Intrusion Injection for Virtualized Systems: Concepts and Approach

Charles F. Gonçalves*†, Nuno Antunes*, Marco Vieira*
*University of Coimbra, CISUC, DEI, Portugal - charles@dei.uc.pt, nmsa@dei.uc.pt, mvieira@dei.uc.pt
†Board of Information Technology, CEFET-MG, Brazil - charles@cefetmg.br

***Abstract*—Virtualization is drawing attention due to countless benefits, leaving Hypervisors with the paramount responsibility for performance, dependability, and security. However, while there are consolidated approaches to assessing the performance and dependability of virtualized systems, solutions to assess security are very limited. Key difficulties are evaluating the system in the presence of unknown attacks and vulnerabilities and comparing the security attributes of different systems and configurations when an intrusion occurs. In this paper, *we propose a novel concept and approach of intrusion injection for virtualized environments*, which consists of directly driving the system into the erroneous states that mimic the ones resulting from actual intrusions (in the same way errors are injected to mimic the effects of residual faults). We present a prototype capable of injecting erroneous states related to memory-corruption in the Xen Hypervisor to show that the concept and approach proposed here are feasible. The prototype is evaluated using publicly disclosed exploits across three different versions of Xen. Results show that our tool can inject erroneous states equivalent to those resulting from attacks that exploit existing vulnerabilities, even on versions where those vulnerabilities do not exist.**

***Index Terms*—security, intrusions, virtualization, hypervisor, security assessment**

I. Introduction

The adoption and success of cloud computing have proven the importance of virtualization technologies. The main component in a virtualization system is the Hypervisor [1] that is responsible for multiplexing all physical resources to the related virtual ones. Virtualization is expanding its applicability from enterprise servers and cloud computing to new realms [2]–[5]. This expansion translates into bigger and more complex codebases and larger attack surfaces, which increase the risk of suffering attacks. However, approaches to assess security in such environments are limited.

It is known that software systems have defects [6] and any security mechanism can fail [7]. Also, the complexity of attacks has been increasing even if the expertise of the average attacker is declining [8] (e.g., ready-to-use scripts available online for non-expert attackers). The current dilemma is not *how* but *when* a vulnerability will be discovered and, eventually, exploited. It is thus of vital importance to understand how complex systems, such as hypervisors, deal with successful intrusions, even if one is not currently aware of vulnerabilities in the system or of attacks that may be performed against it.

In the last decades, solutions have been proposed to assess and benchmark dependability attributes of complex systems (e.g., robustness testing, fault injection, reliability modeling) [9]–[14]. A concrete example is the use of fault injection for evaluating error detection mechanisms [15] and for enabling dependability benchmarks [16]. However, assessing the security of virtualized systems (and of complex systems in general) is a recent demand for which we lack consolidated and practical solutions. In fact, existing work on security assessment has largely focused on techniques for the evaluation and comparison of security tools. A classic case is the use of vulnerability and attack injection to create exploitable code that is then used to evaluate the effectiveness of vulnerability detection tools and intrusion detection systems [17]–[19].

Empirically assessing security by relying on real or injected vulnerabilities is fallacious for several reasons. First, it is impossible to predict which vulnerabilities or attacks are yet to be discovered. Also, most vulnerabilities are fixed before a stable release of the software [20] and, even if well documented, the task of injecting a payload capable of exploiting a known vulnerability can be really hard [21], especially in low-level systems like hypervisors. Furthermore, when a vulnerability is found in a released software, it should be quickly fixed [22] losing meaning for evaluation purposes. Even assuming knowledge about existing vulnerabilities or that representative vulnerabilities and attacks can be injected, designing test cases that cover all vulnerability classes is challenging, especially because software and systems evolve through time, limiting the creation of an attack corpus for testing purposes.

This paper proposes the novel concept and approach of **intrusion injection** for virtualized environments. The core idea is to inject erroneous states (e.g., overwrite an unauthorized memory) that mimic the ones resulting from successful actual intrusions. The reasoning is that in the same way, fault injection can be used to evaluate error detection mechanisms, our approach can be used for assessing how a system behaves in the presence of the injected erroneous states and, consequently, how it would handle intrusions resulting from attacks that take advantage of (potentially unknown) vulnerabilities.

As the same erroneous state may result from different attacks targeting different vulnerabilities, which can lead to the same (or similar) security violation, focusing on the injection of erroneous states increases assessment coverage while decreasing the testing complexity. In fact, we believe that every specific mechanism that needs to be compromised to attack a system can be abstracted, moving the focus to the effects of intrusions that impact system security.

Several key capabilities of intrusion injection, when compared with previous approaches [18], [23]–[25], should be

highlighted: *i)* it is easier to induce a representative erroneous state than effectively attack the system, *ii)* our solution enables testing when a corpus of known exploits/vulnerabilities is not available, *iii)* it enables studying the impact of erroneous states of (potentially unknown) vulnerabilities, *iv)* it allows covering different types of vulnerabilities using a single injection interface, increasing testing coverage, and *v)* it enables portable test cases based on architectural conceptual aspects of the target systems rather than on implementation-dependent ones.

To check the feasibility of the concept and illustrate our approach we implemented a proof-of-concept injector based on Xen [26]. This tool provides an interface with the guest OS to inject memory-corruption erroneous states directly into the hypervisor, allowing a guest to arbitrary access memory under the hypervisor address space. This way, we can change the system behavior to possibly induce security violations like code execution by writing code into the instruction pointer address or map memory pages belonging to different guests by altering the page tables controlled by the hypervisor.

We show that intrusion injection is viable by reproducing the effects of four public exploits that abuse memory-management operations [27]. In practice, we first run real exploits in a Xen-vulnerable version (4.6) and in other versions where the vulnerabilities were already fixed (4.8 and 4.13). With this, we were able to crash the hypervisor (by overwriting a descriptor table handler) and escalate privileges to the privileged domain (by acquiring root privileges on it) on the vulnerable version but not on the other two. Afterwards, using our prototype, we attempted and succeeded in injecting the same erroneous states in the three versions of the hypervisor. For versions 4.6 and 4.8, the injection of the erroneous states leads to the same security violations for every exploit. However, the erroneous states injected do not lead to security violations for two of the four cases in version 4.13, which shows a difference in how it handles those states, suggesting that Xen has been improved and that intrusion injection can be used for assessment.

