Evaluation of Intrusion Detection Systems in Virtualized Environments Using Attack Injection

Aleksandar Milenkoski[1(✉)], Bryan D. Payne[2], Nuno Antunes[3], Marco Vieira[3], Samuel Kounev[1], Alberto Avritzer[4], and Matthias Luft[5]

[1] University of Würzburg, Würzburg, Germany
{milenkoski,skounev}@acm.org
[2] Netflix Inc., Los Gatos, CA, USA
bdpayne@acm.org
[3] University of Coimbra, Coimbra, Portugal
{nmsa,mvieira}@dei.uc.pt
[4] Siemens Corporation, Corporate Technology, Princeton, NJ, USA
alberto.avritzer@siemens.com
[5] Enno Rey Netzwerke GmbH, Heidelberg, Germany
mluft@ernw.de

Abstract. The evaluation of intrusion detection systems (IDSes) is an active research area with many open challenges, one of which is the generation of representative workloads that contain attacks. In this paper, we propose a novel approach for the rigorous evaluation of IDSes in virtualized environments, with a focus on IDSes designed to detect attacks leveraging or targeting the hypervisor via its hypercall interface. We present *hInjector*, a tool for generating IDS evaluation workloads by injecting such attacks during regular operation of a virtualized environment. We demonstrate the application of our approach and show its practical usefulness by evaluating a representative IDS designed to operate in virtualized environments. The virtualized environment of the industry-standard benchmark SPECvirt_sc2013 is used as a testbed, whose drivers generate workloads representative of workloads seen in production environments. This work enables for the first time the injection of attacks in virtualized environments for the purpose of generating representative IDS evaluation workloads.

Keywords: Intrusion detection systems · Virtualization · Evaluation · Attack injection

1 Introduction

Virtualization has been receiving increasing interest as a way to reduce costs through server consolidation and to enhance the flexibility of physical infrastructures. It allows the creation of virtual instances of physical devices, such as network and processing units. In a virtualized system, governed by a hypervisor, resources are shared among virtual machines (VMs).

H. Bos et al. (Eds.): RAID 2015, LNCS 9404, pp. 471–492, 2015.
DOI: 10.1007/978-3-319-26362-5_22

Although virtualization provides many benefits, it introduces new security challenges; that is, the introduction of a hypervisor introduces new threats. Hypervisors expose several attack surfaces such as device drivers, VM exit events, or hypercalls. Hypercalls are software traps from a kernel of a partially or fully paravirtualized VM to the hypervisor. They enable the execution of severe attacks. For instance, triggering a vulnerability of a hypercall handler (i.e., a hypercall vulnerability) may lead to crash of the hypervisor or to altering the hypervisor's memory (see, for example, [1,2]).

The research and industry communities have developed security mechanisms that can detect hypercall attacks. These include intrusion detection systems (IDSes), such as Xenini [3] and the de-facto standard host-based IDS OSSEC (Open Source SECurity),[1] as well as access control systems, such as XSM-FLASK (Xen Security Modules - FLux Advanced Security Kernel), which is distributed with the Xen hypervisor, and McAfee's VM protection system.[2] Under hypercall attack, we understand any malicious hypercall activity, for example, triggering a hypercall vulnerability or covert channel operations [4].

The rigorous evaluation of IDSes designed to detect hypercall attacks is crucial for preventing breaches in virtualized environments. For instance, one may compare multiple IDSes in terms of their attack detection accuracy in order to identify the optimal IDS. Workloads that contain hypercall attacks are a key requirement for evaluating the attack detection accuracy of IDSes designed to detect hypercall attacks. However, the generation of such workloads is challenging since publicly available scripts that demonstrate hypercall attacks are very rare [5,6]. An approach towards addressing this issue is attack injection, which enables the generation of representative IDS evaluation workloads. Attack injection is controlled execution of attacks during regular operation of the environment where an IDS under test is deployed. The injection of attacks is performed with respect to attack models constructed by analysing realistic attacks. Attack models are systematized activities of attackers targeting a given attack surface.

In this paper, we propose an approach for evaluating IDSes using attack injection. As part of the proposed approach, we present *hInjector*, a tool for injecting hypercall attacks. We designed hInjector to achieve the challenging goal of satisfying the key criteria for the rigorous, representative, and practically feasible evaluation of an IDS using attack injection: injection of realistic attacks, injection during regular system operation, and non-disruptive attack injection (e.g., prevention of potential crashes due to injected attacks). The approach we propose may be conceptually applied not only for evaluating IDSes designed to detect hypercall attacks, but also attacks involving the execution of operations that are functionally similar to hypercalls. Such operations are, for example, the ioctl (input/output control) calls that the KVM hypervisor supports.

Our approach uses live IDS testing, since existing IDSes designed to detect hypercall attacks perform on-line monitoring. Further, it enables the evaluation

[1] http://www.ossec.net/; OSSEC can be configured to analyze in real-time log files that contain information on executed hypercalls.

[2] http://www.google.com/patents/US8381284.

of IDSes that do and do not require training (i.e., it involves IDS training, which is needed for evaluating IDSes that require training). We demonstrate the application and practical usefulness of the approach by evaluating Xenini [3], a representative IDS designed to detect hypercall attacks. We inject realistic attacks triggering publicly disclosed hypercall vulnerabilities and specifically crafted evasive attacks. We extensively evaluate Xenini considering multiple configurations of the IDS. Such an extensive evaluation would not have been possible before due to the previously mentioned issues.

This paper is organized as follows: in Sect. 2, we provide the essential background and discuss related work; in Sect. 3, we present an approach for evaluating IDSes; in Sect. 4, we introduce the hInjector tool; in Sect. 5, we demonstrate the application of the proposed approach; in Sect. 6, we discuss future work and conclude this paper.

2 Background and Related Work

Paravirtualization and Hypercalls. Paravirtualization, an alternative to full (native) virtualization, is a virtualization mode that enables the performance-efficient virtualization of VM components based on collaboration between VMs and the hypervisor. VM components that may be paravirtualized include disk and network devices, interrupts and timers, emulated platform components (e.g., motherboards and device buses), privileged instructions, and pagetables.

With recent advances in hardware design, paravirtualizing privileged instructions and pagetables often does not provide performance benefits over full virtualization. However, paravirtualizing the other VM components mentioned above is beneficial. As a result, multiple virtualization modes have emerged, many of which involve paravirtualizing VM components of fully virtualized VMs. Hypercalls are operations that VMs use for working with paravirtualized components. They are software traps from a kernel of a VM to the underlying hypervisor.

The Hypercall Attack Surface. The hypercall interface is an attack surface that can be used for executing attacks targeting the hypervisor or breaking the boundaries set by it. This may result in unauthorized information flow between VMs or executing malicious code with hypervisor privilege (see [1,2]).

In a previous work [5], we have analyzed 35 publicly disclosed hypercall vulnerabilities and identified patterns of activities for triggering the considered vulnerabilities. We categorized the identified patterns into the following attack models: *setup phase* (optional) — execution of one or multiple regular hypercalls (i.e., hypercalls with regular parameter value(s) that may be executed during regular system operation) setting up the virtualized environment as necessary for triggering a given hypercall vulnerability; *attack phase* — execution of a single regular hypercall, or a hypercall with specifically crafted parameter value(s); or, execution of a series of regular hypercalls in a given order. In this work, we use these models for injecting hypercall attacks.

