *Hypercall attack 1* — execution of a single hypercall with a specifically crafted parameter value. (b) *Hypercall attack 2* — repetitive execution of a single regular hypercall. (c) *Hypercall attack 3* — setup phase and execution of a single hypercall with a specifically crafted parameter value. (d) *Hypercall attack 4* — repetitive execution of multiple regular hypercalls. (e) *Hypercall attack 5* — execution of a series of regular hypercalls. (f) *Hypercall attack 6* — setup phase and execution of a single regular hypercall.

IV. Extending the Frontiers

Based on our observations, we now discuss how the state-of-the-art in securing hypercall interfaces can be advanced. We focus on issues related to *i) proactive* approaches for securing hypercall interfaces (i.e., preventing hypercall attacks from occuring) — vulnerability discovery and secure programming practices; and *ii) reactive* approaches for securing hypercall interfaces (i.e., detecting and preventing hypercall attacks as they occur) — security mechanisms.

A. Vulnerability Discovery and Secure Programming Practices

1) Vulnerability Discovery: We note that publicly available techniques and tools for *fuzzing* hypercalls are lacking. Such tools can be very effective for discovering hypercall vulnerabilities, especially those due to implementation errors that can be triggered by executing specifically crafted hypercalls. Hypercall fuzzing tools may contribute towards discovering hypercall vulnerabilities in a time-efficient manner by enabling vulnerability analysts, especially those who had not been involved in the design and/or development of the hypercalls that they analyze, to conveniently discover vulnerabilities. The hypercall attack models that we presented in Section III-C can serve as a basis for designing and developing generation-based fuzzers for hypercalls.

Hypercall fuzzing is challenging since unlike, for example, system calls, many hypercalls perform operations that alter the state not only of the system executing them (i.e., a guest VM), but also of the underlying hypervisor. One challenge originating from this characteristic of hypercalls is that one must ensure that during a fuzzing campaign, hypercalls are executed in a way (e.g., in a given order) such that they do not cause an undesired state of the guest VM *and/or* the hypervisor. This requires very careful planning and an in-depth knowledge on the tasks that the hypercalls to be executed perform. An undesired state of a VM or a hypervisor in a fuzzing campaign is a state that hinders the efficiency of the fuzzing process (e.g., a state that makes the execution of a code segment of a given hypercall handler impossible due to unfulfilled conditions for executing the code, which are related to the state of the hypervisor or the guest VM).

Given the severity of hypercall attacks (see Section III-B), we argue that tools for time-efficient discovery of hypercall vulnerabilities, such as fuzzers, should be designed and developed. We observed that the time period between the introduction and the discovery of many of the hypercall vulnerabilities considered in this study is long. For example, the vulnerability CVE-2012-3494 [19] was present in versions of the Xen hypervisor released over the course of 2 years and 8 months.[5] Our analysis of the hypercall vulnerabilities due to implementation errors revealed that the discovery of some of them is trivial in case fuzzing techniques are used (e.g., CVE-2012-3494).

While fuzzers are effective for discovering some vulnerabilities due to implementation errors, they are not that effective for discovering vulnerabilities due to non-implementation errors. Note that the latter are triggered by executing regular hypercalls in a way such that the hypervisor cannot properly handle. We argue that such vulnerabilities can be discovered using *formal verification methods* that aim at discovering the non-implementation errors causing hypercall vulnerabilities (see Section III-A2), which are currently lacking. We refer the reader to [20] for an overview of the challenges that apply to formally verifying hypervisors.

The Microsoft Hyper-V verification project [20] has made important achievements towards the functional verification of the Hyper-V hypervisor. Many researchers, such as Alkassar et al. [21] and Barthe et al. [22], develop formal models of various hypervisor components used for verifying functional properties of hypervisors, for example, isolation between guest VMs. Freitas and McDermott [23] formally modelled the hypercall interface of the Xen hypervisor to re-engineer it into interface enforcing information-flow security (i.e., one VM should not flow/leak information into another VM). We argue that the development of models for formally verifying the functional correctness of hypercall interfaces, with a focus on hypervisor security, would be a major achievement towards reducing the number of hypercall vulnerabilities due to non-implementation errors.