The rest of the paper is as follows. The next section discusses the related work and the basic differences with our approach. §III explains the basic concepts. §IV presents the intrusion injection approach while §V the prototype for Xen. The capability for reproducing erroneous states, the applicability in non-vulnerable systems, and the implications on security assessment are discussed in §VI, §VII and §VIII, respectively. §IX discusses strengths and limitations. Conclusions and future work are presented in §X.

II. Related Work

Fuzz testing (or Fuzzing) is a set of techniques that deliver inputs that the program may not expect to discover bugs [28]. It has been vastly employed to discover security flaws in software applications [29], [30], OS kernels [31], database systems [32], and many other domains [28]. One limitation of fuzzing is the difficulty of guaranteeing that every searchable space is covered, leaving the possibility of missing critical vulnerabilities. Unlike fuzzing, our technique is not focused on the vulnerability finding problem. Instead, it abstracts the hard part of finding a viable path to a vulnerability and focuses on reproducing a representative erroneous state like if the system was under exploitation.

Many works on formal methods have been employed to analyze and verify the security aspects and properties of hypervisors [33]–[37]. Microverification was used to prove the security properties of a modified KVM core components [36]. A bound model checker was used to evaluate the reachability of security vulnerabilities and to speed-up security test generation in the Xen Hypervisor [33]. However, despite its usefulness in a limited context, the modeling and validation provided by formal methods cannot automatically handle large code bases of commodity hypervisors [33], [36], and it still unknown how to fully verify its security properties.

Fault injection has been utilized in combination with system virtualization to assess non-virtualized systems [38], [39], as well as to evaluate the virtualized systems themselves [40]–[42]. Upon initial examination, our approach may bear some similarities to fault injection, but it is conceptually distinct in several crucial ways. The most significant difference is that we capture aspects inherent to the system's threats by targeting states that are specifically relevant to security. Additionally, we model the advantage an attacker may gain from the system and focus on enabling security-related assessments.

Previous works have adapted the fault injection concept to the security domain, proposing techniques and tools for the injection of vulnerabilities [19], [23] and attacks [18]. While attack injection can be used for vulnerability detection, vulnerability injection (together with attacks crafted for the injected vulnerabilities) can be used to evaluate security tools, such as intrusion detection systems and vulnerability detectors, but not for assessing the security of alternative systems or configurations [43], as is the case of our injection approach.

Studies of attacks and vulnerabilities in virtualized systems can be found in [44]–[47]. Many of the challenges identified in those works can benefit from our intrusion injection concepts and approach. For example, intrusion injection can be use to reveal system components that can not handle intrusions properly, leading to critical security violations. This can be applied to support risk assessment and techniques to minimize the attack surface. In fact, known vulnerabilities can be used for different goals, among others we can cite assessing trustworthiness [43], vulnerability prediction [48], and modeling future vulnerabilities [49], but not for comparing the capability of alternative systems to handle intrusions. Also, none of such works addresses the problem of the impact of potential vulnerabilities (yet to be discovered) like we do.

A tool for injecting hypercall attacks to evaluate the effectiveness of security mechanisms is proposed on [50]. Similar work has been done by [51], but the authors claim that their approach is less intrusive. The behavior of Xen in the presence of soft errors was characterized by [52]. As a new alternative for security assessment, intrusion injection can open new possibilities for researchers to evaluate different aspects of virtualized systems and systems that rely on them.

The main challenge of evaluating the security of computer

systems is to create a malicious faultload (a.k.a. attackload). Exploit databases are usually scarce of readily available scripts [53] and manually creating such exploits is a difficult challenge. Efforts to gather and distribute exploits that enable security researchers to use a common dataset that enabled cross-validation of techniques (e.g., [54]) are not available or evolve for commercial support limiting the community [55]. We argue that intrusion injection is the right mean to overcome these challenges and enable security assessment of virtualization systems and of systems and applications that rely on them.

III. Erroneous States and Intrusion Injection

The definitions of fault, error, and failure have been established in the dependability community a long time ago [56]. A *failure* occurs when the service delivered deviates from fulfilling the system's goal. An *error* is a perturbation of the system state which is prone to lead to a failure. The cause for an error is called a *fault*, which can be active or latent. An active fault leads to an error, otherwise, the fault is latent. These concepts were later extended to the AVI (Attack, Vulnerability, Intrusion) composite fault model, to help understand the mechanisms behind security attacks [18].

A. From Errors to Erroneous States

The AVI model is a specialization of the *fault* $\rightarrow$ *error* $\rightarrow$ *failure* in the context of malicious faults. *Attacks* are the intentional acts that the adversary takes to subvert the system (i.e. a malicious external fault [56]) and can have many steps (e.g, authentication bypass, code execution, etc.) usually using exploits to reach its goal. An exploit is a component, usually a software script, that interacts with the target system activating a software vulnerability (e.g, a sequence of hypercalls done by a malicious guest machine that reaches a vulnerable code). A *vulnerability* is a fault in the system introduced during design (e.g., wrong requirement), development (e.g., a software bug), or operation (e.g., an incorrect configuration). When an exploit activates a vulnerability, it causes an *intrusion* [18].

The first effect of an intrusion is an **erroneous state** (e.g., return address overwritten or memory corrupted). If nothing is done to correct or process this erroneous state, a *security violation* (i.e., a failure that affects a security attribute) may occur. Note that, although the erroneous state has the potential to violate a security property, the adversary does not benefit from this if the system is able to handle the error.

Fig. 1 shows the relation between the chain of dependability threats and the extended-AVI attack model, which supports our concept of intrusion injection. The term *erroneous states* is used instead of *errors* to emphasize that these are intrusion-induced errors. The erroneous state can have one or many forms depending on the vulnerability and how it is exploited. For instance, a buffer overflow can be exploited in many forms depending on the size and content corrupted in the memory. The results can be a crash, a privilege escalation, or even have no impact on the system.

Let us take the XSA-133 [27] (a.k.a VENOM) as an example to clarify the concept further. This vulnerability is

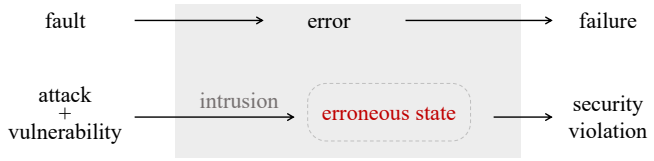

Fig. 1. Chain of dependability threats [56] with the extended-AVI model [18].

a fault in the floppy disk controller (FDC) of the QEMU hypervisor (also affecting KVM, XEN, and VirtualBox), which does not correctly restricts the operations on its input. To attack the system, a malicious user, with privileged access, can create crafted kernel modules to send an input buffer larger than specified to the FDC. When the attack succeeds, an internal buffer of the FDC overflows, and the hypervisor enters an erroneous state where memory that should be inaccessible is corrupted. If there is no mechanism to handle this erroneous state, security violations like a privilege escalation may happen.