Intrusion Detection. Given the high severity of hypercall attacks, the research and industry communities have developed IDSes that can detect such attacks.

Examples are Collabra [7], Xenini [3], C^2(Covert Channel) Detector [4], Wizard [8], MAC/HAT (Mandatory Access Control/Hypercall Access Table) [6], RandHyp [9], and OSSEC. Most of these IDSes have the following characteristics in common:

- *monitoring method* and *attack detection technique* — they perform on-line (i.e., real-time) monitoring of VMs' hypercall activities and use a variety of anomaly-based attack detection techniques, which require training using benign (i.e., regular) hypercall activities;
- *architecture* — they have a module integrated into the hypervisor, intercepting invoked hypercalls and sending information relevant for intrusion detection to an analysis module deployed in a designated VM.

Current IDSes designed to detect hypercall attacks analyze the following properties of VMs' hypercall activities, which we refer to as *detection-relevant properties*: (i) hypercall identification numbers (IDs) and values of parameters of individual, or sequences of, hypercalls, and (ii) hypercall call sites (i.e., memory addresses from where hypercalls have been executed).

IDS Evaluation and Attack Injection. The accurate and rigorous evaluation of IDSes is crucial for preventing security breaches. IDS evaluation workloads that contain realistic attacks are a key requirement for such an evaluation. In Sect. 1, we stated that IDSes designed to detect hypercall attacks currently cannot be evaluated in a rigorous manner due to the lack of publicly available attack scripts that demonstrate hypercall attacks. Attack injection is a method addressing this issue, which is in the focus of this work.

To the best of our knowledge, we are the first to focus on evaluating IDSes designed to operate in virtualized environments, such as IDSes designed to detect hypercall attacks. Further, we are the first to consider the injection of hypercall attacks and of attacks targeting hypervisors in general. Pham et al. [10] and Le et al. [11] focus on injecting generic software faults directly into hypervisors. This is not suitable for evaluating IDSes — IDSes do not monitor states of hypervisors since they are not relevant for detecting attacks in a proactive manner.

Fonseca et al. [12] present an approach for evaluating network-based IDSes, which involves injection of attacks. They built Vulnerability Injector, a mechanism that injects vulnerabilities in the source code of web applications, and an Attack Injector, a mechanism that generates attacks triggering injected vulnerabilities. There are fundamental differences between our work and the work of Fonseca et al. [12], which is focussing on attack injection at application level. This includes the characteristics of the IDSes in focus, the required attack models, and the criteria for designing procedures and tools for injecting attacks.

3 Approach

Figure 1a shows our approach, which has two phases: planning and testing. The planning phase consists of: (i) specification of an IDS monitoring landscape

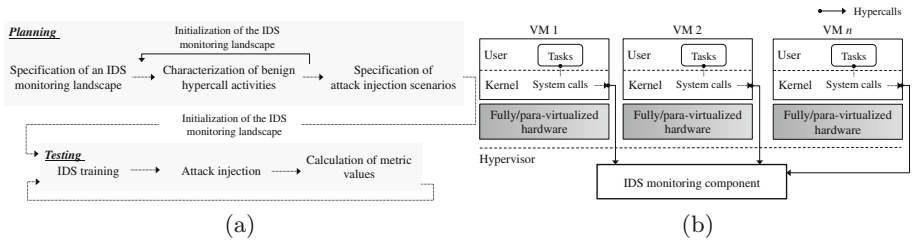


Fig. 1. (a) Approach for evaluating IDSes; (b) IDS monitoring landscape

(i.e., specifying a virtualized environment where the IDS under test is to be deployed), (ii) characterization of benign hypercall activities (i.e., making relevant observations about the benign hypercall activities), and (iii) specification of attack injection scenarios (Sect. 3.1). The testing phase consists of: (i) IDS training, (ii) attack injection, and (iii) calculation of metric values (Sect. 3.2). The activities of the testing phase are performed based on observations made in the planning phase. IDS training needs to be performed only when evaluating an IDS that requires training (i.e., an anomaly-based IDS).

3.1 Planning

Specification of an IDS Monitoring Landscape. A typical IDS designed to detect hypercall attacks monitors the hypercall activity of one or multiple VMs at the same time. VM characteristics influence the hypercall activity:

- *virtualization mode* influences which hypercalls can be executed,
- *workloads* influence which system calls can be executed, many of which map to hypercalls, and
- *system architecture and hardware* influence the VM's interface, and the type and frequency of hypercalls needed (e.g., page table update operations, which take place when page swapping occurs due to insufficient memory).

The aggregate of these characteristics across all VMs on a hypervisor is the *monitoring landscape* of an IDS designed to detect hypercall attacks. Figure 1b depicts an IDS monitoring landscape. The first activity of the planning phase of our approach is to specify an IDS monitoring landscape by defining the characteristics above for the test system. By defining workloads, we mean specifying drivers generating workloads in an automated and repeatable manner. By defining hardware, we mean allocating an amount of hardware resources to VMs that is fixed over time (i.e., disabling CPU or memory ballooning). We discuss more on the importance of specifying an IDS monitoring landscape in Sect. 3.2.

Characterization of Benign Hypercall Activities. Characterization of a VM's benign hypercall activity is crucial for answering two major questions: *How long should the IDS under test be trained*? and *What injected attacks should be*

used for the purpose of rigorous IDS testing? It consists of two parts: (i) estimation of benign hypercall activity steady-state and (ii) calculating relevant statistics. These activities are best performed when hypercall activities are captured in traces for processing off-line.

Estimation of benign hypercall activity steady-state: Steady-state of the benign hypercall activity of a VM can be understood with respect to the sum of first-time occurring variations of a detection-relevant property at a given point in time. We define S_t at time t where S_t is an increasing function such that $\lim_{t\to\infty} S_t = const$. The estimation of steady-state is crucial for determining an optimal length of the period during which an IDS under test should be trained in the testing phase (i.e., for avoiding IDS under-training).

In order to estimate steady-state, an IDS evaluator should first *initialize the IDS monitoring landscape*; that is, bring the VMs in the landscape to the state after their creation and start workloads in the VMs. Then the steady-state of the benign hypercall activities of a VM may be estimated by setting a target for the slope of a growth curve depicting S_t until a given time t_{max}. The slope of such a curve, when observed over a given period, indicates the rate of first-time occurring variations of the detection-relevant property in the period. Letting σ be a target for the slope of a growth curve over a period $t_s = t_{s2} - t_{s1}$, we have $0 <= \frac{S_{t_{s2}} - S_{t_{s1}}}{t_s} <= \sigma$. This process may be repeated multiple times for different values of t_{max} to experimentally determine σ for each VM.[3] Attacks should be injected from a VM until time t_{max}, but only after the VM's hypercall activity has reached steady-state.

The IDS under test should operate in learning mode when steady-state is estimated. This helps to create operating conditions of the overall virtualized environment, which are (almost) equivalent to those when the IDS will be trained in the testing phase. Note that an IDS may have an impact on the time needed for hypercall activities to reach steady-state due to incurred monitoring overhead.