[5]The vulnerability CVE-2012-3494 has been introduced in Xen 4.0.0 (released 7 Apr. 2010), a patch has been released on 5 Sept. 2012, and CVE-2012-3494 has been fixed in Xen 4.1.4 (released 18 Dec. 2012).

2) Secure Programming Practices: In Section III-A1, we mentioned that the trade-off between performance and security of hypercall interfaces is currently an important issue. We observed that missing value validations errors comprise slightly less than 60 percent of all implemenation errors that we reviewed. Missing value validation errors can be addressed by adding program code verifying values of variables. However, when analyzing the hypercall vulnerabilities of the Xen hypervisor, we observed that performing frequent value validations (e.g., of both input parameters and internal variables) may cause an increased frequency of hypercall continuations, therefore reducing the execution speed of hypercalls and increasing the performance overhead incurred by them (see Section III-A1).

We argue that secure hypercall programming practices enforcing, for example, value validations of all variables used within a given hypercall handler, should be developed. Given our observations presented in Section III-A1, one may conclude that the application of such practices would lead to the development of secure hypercall handlers free of missing value validation errors, however, executing slowly (e.g., hypercalls of the Xen hypervisor may exhibit an increased frequency of overhead incurring hypercall continuations). Therefore, a major challenge is the development and application of programming practices for developing hypercall handlers such that rigorous value validations are performed at a reasonable performance cost.

Security enhanced operating modes of hypervisors already exist, such as the XSM-FLASK (Xen Security Modules - FLux Advanced Security Kernel) security module of the Xen hypervisor, which enables access control of hypercalls, however, at the cost of performance [24]. We argue that secure operating modes of hypercalls, characterized by rigorous value validations of both input parameters and internal variables at the cost of performance, may contribute towards improving the security of hypercall interfaces. The use of secure operating modes of hypercalls is a matter of prioritization — such modes are crucial, for example, for mission-critical deployments of hypervisors where security, and not performance, is of the highest importance.

B. Security Mechanisms

The research and industrial communities have designed and developed security mechanisms for detecting and/or preventing hypercall attacks targeting hypervisors, for example, intrusion detection and prevention systems (IDPSes). Such security mechanisms are

- ◦ *Collabra* — Bharadwaja et al. [25] designed a distributed anomaly-based IDPS that labels hypercall invocations as malicious or benign based on hypercall parameter values;
- ◦ *MAC/HAT* (Message Authentication Code/Hypercall Access Table) — Le [26] designed an IDPS that uses

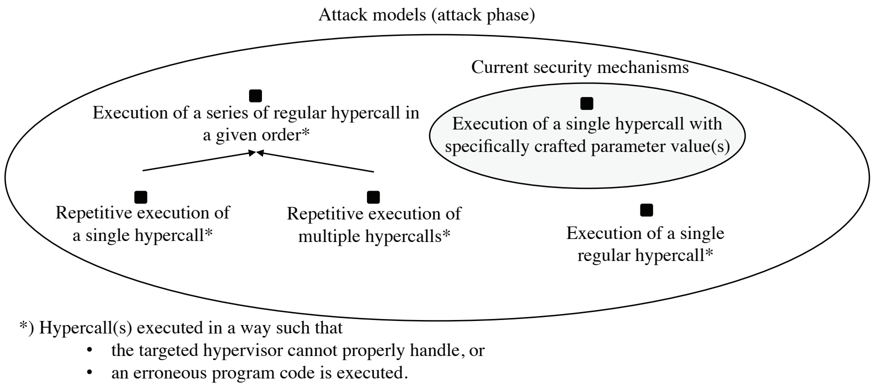
Attack models (attack phase)
Current security mechanisms
Execution of a series of regular hypercall in a given order*
Execution of a single hypercall with specifically crafted parameter value(s)
Repetitive execution of a single hypercall*
Repetitive execution of multiple hypercalls*
Execution of a single regular hypercall*
*) Hypercall(s) executed in a way such that
• the targeted hypervisor cannot properly handle, or
• an erroneous program code is executed.

Figure 2: Coverage of current security mechanisms with respect to hypercall attack models.

policies, which include hypercall parameter values and hypercall call sites, for labeling a given hypercall invocation as benign or malicious;

◦ *RandHyp* — Wang et al. [27] designed an IDPS that randomizes hypercall parameter values in order to detect and block the execution of malicious hypercall invocations that originate from untrusted locations (e.g., attacker's loadable kernel module);

◦ *XSM-FLASK* — the XSM-FLASK security module of the Xen hypervisor enables mandatory access control based on policies, which include hypercalls and hypercall parameter values [24].