B. The Concept of Intrusion Injection

Fault tolerance mechanisms are deployed with the goal of avoiding failures, e.g., stopping error propagation. Fault injection is often used to study the capability of a system to handle residual faults. However, fault injection is extremely complex, not only because fault injection tools are complex, but also because it is difficult to drive the system into a state in which the injected fault is activated. This normally results in very low fault activation rates [57]. A common approach is to directly inject errors instead of root faults. An example is the use of Software Implemented Fault Injection to inject hardware faults: what is indeed injected is an error (a change in the system stated, e.g., a bit-flip) and not the fault that leads to that error [58]. In fact, it is easier to flip a bit in memory by means of software than to expose the hardware to e.g. radiation until a bit flips. The reasoning is that, by injecting errors (the effects of faults), one can still assess and validate the fault tolerance abilities of systems [59], [60].

Our **intrusion injection concept** is based on the idea that, in the same way errors can be injected to emulate the effects of potential faults, they can be injected to mimic the effects of attacks on potential vulnerabilities. Indeed, by injecting erroneous states, one can study how the system deals with those states as if attacks were crafted to exploit known vulnerabilities. Considering the VENOM example, the intrusion injection tool could change the QEMU process to allow the injection of the corresponding error, e.g., by overwriting the FDC request handler method. Thus, when an IO request similar to an attack on VENOM is sent to FDC, memory corruption could happen in QEMU in a similar way to what is done when exploiting the vulnerable code.

An aspect worth of attention is *reachability*, which is related to whether a particular system state can be reached through a specific sequence of actions or events [61]. In our case, if one particular erroneous state can be reached from a given initial state using intrusion injection. In fact, there are several challenges involved in injecting erroneous states. Firstly, it is crucial to differentiate erroneous states (security-related) from those that accidental faults can originate. Secondly,

it is important to understand the erroneous states that are reachable using intrusion injection but not by exploiting any type of known vulnerability (although still potentially useful to assess unexpected/unknown situations, they should be carefully used). Third, the technical feasibility of injecting certain erroneous states must be considered, e.g., when dealing with hardware-enabled vulnerabilities that may not be practically injectable. Lastly, there are erroneous states that may be currently unknown and that will be discovered only in the future in the context of new incidents. Although adequate modeling of real (known) intrusions may address the first three challenges, the last one requires continuous modeling of new knowledge on vulnerabilities and intrusions.

Similarly to fault injection that requires fault models [14], [62], [63], intrusion injection also asks for **intrusion models** (IMs) that define the main aspects of the injection. In other terms, we need to define **the essential characteristics that can be generalized from a collection of exploits to related systems** (i.e., systems with comparable architectures and high-level characteristics, towards which similar attacks are performed). This is crucial for achieving representativeness; otherwise, one may be assessing something that does have security implications to the system. Although a detailed definition of IMs is beyond the scope of this work, in Section IV-B we will elaborate more about it.

It is important to stress that Intrusion Injection **does not assist in finding new vulnerabilities**, and it does not help to assess the reachability of vulnerable code, nor how probable the activation of such areas is. Other techniques can handle those challenges, like fuzzing [31], model checkers [33], among others. Nevertheless, inevitably, there will always be exploits to reach vulnerabilities. Therefore, studying the activation of those erroneous states, how to protect systems from being compromised, or even how to recover are crucial aspects that we aim to support.

Finally, we mitigate the limitation of not knowing the specific path to the vulnerability by abstracting any relevant factor before the intrusion. We capture the main aspects like the vulnerability and the attack type by modeling the intrusion into our technique (see Section IV-B). Still, any unknown vulnerability or attack that leads to similar intrusions, i.e., the same intrusion model, can have its impact assessed using our intrusion injection approach.

C. Potential Applicability

Intrusion injection is a technique that may help system vendors and administrators check if an erroneous state (or a group/set of erroneous states) is detectable, understandable, interpreted, and considered by the system as undesired behavior. In fact, many examples can be given as potential areas of applicability, like evaluating defense mechanisms, assessing the impact of intrusions in systems deployed in virtualized systems, or even "porting erroneous states" from different systems/vendors.

Intrusion injection can be used as an enabler to evaluate a security mechanism that focuses on specific components of the virtualized system. For example, an hazardous type of attack is one where the adversary can write values to an arbitrary location (usually referred to as write-what-where conditions, CWE-123 [64]). As an example of those threats, we can cite the vulnerabilities that enable the attacker to have write access to the page table in a hypervisor, e.g., XSA-212, XSA-302. Assuming a deployed mechanism to prevent unauthorized modification of page tables, the effectiveness of this mechanism can be tested using our approach. For this, we need to model different intrusions that target unauthorized page-table changes and execute a testing campaign injecting various erroneous states using an intrusion injector.

To exemplify the assessment of systems running on top of virtualized systems, we can cite a transactional business-critical system that runs on a public cloud. How can one assess the impact of successful intrusions on the hypervisor in the ability of the transactional system to ensure the ACID (Atomicity, Consistency, Isolation, Durability) properties? Unfortunately, it is impossible to have sufficient coverage by creating attack campaigns based on known exploitable vulnerabilities. Intrusion injection helps mitigate this limitation by enabling the ability to induce erroneous states similar to the ones observed in real hypervisor vulnerabilities.

Injecting the same erroneous states in hypervisors from different vendors or in different versions of a given vendor can also be helpful for evaluation purposes. For example, imagine that cloud provider *X* wants to evaluate how its virtualized environment that uses hypervisor *A* would be affected by a vulnerability similar to one discovered in an hypervisor *B*. This can be achieved by injecting erroneous states from vulnerabilities in B using an intrusion injector in A. Risk assessment for security hardening is also a good example. One can create a campaign to exercise different system components to check which one is more vulnerable to the effects of vulnerabilities in case they exist in those or other components. With the results of this assessment, a hardening strategy can be put together to mitigate impacts and increase resilience. In practice, we have a clear improvement from the engineering perspective since an intrusion injector would reduce the practical barriers of devising different evaluation campaigns.