Calculating relevant statistics: Two key statistics need to be calculated: (i) the average rate of occurrence of the detection-relevant property — this statistic should be calculated using data collected between t_{s1} and t_{max}, and (ii) the number of occurrences of each variation of the detection-relevant property — this statistic should be calculated using data collected while the system is progressing towards a steady state. These statistics help calculate metric values in the testing phase and create realistic attack injection scenarios as discussed next.

Specification of Attack Injection Scenarios. Two characteristics distinguish each attack injection scenario: *attack content* and *attack injection time*.

Attack content is the detection-relevant property of a hypercall attack in the context of a given IDS evaluation study (e.g., a specific sequence of hypercalls). Specification of attack content enables the injection of attacks that conform to representative attack models (see Sect. 2). In addition, it enables the injection of evasive attacks, for example, attacks that closely resemble common regular

[3] This raises the question whether hypercall activities are repeatable. We discuss this topic in Sect. 3.2.

activities — these attacks may be highly effective "mimicry" attacks. Crafting "mimicry" attacks is done based on knowledge on what, and how frequently, detection-relevant properties occur during regular operation of the IDS monitoring landscape (i.e., during IDS training); this is the statistic 'number of occurrences of each variation of the detection-relevant property'.

Attack injection time is the point(s) in time when a hypercall attack consisting of one or more hypercalls is injected. This allows for the specification of arbitrary temporal distributions of attack injection actions. It also allows for the specification of the following relevant temporal properties of malicious activities:

- *Base rate*: Base rate is the prior probability of an intrusion (attack). The error occurring when the attack detection accuracy of an IDS is assessed without taking the base rate into account is known as the *base rate fallacy* [13]. The specification of attack injection times provides a close estimation of the actual base rate in the testing phase. As we demonstrate in Sect. 5, base rate can be estimated by considering the number of injected attacks and the number of variations of the detection-relevant property that have occurred during attack injection. The latter is estimated based on the statistic 'average rate of occurrence of the detection-relevant property'.
- *IDS evasive properties*: Specification of the attack injection time enables the injection of "smoke screen" evasive attacks. In the context of this work, the "smoke screen" technique consists of delaying the invocation of the hypercalls comprising an attack such that a given amount of benign hypercall activity occurs between each hypercall invocation. This is an important test since some IDSes have been shown to be vulnerable to such attacks (e.g., Xenini; see [14]).

3.2 Testing

IDS Training. IDS training is the first activity of the testing phase. We require reinitialization of the IDS monitoring landscape between the planning and testing phases (see Fig. 1a). The rationale behind this is practical: many parameters of the existing IDSes designed to detect hypercall attacks (e.g., length of IDS training period, attack detection threshold) require a priori configuration. These parameters are tuned based on observations made in the planning phase (see Sect. 3.1). This raises concerns related to the non-determinism of hypercall activities, a topic that we discuss in paragraph 'on repeatability concerns'.

Attack Injection. For this critical step, we developed a new tool called *hInjector*. Section 4 introduces this tool and describes how it is used.

Calculation of Metric Values. After attack injection is performed, values of relevant metrics can be calculated (e.g., true and false positive rate). This also raises concerns related to the non-determinism of hypercall activities, which we discuss next.

On Repeatability Concerns. Observations and decisions made in the planning phase might be irrelevant if hypercall activities are highly non-deterministic

and therefore not repeatable. For example, the benign hypercall activities occurring in the testing phase may not reach steady-state at a point in time close to the estimated one in the planning phase.

In addition, metric values reported as end-results of an evaluation study, where workloads that are not fully deterministic are used, have to be statistically accurate. This is crucial for credible evaluation. Principles of statistical theory impose metric values to be repeatedly calculated and their means to be reported as end-results. Therefore, we require repeated execution of the testing phase (see Fig. 1a). However, this may be time-consuming if the number of needed repetitions is high due to high non-determinism of hypercall activities.

Specifying an IDS monitoring landscape as we define it (see Sect. 3.1) alleviates the above concerns; that is, it helps to reduce the non-determinism of hypercall activities by removing major sources of non-determinism, such as non-repeatable workloads. This is in line with Burtsev [15], who observes that, given repeatability of execution of VMs' user tasks is preserved, VMs always invoke the same hypercalls. We acknowledge that achieving complete repeatability of hypercall activities by specifying VM characteristics is infeasible. This is mainly due to the complexity of the architectures and operating principles of kernels.

In Sect. 5, we empirically show that, provided an IDS monitoring landscape is specified, a VM's hypercall activities exhibit repeatability to an extent sufficient to conclude that: (i) the decisions and observations made in the planning phase are of practical relevance when it comes to IDS testing, and (ii) the number of measurement repetitions needed to calculate statistically accurate metric values is small. This is in favor of the practical feasibility of our approach, which involves repeated initialization of an IDS monitoring landscape.

4 hInjector

hInjector is a tool for injecting hypercall attacks. It realizes the attack injection scenarios specified in the planning phase (see Sect. 3.1). The current implementation of hInjector is for the Xen hypervisor, but the techniques are not Xen-specific and can be ported to other hypervisors.

hInjector supports the injection of attacks crafted with respect to the attack models that we developed (see Sect. 2). We extend these attack models with a model involving different hypercall call sites. Hypercall call sites are one of the detection-relevant properties that existing IDSes designed to detect hypercall attacks analyze. We consider that hypercalls can be executed from *regular* or *irregular* call sites. The latter is typically a hacker's loadable kernel module (LKM) used to mount hypercall attacks.

Our design criteria for hInjector are *injection of realistic attacks*, *injection during regular system operation*, and *non-disruptive attack injection*. These criteria are crucial for the representative, rigorous, and practically feasible IDS evaluation. We discuss more in Sect. 4.2.

Availability. hInjector is publicly available at https://github.com/hinj/hInj.

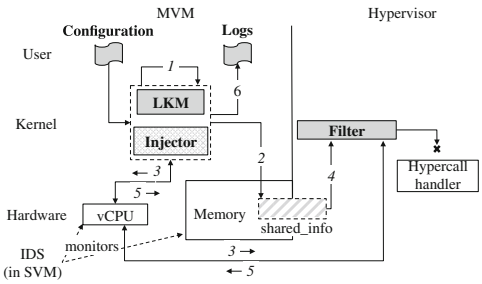


Fig. 2. The architecture of hInjector

4.1 hInjector Architecture

Figure 2 depicts the architecture of hInjector. It shows the primary components: *Injector*, *LKM*, *Filter*, *Configuration*, and *Logs*. We refer to the VM from where hypercall attacks are injected as the malicious VM (MVM). We also depict a typical IDS designed to detect hypercall attacks, with components in the hypervisor and a secured VM (SVM), co-located with MVM (see Sect. 2). The IDS monitors the MVM's hypercall activity by monitoring virtual CPU registers and the virtual memory of MVM using its hypervisor component.

The *Injector* component, deployed in the MVM's kernel, intercepts at a given rate hypercalls invoked by the kernel and modifies hypercall parameter values on-the-fly (i) making them specifically crafted for triggering a vulnerability, or (ii) replacing them with random, irregular values that an IDS may label as anomalous. The Injector injects hypercalls invoked from a regular call site (i.e., from the kernel address space). We discuss more on Injector in Sect. 4.3.