Given that existing security mechanisms take into account hypercall parameter values and hypercall call sites to detect and/or prevent hypercall attacks, we argue that they are not that effective for detecting hypercall attacks that trigger vulnerabilities by executing regular hypercalls in a specific way. This especially holds for hypercall attacks that involve invoking hypercalls from usual call sites (e.g., a routine of the kernel of a guest VM) or frequently used hypercalls that are typically not subject to access control.

As we mentioned in Section III-C, all vulnerabilities due to non-implementation errors and some due to implementation errors can be triggered by executing hypercalls with regular parameter values in a specific way (see for example Figure 1(b), (d)–(f), Section III-C). Note that the vulnerabilities due to non-implementation errors make approximately 50 percent of all vulnerabilities that we analyzed (see Table II). Therefore, one may infer that current security mechanisms do not cover a significant portion of the hypercall attack space. In Figure 2, we depict the coverage of current security mechanisms with respect to the attack models (attack phase only) defined in Section III-C.

Mechanisms for securing hypercall interfaces that consider the way in which hypercalls are executed in addition to hypercall parameter values and call sites, are lacking. Being able to detect and/or prevent not only some attacks triggering vulnerabilities due to implementation errors, we argue that such security mechanisms would be effective against most of the hypercall attacks that can be seen in practice. For example, the attack depicted in Figure 1(e) can be detected if the allocation of hypercalls to vCPUs for execution is taken into account as a factor for detecting attacks. The execution of the *GNTTABOP_SETUP* and *GNTTABOP_TRANSFER* hypercalls at the same time, each at different vCPU, may be considered as an abnormal activity by an anomaly-based IDPS.

Besides the issue discussed above, we raise the issue that currently there are no approaches for generating workloads that contain representative hypercall attacks. The latter are crucial for the accurate and rigorous evaluation of IDPSes designed for detecting and preventing hypercall attacks. An inaccurate evaluation of IDPSes may lead to the deployment of misconfigured or ill-performing IDPSes in production environments, increasing the risk of security breaches. Approaches for generating workloads that contain hypercall attacks may be used for purposes beyond evaluation of IDPSes (i.e., for any cyber security experiment where controlled generation of malicious workloads is needed, such as verification of XSM-FLASK policies).

The main reason for the issue mentioned above is the lack of publicly available information on hypercall vulnerabilities and attacks, a problem that this paper is focussing on. Such information is a basis for the development of approaches for generating artificial hypercall attacks that closely resemble real ones. Bharadwaja et al. [25], Wang et al. [27], and Le [26] acknowledge the above issue and explicitly stress that it is a major problem stating, for example, *"We have some difficulties to fully evaluate the efficiency of our hypercall protection measures ... known attack codes on Xen virtualization are not available."* (Le [26], Section 5.1.2 — "Attack Experimental Issues", pg. 47). The models that we presented in Section III-C may serve as a basis for designing tools for generating activities representative of those of an attacker executing hypercall attacks. Preliminary work in this area is described in [28].

V. Summary – Lessons Learned and Recommendations

With the goal of increasing the amount of publicly available information on vulnerabilities of hypervisors' hypercall handlers (i.e., hypercall vulnerabilities) and attacks triggering them (i.e., hypercall attacks), we analyzed a set of 35 hypercall vulnerabilities. Our vulnerability analysis approach consisted of analyzing publicly available reports describing the considered vulnerabilities (e.g., CVE reports, security advisories), reverse-engineering the patches fixing the vulnerabilities, and developing proof-of-concept code, which allowed us to trigger the vulnerabilities and closely observe all stages of the life cycle of a typical hypercall attack. We made the following observations:

i) A big portion of the implementation errors causing hypercall vulnerabilities are missing value validations, both of input parameters and internal variables (i.e., variables that are created, and to which values are assigned, within a hypercall handler). Eliminating missing value validation errors by adding program code verifying values of input parameters and internal variables may reduce the execution speed of hypercalls (e.g., hypercalls of the Xen hypervisor may exhibit an increased frequency of overhead incurring hypercall continuations). The impact of variable value validations on the execution speed of the hypercalls of the Xen hypervisor has been a key factor for the use of programming practices for boosting the latter, which has led to introducing vulnerabilities.