IV. Injecting Intrusions in Virtualized Systems

This section presents our novel approach to support the study of how systems deal with potential intrusions effects.

A. The Intrusion Injection Approach

Fig. 2 shows an overview of the approach and its key concepts. The top half of the figure represents a **traditional scenario**, where an intrusion (i.e., an exploit that effectively reaches a vulnerability) induces an erroneous state in the system. Once the erroneous state is reached, the system may experience a security violation or may handle the erroneous state (as previously discussed). On the bottom half, we can see an overview of our **approach functioning** and how it simplifies the process, following the red dotted arrows.

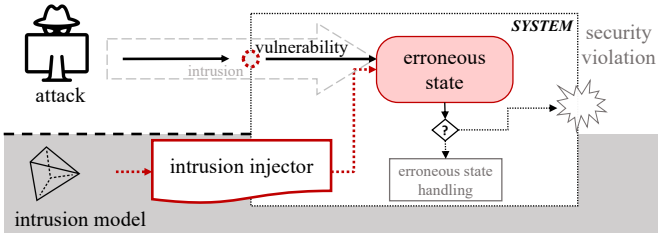

Fig. 2. Overview of the methodology key components.

The process is guided by an **intrusion model** that abstracts the main aspects of a set of intrusions and determines what erroneous state should be activated. The definition and selection of the IMs are further discussed in Section IV-B.

The **intrusion injector** is the component that injects the **erroneous state** into the hypervisor (based on the IM), thus reproducing the effects of a hypothetical intrusion in the system. Several alternatives may exist to implement such an injector. For example, it can be an existing system configuration or functionality used in a non-conforming manner or a specific component implemented for that end. In practice, several implementations of this component may be needed, as different erroneous states may require different injection approaches and locations. In Section V, we present a prototype of an intrusion injector for memory-related erroneous states.

The final step is to inspect the system behavior to understand the potential consequences of the erroneous state injected. As a security violation may happen or not, depending on the capacity of the system to deal with intrusions, system monitoring is needed to evaluate how the system behaves in the presence of the erroneous state. System monitoring is extensively addressed in the literature [65]–[69] and it is also not the focus of this work.

B. Intrusion Models for Virtualized Systems

Although new vulnerabilities are discovered and new attacks are performed annually against computer systems, it is known that the erroneous states caused by the consequent intrusions are usually similar to others observed for past vulnerabilities and attacks [45], [47], [70]–[72]. In other words, existing works show that different vulnerabilities can lead to similar erroneous states. For example, the work in [73] shows how the strategy to exploit the XSA-148 vulnerability is used to exploit the XSA-182 vulnerability [27], leading to the same erroneous state and security violation. This way, an Intrusion Model may be representative of a set of (known and unknown) vulnerabilities and attacks that might lead to the same erroneous state.

Intrusion models for virtualized systems must be closely related to the attacks that target such systems, and should generalize a set of intrusions into a single definition. Thus, the main concepts of *attack* and *erroneous state* should be present to characterize the intrusion model and represent the essential aspects of such a process.

We base our concept of intrusion model on the definitions of exploit and weird machines presented in [74]–[76]. Such works approach definitions and models of vulnerability exploitation. They show that exploiting a vulnerability can be seen as a particular form of programming using only interactions with the system. Let us consider a simple case to illustrate the reasoning. A program is built by combining instructions that use computational resources (e.g., memory, CPU, devices) to process input data and deliver the intended service. This concrete implementation can be abstracted as a state machine that transitions between states when processing instructions. When attacking this program, an adversary interacts with it, trying different inputs to see if one activates a vulnerability and put the system into an erroneous state (or *weird states* as in [74]) that enables the attack to advance. When this step succeeds, the attacker discloses an unintended functionality (e.g., write unauthorized memory) that takes the system from a given state and places it directly into an erroneous state (e.g., malicious return address on the stack).

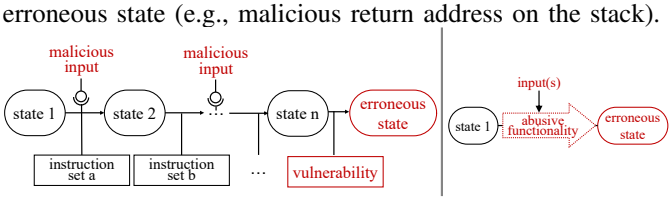

Fig. 3. A diagram of the intrusion internal impact on a system (left) and its abstraction with core aspects of an intrusion model (right)

The diagram in the left part of Fig. 3 represents the internal transitions of a system when an intrusion occurs. The system is ready to receive inputs in state 1 (initial state). Eventually, malicious inputs arrive via the interface to state 1 which process the *instruction set a* and transitions to state 2. The system keeps processing instructions, possibly receiving new inputs, and changing states until the point in which the vulnerability is activated, leading to an erroneous state. The vulnerability activation has been abstracted to a simple transition on this state machine, but its actual materialization can be complex depending on many factors [71].

From an external point of view, this set of steps represents the execution of an unintended functionality that the system was built with, which we call **abusive functionality**. The diagram on the right part of Fig. 3 represents a conceptual model of an intrusion from an external point of view (attacker perspective). Both diagrams are equivalent in functionality, i.e., putting the system into a specific erroneous state based on a given input (or set of inputs). Note that the right part represents all concepts that model an intrusion: the initial state where the system processes the input, the adversary-system interaction (i.e., execution of the abusive functionality), and the erroneous state induced.

Based on the previous explanation, an **Intrusion Model** (IM) *abstracts how an erroneous state is achieved when using an abusive functionality through a given interface.* To illustrate the concepts, let us discuss two Xen vulnerabilities. The XSA-387 is a bug on *Grant table v2* status pages that should be released to Xen when a guest switches from *grant table v2* to *v1*. The XSA-393 is a bug on the code that removes a page mapping from a guest that can be activated when the `XENMEM_decrease_reservation` hypercall is issued after

a cache maintenance instruction. In both vulnerabilities, a malicious guest can retain access to Xen pages even after they are used for other purposes (e.g., assigned to other VMs).

In both cases, the erroneous state is the page reference held by the VM to which the attacker has access. However, the specifics of the erroneous states are different (i.e., the memory region and page type). Conceptually, both allow an adversary to keep access to a memory page after releasing it to the hypervisor. Consequently, the abusive functionality achieved by the attacker is of the type *Keep Page Reference*.