The *LKM* component, a module in MVM's kernel, invokes hypercalls with regular or specifically crafted parameter value(s), including a series of hypercalls in a given order. The LKM injects hypercalls invoked from an irregular call site (i.e., from a loadable kernel module).

The *Filter* component, deployed in the hypervisor's hypercall handlers, identifies hypercalls injected by the Injector or the LKM, blocks the execution of the respective hypercall handlers, and returns valid error codes. The Filter identifies injected hypercalls based on information stored by the Injector/LKM in the *shared_info* structure, a memory region shared between a VM and the hypervisor. To this end, we extended *shared_info* with a string field named *hid* (hypercall identification), which contains identification information on injected hypercalls. We discuss more about the Filter when we discuss the design criterion 'non-disruptive attack injection' in Sect. 4.2.

The *Configuration* component is a set of user files in XML containing configuration parameters for managing the operation of the Injector and the LKM. It allows specifying, for example, parameter values for a given hypercall (relevant to the Injector and the LKM), ordering of a series of hypercalls (relevant to the LKM), and temporal distribution of injection actions.

The *Logs* are user files containing records about invoked hypercalls that are part of attacks; that is, hypercall IDs and parameter values, as well as timestamps. The logged data serves as reference data (i.e., as "ground truth") used for distinguishing false positives from injected attacks and calculating IDS attack detection accuracy metrics, such as true and false positive rate.

We now present an example of the implemented hypercall attack injection procedure. Figure 2 depicts the steps to inject a hypercall attack by the LKM: *(1)* the LKM crafts a parameter value of a given hypercall as specified in the configuration; *(2)* the LKM stores the ID of the hypercall, the number of the crafted parameter, and the parameter value in *hid*; *(3)* the LKM passes the hypercall to MVM's vCPU, which then passes control to hypervisor; *(4)* the Filter, using the data stored in *hid*, identifies the injected hypercall when the respective hypercall handler is executed; *(5)* the Filter updates *hid* indicating that it has intercepted the injected hypercall, then returns a valid error code to block execution of the handler; *(6)* after the error code arrives at MVM's kernel, the LKM first verifies whether *hid* has been updated by the Filter and then logs the ID and parameter values of the injected hypercall.

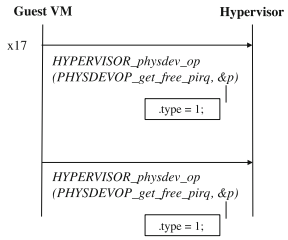
4.2 hInjector Design Criteria

Injection of Realistic Attacks. The injection of realistic attacks is crucial for the representative IDS evaluation. In order to inject realistic hypercall attacks, hInjector requires representative hypercall attack models. hInjector supports the injection of attacks crafted with respect to arbitrary attack models, for example, the models that we developed [5] (see Sect. 2).

We developed proof-of-concept code for triggering the hypercall vulnerabilities that we analyzed [5].[4] The proof-of-concept code enables granularization of the attack models. For example, we can specify specific parameter values or the order of a series of hypercalls that trigger a hypercall vulnerability. This enables the injection of realistic hypercall attacks, crafted to trigger publicly disclosed hypercall vulnerabilities. In Fig. 3a, we show how we triggered the vulnerability CVE-2012-3495 of the Xen hypervisor in a testbed environment. In Fig. 3b, we present the configuration of hInjector for injecting an attack triggering CVE-2012-3495. Configuration files for injecting attacks that trigger publicly disclosed hypercall vulnerabilities are distributed with hInjector.

Injection During Regular System Operation. Benign activities, mixed with attacks, are needed to subject an IDS under test to realistic attack scenarios. hInjector is designed to inject hypercall attacks *during* regular operation of guest VMs. Thus, provided that during an IDS evaluation experiment representative user tasks run in the VMs in the IDS monitoring landscape, the presence of representative benign hypercall activities is guaranteed.

[4] We developed proof-of-concept code based on reverse-engineering the released patches fixing the considered vulnerabilities.



(a)

```
<scenario rate="1" repeat="1">
 <hcall name="physdev_op" repeat="18">
  <parameter number="1">
   <value type="int">23</value>
  </parameter>
  <parameter number="2">
   <struct name="physdev_get_free_pirq">
    <member name="type">
     <value type="int">1</value>
    </member>
   </struct>
  </parameter>
 </hcall>
</scenario>
```

(b)

Fig. 3. (a) Triggering CVE-2012-3495 [the hypercall *physdev_op* is executed 18 times: the value of its first parameter is 23 (*PHYSDEVOP_get_free_pirq*); the value of the field *type* of its second parameter (struct *physdev_get_free_pirq*) is 1]; (b) Configuration of hInjector for injecting an attack triggering CVE-2012-3495

Non-disruptive Attack Injection. The state of the hypervisor or the VM(s) from where attacks are injected may be altered by the attacks injected by hInjector. This may cause crashes obstructing the execution of the IDS evaluation process. Filter prevents crashes by blocking the execution of the hypervisor's handlers that handle the injected hypercalls. This preserves the states of the hypervisor and of the VM(s) from where attacks are injected, and, in addition, it ensures that injected attacks do not impact the operation of the IDS under test, which normally has components in the hypervisor and in a VM (see Sect. 2). After blocking the execution of hypervisor's handlers, Filter returns valid error codes. This allows the control flow of the kernel of the VM from where hypercall attacks are injected to properly handle failed hypercalls that have been executed by it and have been modified by the Injector on-the-fly.

4.3 Injector: Performance Overhead

The rate at which the kernel invokes hypercalls is high (i.e., in some cases more than 30000 hypercalls per second, see Sect. 5). Therefore, Injector, which manipulates hypercalls on-the-fly, can easily incur intolerable system performance overhead. We made the following observation when developing Injector: manipulating orders of series of hypercalls is very performance-expensive; therefore, Injector can manipulate only hypercall parameter values. Further, we measured the overhead incurred by Injector on the execution rate of hypercalls, relative to this rate when Injector is inactive, when replacing regular hypercall parameter values with random, irregular values. In Fig. 4, we depict this overhead, which we measured as follows. We deployed Injector in the kernel of a Debian 8.0 operating system running on top of Xen 4.4.5. We invoked the *mmuext_op* hypercall 40000 times using a loadable kernel module. We measured the time, in microseconds (μs), needed for the invoked hypercalls to complete their operation ('Execution time' in Fig. 4) in scenarios where: (i) Injector is inactive ('Base' in Fig. 4), and (ii) Injector manipulates the value of the second parameter of *mmuext_op* at the

rate of 1:50 (i.e., Injector manipulates parameter value once in 50 invocations of *mmuext_op*), 1:100, 1:500, 1:1000, and 1:10000. We repeated the measurements 30 times and averaged the results.