ii) Non-implementation errors causing hypercall vulnerabilities, many of which can be triggered unintentionally as part of regular system operation, are common. As a result, given that hypervisors are often mission- and business-critical systems, we recognize the need for advances that enable the system reliability community to reduce failures to be adapted and shared with the hypervisor community for implementation.

iii) Hypercall attacks can be effective hypervisor DoS attacks. Many hypercall attacks corrupt the state of the targeted hypervisor, and some lead to information leakage, which makes them an effecitive mechanism for intruding hypervisors when executed as part of a multi-step attack. Although possible (see [3]), it is less likely that the execution of only a single hypercall attack will lead to an intrusion resulting in malicious code execution with hypervisor privilege.

iv) Attackers' activities for executing hypercall attacks can be categorized into the following attack models:

- execution of a single hypercall with regular parameter value(s) (i.e., regular hypercall), or parameter value(s) specifically crafted for triggering a given vulnerability (which is typical for triggering some vulnerabilities due to implementation errors), or
- execution of a series of regular hypercalls in a given order, including repetitive execution of a single or multiple hypercalls,

where the attack models involving execution of regular hypercalls assume that the hypercalls are executed in a way such that *a)* the targeted hypervisor cannot properly handle (which is typical for triggering vulnerabilities due to non-implementation errors); or *b)* an erroneous program code is executed (which is typical for triggering some vulnerabilities due to implementation errors).

We presented an action plan for improving the security of hypercall interfaces:

i) Tools for fuzzing hypercalls, which can be used for the convenient and time-efficient discovery of hypercall vulnerabilities, are lacking and should be developed. Such tools would significantly speed up the process of discovering hypercall vulnerabilities, especially those due to implementation errors. Hypercall fuzzing is challenging since, unlike, for example, system calls, many hypercalls perform operations that alter the state not only of the system executing them (i.e., a guest VM), but also of the underlying hypervisor (see Section IV-A1).

ii) Our study revealed that non-implementation errors causing hypercall vulnerabilities are common. As a result, we argue that methods for formally verifying the functional correctness of hypercalls that aim at discovering the non-implementation errors causing hypercall vulnerabilities (see Section III-A2) should be developed (see Section IV-A1).

iii) Secure hypercall programming practices enforcing, for example, value validations of all variables used within a given hypercall handler (i.e., input parameters and internal variables), would help to eliminate missing value validation errors. The application of such practices would lead to the development of secure hypercalls, which, however, may execute slowly. This poses the challenge of developing hypercall programming practices such that, for example, rigorous and frequent value validations are performed at a reasonable performance cost.

iv) Existing security mechanisms (e.g., IDPSes), which take into account hypercall parameter values to detect and/or prevent hypercall attacks, are not effective against hypercall attacks carried out by executing regular hypercall in a specific way (see Section III-C). Given that many of the current hypercall vulnerabilities can be triggered by executing regular hypercalls in a specific way, we argue that security mechanisms that do not only consider hypercall parameter values, but also the way in which hypercalls are executed, should be developed.

v) Approaches for generating artificial workloads that contain representative hypercall attacks should be developed since they are crucial for the accurate and rigorous evaluation of IDPSes designed for detecting and preventing hypercall attacks. The development of approaches for generating workloads that contain hypercall attacks is challenged by the lack of publicly available information on hypercall vulnerabilities and attacks, a problem that this paper deals with.

Hypercall interfaces of hypervisors are critical attack surfaces, which pose challenges that may serve as a motivation for innovative advances towards improving the security of virtualized environments.

Acknowledgment

This work is a part of the RELATE project, which is supported by the European Commission under the Seventh Framework Programme FP7 with Grant Agreement No. 264840ITN.

References

[1] F. Gens, R. Mahowald, L. R. Villars, D. Bradshaw, and C. Morris, "Cloud Computing 2010: An IDC Update," 2010.

[2] D. Perez-Botero, J. Szefer, and R. B. Lee, "Characterizing Hypervisor Vulnerabilities in Cloud Computing Servers," in *Proceedings of the 2013 International Workshop on Security in Cloud Computing*. ACM, 2013, pp. 3–10.