Note that the abusive functionality is an abstraction for limiting the injections to the ones that are security relevant (i.e., erroneous states). For the previous example, there are many ways to corrupt a page reference, but few leave a page reference bounded to a domain after a hypercall invocation. Despite a powerful abstraction, it is very challenging to model this example properly as, in a realistic workflow, only some pages reference can be kept after an hypercall execution. Furthermore, some states like this may not even be reachable during actual execution. Thus, the abusive functionality should be cautiously modeled, taking *reachability* into account.

C. Instantiating and Using Intrusion Models

The previous definition of IM is generic and should be instantiated depending on the target virtualized system and the evaluation objectives. If the evaluation includes several different threat models (e.g., untrusted `dom0` or malicious kernel), we may need to instantiate several IMs, each one defining a triggering source (e.g., a privileged user in a guest), a target component (e.g., memory management component), and the interaction interface (e.g., hypercall).

Let us illustrate this instantiation. In XSA-148, an attacker adds the PSE flag to an L2 page table entry to acquire a writable page-table reference. In XSA-182, the code to validate the pre-existing L4 page tables was faulty and allows the adversary to acquire a writable page table reference. The erroneous state is a guest-writable page table, although the specifics are different (i.e., which page level and where the reference is mapped), but the ability to acquire a guest-writable page-table entry (abusive functionality) is leveraged from those vulnerabilities. Two possible instantiations of the IM could be: *i)* an unprivileged guest abusing *hypercalls* to acquire *guest-writable page table entries* to access hypervisor pages, and *ii)* a privileged guest (`dom0`) abusing *hypercalls* to acquire *guest-writable page table entries* to access pages of guest VMs.

Deciding on the concrete intrusion models to consider is tightly related to the objectives of the evaluation. For instance, IMs related to memory access should be used if the main goal is to assess integrity or confidentiality properties. On the other hand, IMs targeting event handlers may be adequate if the focus is availability. The detailed presentation of IMs and their instantiation, supported by an extensive field study, is out of the scope of this paper.

The actual erroneous states injected into the system directly correlate with the Intrusion Model instantiation. Effectively, the erroneous states will be materialized by the inputs provided to the abusive functionality (see the right part of Fig. 3). Additionally, they will be conditioned by the triggering source in the virtualized system, e.g., originated in a device driver. Relevant erroneous states can be difficult to be designed by a tester. Thus, complex techniques might be required to fulfill such a task. One possibility is to randomize inputs to an injector, creating an approach that resembles fuzzing testing but in another level of interaction, in a post-attack phase. Another option would be using methods like bounded model checking [33] to define possible inputs from intrusion injectors. We do not address this problem and future work should investigate the best approach for such a complex task.

D. Preliminary Abusive Functionality Study

One important assumption that we make in the IM definition is the existence of a set of **abusive functionalities** capable of generalizing the outcome of different classes of intrusions. Although we can not yet fully ensure that this statement is true, a preliminary ad-hoc evaluation of possible abusive functionalities over Xen vulnerabilities shows evidence that this might be possible. In fact, in an initial investigation, we focused only on memory-related vulnerabilities. We randomly selected 100 CVEs from the Xen Security Advisory list to define possible abusive functionalities that attackers might acquire by exploiting those vulnerabilities.

We individually assessed each vulnerability by going through all related metadata for some context. By related metadata, we mean all online information that can be gathered for the vulnerability, i.e., the advisory report, NVD and CVE database data, patches and their comments, source code, official mailing list, etc. Sometimes it is easy and clear to define the advantage that an adversary can acquire from a vulnerability. For instance, *Read Unauthorized Memory* can be achieved when some uninitiated field or padding data is leaked by a parameter in a hypercall. Conversely, some definitions can be unclear or difficult to define precisely since limited information is provided in the metadata. This is the case for the abusive functionalities *Fail a Memory Access* and *Fail a Memory Mapping*. The information is too unspecific, and we can only infer that somehow the operation fails, but we cannot state anything more, for instance, if an error happens, if it is a silent failing, etc.

Table I shows our preliminary classification. Interestingly, during the classification process, we noted that classes of functionalities that could be used for grouping them according to their primary goal (memory access, memory management, and exceptional conditions) started materializing. Also, even restricting the analysis to memory-related vulnerabilities, abusive functionalities affecting other components of the system can be identified, such as generating uncontrolled interrupts or causing the CPU to hang. Note that the total amount of functionalities classified is greater than 100 (see Table I). This is because some CVEs can have more than one abusive functionality depending on how they are exploited, e.g., CVE-2019-17343, CVE-2020-27672. Functionalities of the Exceptional Conditions class are the ones that trigger exception

mechanisms in the system either on hardware (e.g., accessing unaligned memory) or explicitly on software as an abort/assert mechanism (e.g., kernel panic, bug statements). They often represent a defensive mechanism, such as a default switch condition with a FATAL directive that crashes the system because it is "impossible" to reach such a state.

An extended study to cover all vulnerabilities on Xen is planned for future work. We want to study in detail known vulnerabilities and their abusive functionalities to properly understand what are the possible set of erroneous states that we may inject and which IMs we can abstract from them.

TABLE I
EXAMPLE OF **ABUSIVE FUNCTIONALITIES** THAT CAN BE OBTAINED FROM ACTIVATING XEN VULNERABILITIES.

Memory Access – 35 CVEs	
Read Unauthorized Memory	11
Write Unauthorized Memory	09
Write Unauthorized Arbitrary Memory	02
R/W Unauthorized Memory	12
Fail a Memory Access	01
Memory Management – 40 CVEs	
Corrupt Virtual Memory Mapping	04
Corrupt a Page Reference	04
Decrease Page Mapping Availability	01
Guest-Writable Page Table Entry	07
Fail a memory mapping	02
Uncontrolled Memory Allocation	11
Keep Page Access	11
Exceptional Conditions – 11 CVEs	
Induce a Fatal Exception	06
Induce a Memory Exception	05
Non-Memory Related – 22 CVEs	
Induce a Hang State	20
Uncontrolled Arbitrary Interrupts Requests	02

V. A PROTOTYPE INJECTOR FOR UNAUTHORIZED MEMORY ACCESSES IN THE XEN HYPERVISOR

The intrusion injector is a key component to make our methodology possible. This section presents the implementation of a prototype injector that is able to inject erroneous states resulting from arbitrary unauthorized memory accesses in a Xen Hypervisor. A prevalence of memory access vulnerabilities can be observed in the body of virtualization vulnerabilities published in the National Vulnerability Database [77] and also in related literature [45], [47], [72]. Free access to any memory location can be used to reproduce erroneous states that can be caused by exploiting many different vulnerabilities, which may impact all security attributes.