Based on the results from the above experiment, we conclude that a user should constrain the rate at which Injector manipulates hypercall parameter values to a value such that the incurred overhead is not higher than 2 %. This is important since we observed that overheads higher than 2 % often cause noticeable system slowdowns or crashes. We showed that Injector normally incurs overheads higher than 2 % when it manipulates hypercall parameter values approximately once in less than 500 hypercall invocations (see Fig. 4). Note that overheads incurred by Injector for hypercalls other than *mmuext_op* do not significantly differ from those depicted in Fig. 4 since the implementation of Injector is the same for all hypercalls.

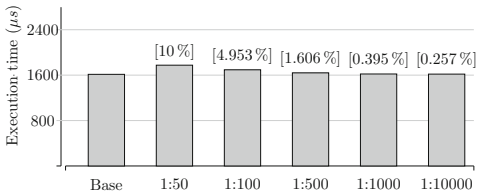


Fig. 4. Overhead incurred by Injector [measurements of the incurred overhead are depicted in square brackets]

5 Case Study

We now demonstrate the application of our approach by evaluating Xenini [3] following the steps presented in Sect. 3. Xenini is a representative anomaly-based IDS. It uses the popular Stide [16] method. Xenini slides a window of size k over a sequence of n hypercalls and identifies mismatches (anomalies) by comparing each k-length sequence with regular patterns learned during IDS training. Xenini records the number of mismatches as a percentage of the total possible number of pairwise mismatches for a sequence of n hypercalls (i.e., $(k-1)(n-k/2)$). We call this percentage *anomaly score*. When the anomaly score exceeds a given threshold $th \in [0;1]$, Xenini fires an alert. For the purpose of this study, we configured Xenini such that its detection-relevant property is sequences of hypercall IDs of length 4 (i.e., $k = 4$; $n = 10$).

It is important to emphasize that we focus on demonstrating the feasibility of attack injection in virtualized environments for IDS testing purposes and not on discussing the behavior of Xenini in detail or comparing it with other IDSes. We specify arbitrary attack injection scenarios and evaluate Xenini with the sole purpose of demonstrating all steps and functionalities of the proposed approach. We refer the reader to Sect. 5.3 for an overview of further application scenarios.

5.1 Case Study: Planning

Specification of an IDS Monitoring Landscape. We use the SPECvirt_sc2013 benchmark to specify an IDS monitoring landscape.[5] SPECvirt_sc2013 is an industry-standard virtualization benchmark developed by SPEC (Standard Performance Evaluation Corporation). Its complex architecture matches a typical server consolidation scenario in a datacenter — it consists of 6 co-located front- and back-end server VMs (i.e., web, network file, mail, batch, application, and database server VM) and 4 workload drivers that act as clients generating workloads for the front-end servers. The workload drivers are heavily modified versions of the drivers of the SPECweb 2005, SPECimap, SPECjAppServer2004, and SPECbatch (i.e., SPEC CPU 2006) benchmarks. They generate workloads representative of workloads seen in production virtualized environments.

In Fig. 5, we depict the deployment of SPECvirt_sc2013 as an IDS monitoring landscape. The workload drivers generate workloads that map to hypercalls. We used Xen 4.4.1 as hypervisor and we virtualized the VMs using full paravirtualization.[6] To each server VM, we allocated 8 virtual CPUs pinned to separate physical CPU cores of 2 GHz, 3 GB of main memory, and 100 GB of hard disk memory. In Fig. 5, we depict the operating systems and architectures of the server VMs, and the server software we deployed in the VMs.[7]

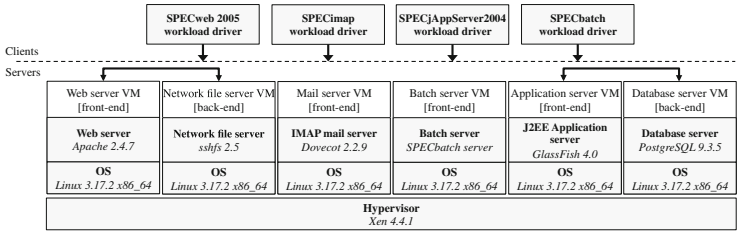


Fig. 5. SPECvirt_sc2013 as an IDS monitoring landscape [IMAP stands for Internet Message Access Protocol; J2EE stands for Java 2 Enterprise Edition]

Characterization of Benign Hypercall Activities. We now estimate steady-states of the benign hypercall activities of the server VMs and calculate the relevant statistics (see Sect. 3.1). We initialized the IDS monitoring landscape and deployed Xenini before the characterization. We used *xentrace*, the tracing facility of the Xen hypervisor, to capture hypercall activities in trace files.

[5] http://www.spec.org/virt_sc2013/.

[6] We did not use any other virtualization mode because of a technical limitation; that is, the *xentrace* tool, which we use to capture benign hypercall activities in files for processing off-line, currently supports only full paravirtualization. However, support for other virtualization modes is currently being implemented.

[7] An overview of the software and hardware requirements for deploying and running SPECvirt_sc2013 is available at https://www.spec.org/virt_sc2013/docs/SPECvirt_UserGuide.html.

Table 1. Benign workload characterization

	Run 1		Run 2	
Server VM	t_s (sec.)	r (occ./sec.)	t_s (sec.)	r (occ./sec.)
Web	5350	19644.5	5357	19627.3
Network file	5343	10204.9	5360	10231.3
Mail	5391	3141.5	5382	3148.7
Batch	5315	633.4	5330	623.8
Application	5367	31415.9	5377	31437.5
Database	5285	27294.9	5273	27292.3

Figure 6 a–f show growth curves depicting S_t until time $t_{max} = 5500$ s for each server VM (see the curves entitled 'Run 1'). We set the target σ to 15 over a time period of 100 s for the slope of each growth curve. In Table 1, column 'Run 1', we present t_s (in seconds – *sec.*), which is the time at which the VMs' hypercall activities reach steady-state. We also present r (in number of occurrences per second – *occ./sec.*), which is the average rate of occurrence of the detection-relevant property. We also calculated the statistic 'number of occurrences of each variation of the detection-relevant property' (not presented in Table 1), which we use to craft "mimicry" attacks (see Sect. 5.2).

We now empirically show that, provided an IDS monitoring landscape is specified, VMs' hypercall activities exhibit repeatability in terms of the characteristics

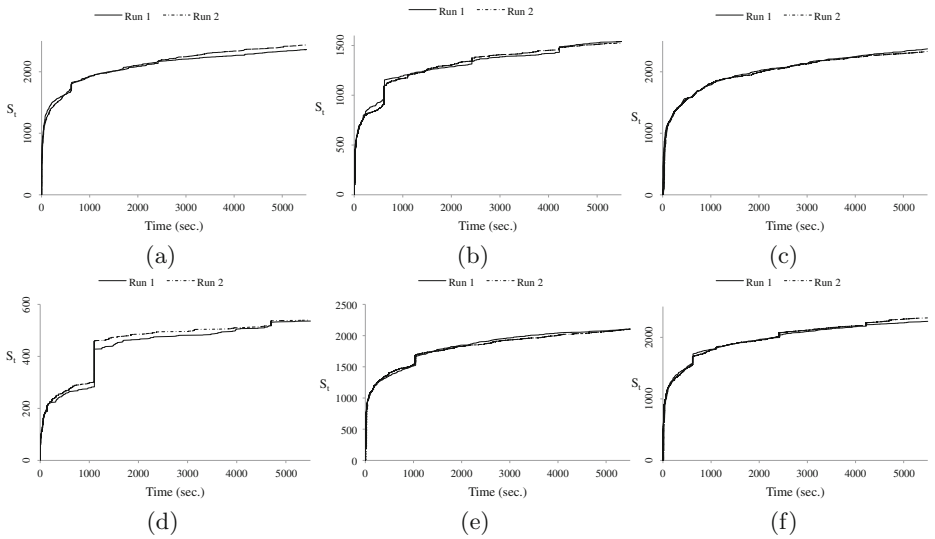


Fig. 6. Growth curves: (a) web (b) network file (c) mail (d) batch (e) application (f) database server VM.