[3] J. Rutkowska and R. Wojtczuk, "Xen 0wning Trilogy: Part Two," Talk at Black Hat 2008. [Online]. Available: http://invisiblethingslab.com/resources/bh08/part2.pdf

[4] Multiple TMEM hypercall vulnerabilities. [Online]. Available: http://lists.xen.org/archives/html/xen-announce/2012-09/msg00006.html

[5] CVE Details. [Online]. Available: http://cvedetails.com

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. ACM, 2003, pp. 164–177.

[7] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: the Linux Virtual Machine Monitor," in *Proceedings of the Linux Symposium*, vol. 1, 2007, pp. 225–230.

[8] K. Tsipenyuk, B. Chess, and G. McGraw, "Seven pernicious kingdoms: a taxonomy of software security errors," *IEEE Security and Privacy*, vol. 3, no. 6, pp. 81–84, Nov 2005.

[9] CVE-2012-5525. [Online]. Available: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-5525

[10] CVE-2012-3495. [Online]. Available: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-3495

[11] Hypervisor Top-Level Functional Specification: Windows Server 2012. [Online]. Available: http://microsoft.com/en-us/download/details.aspx?id=39289

[12] CVE-2012-5513. [Online]. Available: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-5513

[13] CVE-2012-5510. [Online]. Available: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-5510

[14] CVE-2013-4494. [Online]. Available: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-4494

[15] CVE-2013-1964. [Online]. Available: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-1964

[16] CVE-2012-3496. [Online]. Available: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-3496

[17] S. Amin, G. Schwartz, and S. Sastry, "On the interdependence of reliability and security in Networked Control Systems," in *50th IEEE Conference on Decision and Control and European Control Conference (CDC-ECC)*, Dec 2011, pp. 4078–4083.

[18] W. Young and N. Leveson, "Systems Thinking for Safety and Security," in *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC 2013)*. ACM, 2013, pp. 1–8.

[19] CVE-2012-3494. [Online]. Available: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-3494

[20] D. Leinenbach and T. Santen, "Verifying the Microsoft Hyper-V Hypervisor with VCC," in *Proceedings of the 2nd World Congress on Formal Methods*. Springer-Verlag, 2009, pp. 806–809.

[21] E. Alkassar, M. Hillebrand, W. Paul, and E. Petrova, "Automated Verification of a Small Hypervisor," in *Verified Software: Theories, Tools, Experiments*. Springer Berlin Heidelberg, 2010, vol. 6217, pp. 40–54.

[22] G. Barthe, G. Betarte, J. Campo, and C. Luna, "Formally Verifying Isolation and Availability in an Idealized Model of Virtualization," in *FM 2011: Formal Methods*. Springer Berlin Heidelberg, 2011, vol. 6664, pp. 231–245.

[23] L. Freitas and J. McDermott, "Formal methods for security in the Xenon hypervisor," *International Journal on Software Tools for Technology Transfer*, vol. 13, no. 5, pp. 463–489, 2011.

[24] XSM-FLASK. [Online]. Available: http://wiki.xen.org/wiki/Xen_Security_Modules_:_XSM-FLASK

[25] S. Bharadwaja, W. Sun, M. Niamat, and F. Shen, "Collabra: A Xen Hypervisor Based Collaborative Intrusion Detection System," in *Proceedings of the 2011 Eighth International Conference on Information Technology: New Generations*. IEEE Computer Society, 2011, pp. 695–700.

[26] C. Hoang Le, "Protecting Xen hypercalls: Intrusion Detection/Prevention in a Virtualization Environment," Master's thesis, UBC, Vancouver, Canada, 2009.

[27] F. Wang, P. Chen, B. Mao, and L. Xie, "RandHyp: Preventing Attacks via Xen Hypercall Interface," in *Information Security and Privacy Research*. Springer Berlin Heidelberg, 2012, vol. 376, pp. 138–149.

[28] A. Milenkoski, B. D. Payne, N. Antunes, M. Vieira, and S. Kounev, "HInjector: Injecting Hypercall Attacks for Evaluating VMI-based Intrusion Detection Systems," in *Poster Reception at the 2013 Annual Computer Security Applications Conference (ACSAC 2013)*. Maryland, USA: Applied Computer Security Associates (ACSA), 2013.