Although our injector was implemented to work in Xen [26], an open-source and widely adopted hypervisor in the industry and academia, this does not limit the applicability of the approach to it. In fact, implementation in other hypervisors is just a technical aspect.

A. Xen Memory Management

The hypervisor is responsible for multiplexing all physical resources between virtual machines, and it does that by using virtualization-assisted instructions (on hardware), intercepting operations, or using paravirtualization (PV) [26]. PV is a lightweight virtualization technique that does not require virtualization extensions, providing a software layer that uses hypercalls to behave similarly to the hardware. Hypercalls correspond to system calls in a virtualization context and allow guests to invoke privileged operations in the hypervisor [26], [78]. The idea behind our injector is to expose hypercalls that allow similar operations without the restriction mechanism that ensures a secure execution. For the time being, our tool supports memory-related operations under paravirtualzation.

All hypervisors use an additional layer of abstraction to virtualize memory. In Xen, this abstraction is the pseudo-physical addresses mapped to the physical address through the Physical to Machine (P2M) mapping [79]. In Xen PV the abstraction is done using a technique called Direct Paging [79], where guests write page-table entries directly to physical addresses using hypercalls, i.e., any page-table change must be handled and validated by the hypervisor. Additionally, the memory layout of Xen has segmented areas with different access permission levels by definition. These segments define how virtual address ranges can be used by the guests and the hypervisor, e.g, the range `0xffff800000000000 - 0xffff807fffffffff` is read-only for guest domains. These rules and definitions are checked and must be enforced by the hypervisor. Any error in this memory layout implementation directly affects the system security.

B. Injector Implementation

To induce *erroneous states* in a virtualization system, the injector must implement mechanisms to handle different address modes and thus help the tester to bypass memory protection mechanisms. With that goal, we created a new hypercall to support arbitrary access operations. The interface is as follows:

```
int do_arbitrary_access(
  unsigned long addr,  // Target Address
  void *buff,          // Buffer to read/write
  size_t n,            // Buffer size
  int action);         // Operation and Mode
```

This hypercall allows a guest kernel user to read/write `n` bytes from memory starting from address `addr`. The `action` parameter specifies the operation (read/write) and the address mode (linear/physical). A linear (i.e., virtual) address is already mapped in the hypervisor and can be used directly. Some privileged instructions (e.g, `sidt`) return linear addresses. A physical address refers to the physical hardware memory, and it must be mapped prior to use. This simple interface was designed thinking of possible portability across different virtualization systems to enable the reuse of testing scripts.

The internal mechanism of our `arbitrary_access()` hypercall is straightforward. If `addr` is a physical address, we map it into the Xen linear address space and perform the read/write operation using Xen/Kernel directives that handle user/kernel memory exchanging, specifically `__copy_from_user` and

`__copy_to_user`. When `addr` is a linear address, the mapping is not needed and the hypervisor can access it directly.

We implemented the prototype on three distinct versions of Xen: 4.6, 4.8, and 4.13. Although the core of the injector is the same, small changes in the hypercalls table had to be done to add the new hypercall into the code base for each version (due to small architectural differences between versions). The build and experimental environment are kept the same during all process to restrict the differences in the run-time evaluation.

VI. Reproducing Erroneous States for Known Vulnerabilities and Attacks

In this section, we aim to answer the following Research Question (RQ):

- **RQ1:** Is it possible to inject erroneous states in a virtualized system in a way that emulates the effects of attacks exploiting real vulnerabilities?

Our prototype injector (see Section V-B) allows unlimited R/W operations, and it is quite obvious that this is not realistic. In the context of a hypervisor with shared memory among different tenants, the *reachability* problem involves determining which memory regions can be accessed or modified by an attacker who has exploited a vulnerability to break isolation mechanisms. Since we have not yet fully developed the IMs, we must have the means to ensure that the states that we are reaching are realistic. For this, we grounded the experiments on working exploits, assuming that, if an exploit is capable of triggering an erroneous state, then it is indeed reachable.

We use our intrusion injection prototype to test whether we can reproduce the effects of existing third-party-developed exploits by directly injecting the same erroneous states using the intrusion models derived from the underlying vulnerability In the following, we briefly explain the details of each exploit and the corresponding attack strategy. Then, we show how to recreate the effects of those exploits with intrusion injection. Finally, we discuss the experimental evaluation and results.

A. The Third-party Exploits

It is challenging to find publicly disclosed working exploits to reproduce attacks in Xen. In fact, there are not many public exploits available, and from the limited number, we must select those related to memory access violations in PV hosts.

To the best of our knowledge, there are four publicly available exploits that can be used under our requirements: two related to XSA-212 vulnerability [80], a third related to XSA-148 [81], and the fourth for XSA-182 [73]. We named these exploits by adding a meaningful suffix to its reference: *XSA-212-priv* and *XSA-148-priv* as these perform a privilege escalation attack, *XSA-212-crash* as it crashes the hypervisor, and *XSA-182-test* because it is intended to test if a vulnerability does exist in the system.

XSA-212 [80] describes a vulnerability in the hypercall `memory_exchange()` caused by an insufficient check on the input address. This allows a malicious guest to access all the system memory, enabling security violations such as privilege escalation with possible code execution, host/guest crashes, etc. The Google Project Zero Page presents two Proofs of Concept (PoC) for this vulnerability [80]: the first leads to a Xen crash (*XSA-212-crash*) by corrupting the page fault handler in the Interrupt Descriptor Table, and the second performs a privilege escalation (*XSA-212-priv*) on all guests in the environment by hiding another exploit in the system memory and executing it. Full details about those exploits can be found in a technical report [80].

As explained in Section IV-B, XSA-148 [27] and XSA-182 [27] are both vulnerabilities that allow creating writable page table mappings in PV guests. The first is due to a missing checking on the invariant of Xen L2 page-table entries. In the latter, the code that optimizes an L4 page update in safe cases was buggy. The re-validation of page tables is very resource expensive and the goal of those optimizations is to avoid them.