of interest to an extent sufficient for accurate IDS testing (see Sect. 3.2). We performed the above characterization campaign twice and compared the results. In Fig. 6 a–f, we depict the obtained growth curves (see the curves entitled 'Run 1' and 'Run 2'). These curves are very similar, which indicates that the characteristics of the VMs' hypercall activities of interest are also similar. In Table 1, we present t_s and r for each server VM (see column 'Run 1' and 'Run 2'). We observe a maximum difference of only 17 *sec.* for t_s and 26.4 *occ./sec.* for r. We repeated this process over 30 times and calculated maximum standard deviation of only 8.036 for t_s and 15.95 for r. These small deviations indicate that benign hypercall activities exhibit non-repeatability to such a small extent that it has no significant impact on metric values, which we repeatedly calculate for statistical accuracy (see Sect. 3.2).

Specification of Attack Injection Scenarios. We now specify attack injection scenarios that we will realize in separate testing phases. We focus on injecting attacks triggering publicly disclosed hypercall vulnerabilities. However, the injection of any malicious hypercall activity using hInjector is possible (e.g., covert channel operations as described in [4]), in which case an IDS evaluation study would be performed following the same process we demonstrate here.

***Scenario #1*:** We will first evaluate the attack coverage of Xenini when configured such that $th = 0.3$. We will evaluate Xenini's ability to detect attacks triggering the vulnerabilities CVE-2012-5525, CVE-2012-3495, CVE-2012-5513, CVE-2012-5510, CVE-2013-4494, and CVE-2013-1964. We thus demonstrate injecting realistic attacks that conform to the attack models that we constructed [5]. We will inject attacks from the web and mail server VM using the LKM component of hInjector.

Attack contents: In Fig. 7 (a)–(e), we depict the contents of the considered attacks (the content of the attack triggering CVE-2012-3495 is depicted in Fig. 3a; we will inject this attack from the web server VM). The semantics of these figures is the same as that of Fig. 3a — we depict the hypercalls executed as part of an attack and relevant hypercall parameters; that is, integer parameters defining the semantics of the executed hypercalls (e.g., *XENMEM_exchange*), and, where applicable, parameters with values specifically crafted for triggering a vulnerability, which are marked in bold.

Attack injection times: After the hypercall activities of both the web and mail server VM have reached a steady state, we will inject the considered attacks, with 10 s of separation between each attack, and, where applicable, with no delays between the invocation of the hypercalls comprising an attack.

***Scenario #2*:** We will investigate the accuracy of Xenini at detecting the attacks considered in *Scenario #1*, however, modified such that they have IDS evasive characteristics (i.e., they are "mimicry" and "smoke-screen" attacks). We will inject from the database server VM, using the LKM component of hInjector, both the unmodified attacks that consist of multiple hypercalls (i.e., we exclude the attack triggering CVE-2012-5525) and their modified counterparts as part of three separate testing phases. Therefore, we will observe how successful the modified attacks are at evading Xenini.

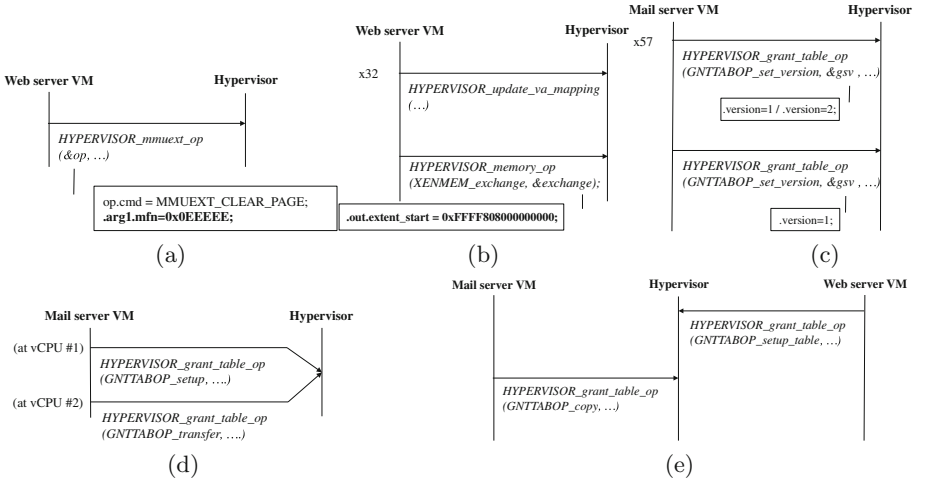


Fig. 7. Injecting attacks that trigger: (a) CVE-2012-5525; (b) CVE-2012-5513; (c) CVE-2012-5510; (d) CVE-2013-4494 [invoking hypercalls from two virtual CPUs (vCPUs)]; (e) CVE-2013-1964 [this vulnerability can also be triggered by invoking hypercalls from one VM]

Attack contents: The contents of the unmodified attacks and the "smoke-screen" attacks we will inject are depicted in Figs. 3a and 7 (b)–(e). To craft "mimicry" attacks, we place each individual hypercall that is part of an attack in the middle of a sequence of 20 injected hypercalls (i.e., at position 10). We built this sequence by starting with the most common detection-relevant property we observed in the planning phase — *iret*, *iret*, *iret*, *iret*. We then added 16 hypercalls such that sliding a window of size 4 over the sequence provides common detection-relevant properties seen during IDS training (i.e., while the hypercall activity of the database server VM has been progressing towards a steady state); we were able to perform this because we calculated the statistic 'number of occurrences of each variation of the detection-relevant property' (see Sect. 3.1). Therefore, we obscure attack patterns making them similar to regular patterns. For example, in Fig. 8a, we depict the content of the "mimicry" attack triggering CVE-2013-1964.

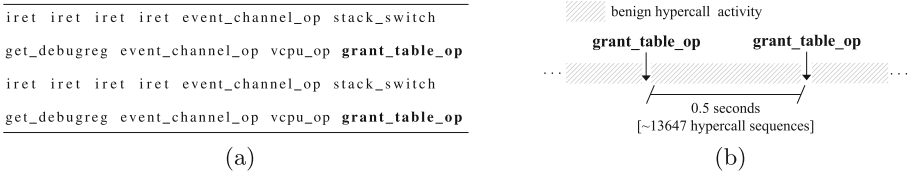


Fig. 8. Injecting IDS evasive attacks triggering CVE-2013-1964: (a) "mimicry" attack; (b) "smoke screen" attack [the hypercalls triggering CVE-2013-1964 are marked in bold]

Attack injection times: We craft "smoke screen" attacks by specifying attack injection times (see Sect. 3.1). We will inject a "smoke screen" attack by delaying for 0.5 s the invocation of the hypercalls comprising the attack. Since the average rate of occurrence of the detection-relevant property for the database server VM is 27294.9 occ./sec. (see Table 1, column 'Run 1'), we obscure attack patterns by making Xenini analyze approximately 13647 benign occurences of the detection-relevant property before encountering a hypercall that is part of an attack. For example, in Fig. 8b, we depict the "smoke screen" attack triggering CVE-2013-1964.