A detailed report and a working PoC for the XSA-148 is in [81]. This exploit enables **remote privilege escalation**. It does so by scanning all physical memory and looking for `dom0` *startup_info* page which can be easily fingerprinted in memory. Once found, the code searches for the vDSO (virtual Dynamic Shared Object) [82] page, in which it installs a backdoor to open a reverse shell to the outside attacker.

For the XSA-182, a detailed discussion on how to escalate privileges using this vulnerability is in [73]. In short, it follows the same attack strategy of the *XSA-148-priv*. The PoC presented is a test to check if a system is vulnerable or not to XSA-182 [73]. The test creates a self-mapping L4 page without the write flag, then uses the vulnerability to create a writable mapping, and finally, crafts a virtual address to point to the self-mapping page with writable permissions and adds the user flag to enable writing from user space.

The IMs for all exploits explained in this section are similar, differing only in the abusive functionality, as shown in Table II. In practice, our four use cases cover two distinct abusive functionalities, where the full instantiation is an unprivileged guest virtual machine that uses an hypercall to target the memory management component in the virtualization layer.

TABLE II
Intrusion Models Abusive Functionalities from the Use Cases.

Use Case	Abusive Functionality
XSA-212-crash	Write Arbitrary Memory
XSA-212-priv	Write Arbitrary Memory
XSA-148-priv	Write Page Table Entries
XSA-182-test	Write Page Table Entries

B. Injecting Intrusions

This section illustrates how our prototype can be used to inject the erroneous state corresponding to the *XSA-212-priv* use case, which is based on a complex attack strategy to escalate privileges and create conditions to run shell commands as a superuser in all virtual machines in the host (including the `dom0`). The attacker uses the vulnerability to inject malicious code into the physical memory. The code is mapped using a virtual address space that *all guests can access*. Finally, the hidden code is executed by registering a new interrupt handler entry in the IDT for every CPU and invoking it.

As discussed in section V-A, Xen restricts operations in the virtual memory page table to ensure that guest users do not abuse the system memory. As shown in the following script, the *XSA-212-priv* exploit uses the `memory_exchange()` vulnerability to manipulate the virtual memory and link a fake L2 (PMD) [83] page into an L3 (PUD) [83] page from a valid virtual address. This entry in the L2 page-table points to a forged L1 page that holds all code needed to fulfill the attack. In practice, the XSA-212 vulnerability allows an arbitrary memory write by encoding the target address in the hypercall parameter, specifically: `exch.out.extent_start + 8 * exch.nr_exchanged`.

```
exch = (struct xen_memory_exchange){
  .in = {
    .extent_start = (u64)&in_extent - (nr_exchanged * 8),
    .nr_extents = nr_extents,
    .domid = DOMID_SELF
  },
  .out = {
    .extent_start = out_extent_base_addr,
    .nr_extents = nr_extents,
    .domid = DOMID_SELF
  },
  .nr_exchanged = (target_addr - out_extent_base_addr) / 8;
};
ret = HYPERVISOR_memory_op(XENMEM_exchange, &exch);
```

The corresponding erroneous state can be injected by our hypercall injector (see Section V) by specifying the encoded address, as in the following script snippet:

```
HYPERVISOR_arbitrary_access(
    exch.out.extent_start + 8 * exch.nr_exchanged,
    &val,  sizeof(u64), ARBITRARY_WRITE_LINEAR);
```

Following this procedure, we created scripts to inject the erroneous states for each of the use cases presented in Table II.

C. Experiments and Results

An overview of the experiments to answer RQ1 is depicted in Fig. 4. The idea is to study whether it is possible to inject erroneous states in a virtualized system in a way that emulates the effects of exploiting real vulnerabilities. This can be achieved by comparing the security violation observed when injecting the erroneous state using the prototype (bottom of the figure) with the security violation observed when attacking the vulnerability using the original PoC (top of the figure), *for the same version of Xen* (in this case, the vulnerable version 4.6). Also, we want to observe if the states injected are similar to those induced by the exploits. If the violations and erroneous states observed are the same, it means that we could emulate effects caused by real intrusions.

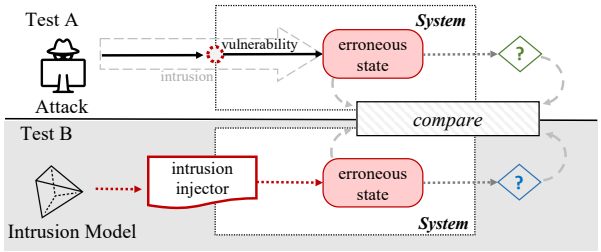

Fig. 4. Overview of the experimental validation strategy.

Following the steps described above, we first executed all PoCs in the Xen *4.6* version, being able to exploit the respective vulnerabilities in that version. We also **performed the erroneous states injection** in Xen *4.6*, as follows:

1) XSA-212-crash: the execution of the exploit generates the following output in the monitoring terminal:

```
(XEN) *** DOUBLE FAULT ***
(XEN) ----[ Xen-4.6.0  x86_64  debug=n  Tainted:    C ]----
...
(XEN) ****************************************
(XEN) Panic on CPU 23:
(XEN) DOUBLE FAULT -- system shutdown
(XEN) ****************************************
(XEN) Reboot in five seconds...
```

The erroneous effect of overwriting the IDT page fault handler is immediately followed by a crash of the hypervisor, being straightforward checking if the erroneous state and the security violation have occurred.

The ability to write to the IDT address suffices to verify if the erroneous states induced are the same. This is because the `sidt` assembler instruction fetches the IDT address that is protected for write access. Since our prototype implementation could write in this address without any exception, the same erroneous state was induced and the consequent double-fault (security violation) immediately followed.

2) XSA-212-priv: to run the exploit we need to execute the following command at the compromised guest:

```
root@guest03 ~/xsa212/privesc_poc:
$ ./attack 'echo "|$(id)|@$hostname)"' > /tmp/injector_log
press enter to contine

root@guest03 ~/xsa212/privesc_poc:
```

While executing, the script outputs messages that indicate that the L2 page was linked to the L3 page:

```
[  116.268081] ### crafted PUD entry written
[  116.284080] going to link PMD into target PUD
[  116.292081] linked PMD into target PUD
```

When the process ends, a file `/tmp/injector_log` appears in every domain with a content similar to the one below:

```
root@xen3 ~:
$ cat /tmp/injector_log
|uid=0(root) gid=0(root) groups=0(root)|@xen3
```

When injecting the erroneous state (using the script in Section VI-B, the presence of the file in each domain shows that the same security violation happened. The observation of the same output message and the same linked pages, in both executions, shows that the erroneous states are the same.