After the hypercall activities of the database server VM have reached a steady state, we begin three separate attack injection campaigns: unmodified attacks, "mimicry" attacks, and "smoke screen" attacks. Each campaign injects 6 attacks, with 10 s of separation between each attack.

5.2 Case Study: Testing

We now test Xenini with respect to the scenarios presented in Sect. 5.1.

Scenario #1

IDS Training. We deployed and configured Xenini and hInjector. We initalized the IDS monitoring landscape and we trained Xenini until time $t_s = 5391$ s. This is the time period needed for the hypercall activities of both the web and mail server VM to reach steady-state (see Table 1, column 'Run 1').

Attack Injection and Calculation of Metric Values. We injected the considered attacks over a period of $t_{max} - ts = 109$ s and then calculated metric values, that is, true and false positive rate. These are calculated as ratios between the number of true, or of false, alerts issued by Xenini, and the total number of injected attacks, or of benign variations of the detection-relevant property occuring during attack injection, respectively. We estimate the latter based on the statistic 'average rate of occurrence of the detection-relevant property'. We repeated the testing phase only 3 times in order to calculate statistically accurate metric values with a relative precision of 2 % and 95 % confidence level.[8]

Performing repeated measurements is important for calculating a statistically accurate value of the false positive rate. This is because the number of issued false alerts and the total number of benign variations of the detection-relevant property occuring during attack injection vary between measurements due to the non-determinism of benign hypercall activities. We observed that the true positive rate normally does not vary, since the number and properties of injected attacks (i.e., the attacks' contents and attack injection times) are fixed.

[8] In addition, we repeated the testing phase over 30 times observing that the obtained metric values negligibly differ from those we present here. This is primarily because of the high repeatability of hypercall activities and it indicates that only a small number of repetitions is needed to calculate statistically accurate metric values.

In Table 2, we present Xenini's attack detection score. It can be concluded that Xenini exhibited a true positive rate of 0.5 when configured such that $th = 0.3$. We now consider multiple IDS operating points (i.e., IDS configurations which yield given values of the false and true positive rate). In Fig. 9, we depict a ROC (Receiver Operating Characteristic) curve, which plots operating points for different values of th. We executed separate testing phases to quantify the false and true positive rate exhibited by Xenini for each value of th. We quantified these rates by comparing the output of Xenini with the "ground truth" information recorded by hInjector. We considered the total number of true and false alerts issued by Xenini (i.e., 6 and 6), injected attacks, and occurences of the detection-relevant property during attack injection, originating from both the web and mail server VM. The results depicted in Fig. 9 match the expected behavior of Xenini (i.e., the lesser the value of th, the more sensitive the IDS, which results in higher true and false positive rates; see [3]). This shows the practical usefulness of our approach.

Table 2. Detection score of Xenini [✓: detected/x: not detected, $th = 0.3$]

Targeted vulnerability (CVE ID)	Detected
CVE-2012-3495	✓
CVE-2012-5525	x
CVE-2012-5513	✓
CVE-2012-5510	✓
CVE-2013-4494	x
CVE-2013-1964	x

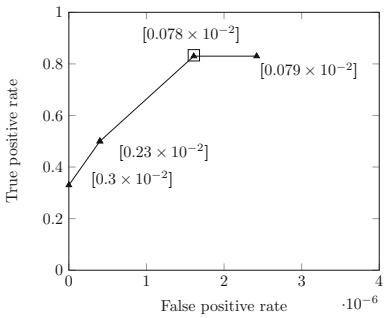


Fig. 9. Attack detection accuracy of Xenini [$th = 0.1$: $(2.42 \times 10^{-6}; 0.83)$ • $th = 0.2$: $(1.61 \times 10^{-6}; 0.83)$ • $th = 0.3/th = 0.4$: $(0.4 \times 10^{-6}, 0.5)$ • th=0.5: (0, 0.33) • □ marks the optimal operating point]

We now calculate values of the 'expected cost' metric (C_{exp}) developed by Gaffney and Ulvila [17], which expresses the impact of the base rate (see Sect. 3.1). This metric combines ROC curve analysis with cost estimation by associating an estimated cost with each IDS operating point. The measure of cost is relevant in scenarios where a response that may be costly is taken when an IDS issues an alert. Gaffney and Ulvila introduce a cost ratio $C = C_\beta/C_\alpha$, where C_α is the cost of an alert when an intrusion has not occured, and C_β is the cost of not detecting an intrusion when it has occurred. To calculate values of C_{exp}, we set C to 10 (i.e., the cost of not responding to an attack is 10 times higher than the cost of responding to a false alert; see [17]).

We estimate the base rate as follows. We have injected 6 attacks consisting of 115 hypercalls over 109 s. Further, the average rate of occurence of the detection relevant property originating from the web and mail server VM during attack injection is estimated at $19644.5 + 3141.5 = 22786$ occ./sec. (see Table 1, column 'Run 1'). Therefore, the base rate is $\frac{115}{(22786 \times 109 + 3)} = 0.5 \times 10^{-4}$.

We calculated the actual base rate by calculating the actual average rate of occurence of the detection relevant property during attack injection. We observed that the difference between the actual and estimated base rate is negligible and has no impact on values of C_{exp}. This is primarily because the difference between the actual and estimated value of the average rate of occurence of the detection relevant property is small. Further, the ratio between the number of injected attacks and the number of occurences of the detection-relevant property during attack injection is very low due to the typical high value of the latter. This indicates the practical relevance of the planning phase.

In Fig. 9, we depict in square brackets values of C_{exp} associated with each IDS operating point. The 'expected cost' metric enables the identification of an optimal IDS operating point. An IDS operating point is considered optimal if it has the lowest C_{exp} associated with it compared to the other operating points. We mark in Fig. 9 the optimal operating point of Xenini.

Scenario #2

IDS Training. We deployed and configured Xenini and hInjector. We initalized the IDS monitoring landscape and, since we will inject attacks from the database server VM, we trained Xenini over a period of 5285 s.

Attack Injection and Calculation of Metric Values. We injected the unmodified, the "mimicry", and the "smoke screen" attacks as part of three separate testing phases. In Table 3, we present the anomaly scores reported by Xenini for the injected attacks. We thus quantify the success of the "mimicry" and "smoke screen" attacks at evading Xenini. Their evasive capabilities are especially evident in the case of the attacks triggering CVE-2012-3495 and CVE-2012-5510. That is, these attacks, when unmodified, can be very easily detected by Xenini (see the high anomaly scores of 1.0 in Table 3). However, when transformed into "mimicry" attacks, the detection of these attacks is significantly challenging (see the low anomaly scores of 0.17 and 0.14 in Table 3).

Table 3. Anomaly scores for the injected non-evasive and evasive attacks

Targeted vulnerability (CVE ID)	Anomaly scores		
	Unmodified	"Mimicry"	"Smoke screen"
CVE-2012-3495	1.0	0.17	0.25
CVE-2012-5513	0.32	0.107	0.28
CVE-2012-5510	1.0	0.14	0.31
CVE-2013-4494	0.21	0.14	0.14
CVE-2013-1964	0.25	0.14	0.14

The results presented in Table 3 match the expected behavior of Xenini when subjected to evasive attacks (i.e., Xenini reports lower anomaly scores for the evasive attacks than for the unmodified attacks; see [14]). This shows the practical usefulness of our approach and the relevance of the observations made in the planning phase, which we used to craft evasive attacks.

5.3 Further Application Scenarios

Besides evaluating typical anomaly-based IDSes, such as Xenini, our approach, or hInjector in particular, can be used for:

- ***evaluating hypercall access control (AC) systems*** — an example of such a system is XSM-FLASK. By evaluating AC systems, we mean verifying AC policies for correctness. This is performed by first executing hypercalls whose execution in hypervisor context should be prohibited and then verifying whether their execution has indeed been prohibited. hInjector can greatly simplify this process since it allows for executing arbitrary hypercall activities and recording relevant information (e.g., information on whether invoked hypercalls have been executed in hypervisor context, see Sect. 4.1);
- ***evaluating whitelisting IDSes*** — by whitelisting IDS, we mean IDS that fires an alarm when it observes an activity that has not been whitelisted, either by an user or by the IDS itself while being trained. For example, OSSEC can be configured to whitelist the hypercall activities it observes during training — our approach involves both rigorous IDS training and execution of arbitrary hypercall activities (see Sect. 3); RandHyp [9] and MAC/HAT [6] detect and block the execution of hypercall invocations that originate from untrusted locations (e.g., a loadable kernel module) — hInjector supports the injection of hypercall attacks both from the kernel and a kernel module (see Sect. 4.1).

6 Conclusion and Future Work

We presented an approach for the live evaluation of IDSes in virtualized environments using attack injection. We presented *hInjector*, a tool for generating IDS

evaluation workloads that contain virtualization-specific attacks (i.e., attacks leveraging or targeting the hypervisor via its hypercall interface — hypercall attacks). Such workloads are currently not available, which significantly hinders IDS evaluation efforts. We designed hInjector with respect to three main criteria: injection of realistic attacks, injection during regular system operation, and non-disruptive attack injection. These criteria are crucial for the representative, rigorous, and practically feasible evaluation of IDSes. We demonstrated the application of our approach and showed its practical usefulness by evaluating a representative IDS designed to detect hypercall attacks. We used hInjector to inject attacks that trigger real vulnerabilities as well as IDS evasive attacks.

Our work can be continued in several directions:

- We plan to explore the integration of VM replay mechanisms (e.g., XenTT [15]) in our approach. This may help to further alleviate concerns related to the repeatability of VMs' hypercall activities;
- We intend to establish a continuous effort on analyzing publicly disclosed hypercall vulnerabilities in order to regularly update hInjector's attack library (see Sect. 4.2). This is an important contribution since the lack of up-to-date workloads is a major issue in the field of IDS evaluation;
- We plan to extensively evaluate a variety of security mechanisms (see Sect. 5.3) and work on applying our approach for injecting attacks involving operations that are functionally similar to hypercalls, such as KVM ioctl calls.

We stress that robust IDS evaluation techniques are essential not only to evaluate specific IDSes, but also as a driver of innovation in the field of intrusion detection by enabling the identification of issues and the improvement of existing intrusion detection techniques and systems.

Acknowledgments. This research has been supported by the Research Group of the Standard Performance Evaluation Corporation (SPEC; http://www.spec.org, http://research.spec.org).

References

1. Rutkowska, J., Wojtczuk, R.: Xen Owning Trilogy: Part Two. http://invisiblethingslab.com/resources/bh08/part2.pdf
2. Wilhelm, F., Luft, M., Rey, E.: Compromise-as-a-Service. https://www.ernw.de/download/ERNW_HITBAMS14_HyperV_fwilhelm_mluft_erey.pdf
3. Maiero, C., Miculan, M.: Unobservable intrusion detection based on call traces in paravirtualized systems. In: Proceedings of the International Conference on Security and Cryptography (2011)
4. Wu, J.Z., Ding, L., Wu, Y., Min-Allah, N., Khan, S.U., Wang, Y.: C^2Detector: a covert channel detection framework in cloud computing. Secur. Commun. Netw. **7**(3), 544–557 (2014)
5. Milenkoski, A., Payne, B.D., Antunes, N., Vieira, M., Kounev, S.: Experience report: an analysis of hypercall handler vulnerabilities. In: Proceedings of the 25th IEEE International Symposium on Software Reliability Engineering. IEEE (2014)

6. Le, C.H.: Protecting Xen Hypercalls. Master's thesis, UBC (2009)
7. Bharadwaja, S., Sun, W., Niamat, M., Shen, F.: A Xen hypervisor based collaborative intrusion detection system. In: Proceedings of the 8th International Conference on Information Technology, pp. 695–700. IEEE (2011)
8. Srivastava, A., Singh, K., Giffin, J.: Secure observation of kernel behavior (2008). http://hdl.handle.net/1853/25464
9. Wang, F., Chen, P., Mao, B., Xie, L.: RandHyp: preventing attacks via Xen hypercall interface. In: Gritzalis, D., Furnell, S., Theoharidou, M. (eds.) SEC 2012. IFIP AICT, vol. 376, pp. 138–149. Springer, Heidelberg (2012)
10. Pham, C., Chen, D., Kalbarczyk, Z., Iyer, R.: CloudVal: a framework for validation of virtualization environment in cloud infrastructure. In: Proceedings of DSN 2011, pp. 189–196 (2011)
11. Le, M., Gallagher, A., Tamir, Y.: Challenges and opportunities with fault injection in virtualized systems. In: VPACT (2008)
12. Fonseca, J., Vieira, M., Madeira, H.: Evaluation of web security mechanisms using vulnerability and attack injection. IEEE Trans. Dependable Secure Comput. **11**(5), 440–453 (2014)
13. Axelsson, S.: The base-rate fallacy and its implications for the difficulty of intrusion detection. ACM Trans. Inf. Syst. Secur. **3**(3), 186–205 (2000)
14. Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In: Proceedings of the 9th ACM Conference on Computer and Communications Security, pp. 255–264 (2002)
15. Burtsev, A.: Deterministic systems analysis. Ph.D. thesis, University of Utah (2013)
16. Forrest, S., Hofmeyr, S., Somayaji, A., Longstaff, T.: A sense of self for Unix processes. In: IEEE Symposium on Security and Privacy, pp. 120–128, May 1996
17. Gaffney, J.E., Ulvila, J.W.: Evaluation of intrusion detectors: a decision theory approach. In: Proceedings of the 2001 IEEE Symposium on Security and Privacy, pp. 50–61 (2001)