3) XSA-148-priv: this use case involves an additional host where the reverse shell is connected. It starts by creating a remote connection listener in the host where the attack will be held. We used the following command to listen for connections: `nc -l -vvv -p 1234`. Then, in the compromised guest, we run the exploit, which sets up a crafted page table to access unrestricted memory addresses (‘`start_dump ok`’ line below) and searches for the hypervisor memory fingerprints to install the backdoor. Once the vDSO library is patched, a connection with the remote host is established, from where the user can execute commands as a root user.

The steps logged on guest when the exploit runs are:

```
[132.1765] xen_exploit: - xen_version = 4.6
[132.1768] xen_exploit: - aligned_mfn_va = ffff880078000000
[132.1768] xen_exploit: - aligned_mfn_va mfn = 0x81400000
[132.1771] xen_exploit: - l2_entry_va = ffff880077600000
[132.1771] xen_exploit: - l2_entry_va mfn = 0x81e00000
[132.1772] xen_exploit: - startup_dump ok
[134.1972] xen_exploit: - start_info page: 0x212bc2
[134.1993] xen_exploit: - dom0!
[135.1652] xen_exploit: - dom0 vdso : 0x212219c
[135.1747] xen_exploit: - patch.
```

In the example below, the attacker can read a message left in the privileged domain root directory:

```
xen@xen2:~$ whoami
xen
xen@xen2:~$ nc -l -vvv -p1234
Listening on [0.0.0.0] (family 0, port 1234)
Connection from [10.3.1.181] port 1234 [tcp/*]
whoami && hostname
root
xen3
cat /root/root_msg
Confidential content in root folder!
```

After reproducing the erroneous state, we observed the same output messages, showing that it could read an arbitrary page and launch a connection to the remote host. We were able to escalate the privilege as the original exploit. A page-table walk to audit the same erroneous state was performed.

4) XSA-182-test: this is a test that checks the ability to create a writable self-mapping L4 page followed by the attempt to update its content. The code consists of an explicit process of checking such erroneous state. The debug messages show every memory address changed by this simple exploit:

```
poc:38 - xen_version = 4.6
poc:39 - page_directory mfn = 0x82da9
poc:40 - page_directory[260] = 0x0000000200dac063
poc:41 - page_directory[42] = 0x0000000000000000
poc:45 - rc = 0x0
poc:46 - page_directory[42] = 0x0000000082da9005
poc:50 - rc = 0x0
poc:51 - page_directory[42] = 0x0000000082da9007
poc:59 - writable_page_directory = 0x0000150a8542a000
poc:62 - writable_page_directory[260] = 0x0000000200dac063
poc:64 - writable_page_directory[260] = 0x0000000200dac067
poc:65 - vulnerable
```

The injector version was able to print the `page_directory[42] = 0x0000000082da9007` line, which is enough to show the write flag in the self-mapping L4 page.

In summary, the analysis in the previous paragraphs suggests a positive **answer to *RQ1***: it is possible to use intrusion injection to correctly induce the erroneous states and their security violations for the Xen 4.6 version, exactly as caused by the exploit scripts.

VII. Injecting Erroneous States in Non-Vulnerable Versions

In this section, we are interested in studying if the injection of erroneous states in non-vulnerable versions (i.e., versions without known vulnerabilities) is similar to the ones observed in vulnerable versions by answering the following RQ:

- **RQ2:** Can intrusion injection induce the erroneous states (similar to those observed in real intrusions) in non-vulnerable versions[1]?

We first confirmed that the original exploits (see Section VI-A) could not execute in Xen 4.8 and 4.13 versions, attesting that the vulnerabilities were indeed fixed. All attempts to execute the *original PoCs* resulted in an error, i.e., **we could not induce the erroneous states**. For example, when running the *XSA-212-crash* in those versions, the exploit execution fails with a return code of `-EFAULT` (bad address return code). Also, when running the *XSA-182-test*, the output shows a `not vulnerable` output. For the *XSA-212-priv* and *XSA-148-priv* the code fails with a kernel exception being unable to handle a page request. In summary, we *were not able to execute any of the exploits in versions 4.8 and 4.13* (all ended in errors showing that the vulnerabilities were fixed).

In the following, we discuss how we injected the **erroneous states** in non-vulnerable versions of Xen (4.8 and 4.13). We have to ensure that the erroneous states induced by the original exploit code are correctly injected by our prototype injector. For every script, we have collected evidence that show that the erroneous states have been injected:

1) XSA-212-crash: for both non-vulnerable versions we could write the IDT fault handler address without any exception, which assures that the erroneous state is the same.

2) XSA-212-priv: we checked the correct page linking by performing a page-table walk for the virtual address for both scripts and verified that indeed the same pages (i.e, the erroneous state) were linked in both 4.8 and 4.13 versions. Also, the same output message was displayed in the guest terminal: `linked PMD into target PUD`.

3) XSA-148-priv: again, for versions 4.8 and 4.13, the output messages showing the ability to read an arbitrary page and the auditing of the page-table entries written on memory confirmed the exact same erroneous states were induced.

4) XSA-182-test: for this test, we could observe the output message showing that the RW flag was added to the content of the L4 page in both non-vulnerable versions.

The overall results are presented in *Err. State* column in Table III, under versions 4.8 and 4.13. As the table shows, our prototype is effective in injecting memory-related erroneous states in Xen even as the software evolves. These observations provide an **answer to *RQ2***: intrusion injection can induce erroneous states similar to real intrusion in versions where related vulnerabilities are already fixed.

TABLE III
Results of the injection campaign in non-vulnerable versions. A property correct induced is represented by ✓. The shield means that an erroneous state is handled by the system.

Use Case	Xen 4.8		Xen 4.13	
	Err. State	Sec. Viol.	Err. State	Sec. Viol.
XSA-212-crash	✓	✓	✓	✓
XSA-212-priv	✓	✓	✓	⛉
XSA-148-priv	✓	✓	✓	✓
XSA-182-test	✓	✓	✓	⛉

VIII. Intrusion Injection for Security Assessment

We now want to study if it can be used for security assessment. The RQ is:

- **RQ3:** Can intrusion injection potentially support the assessment of security attributes in virtualized systems?

The idea consists of analyzing and compare the **security violations** observed in vulnerable versions and non-vulnerable versions of Xen. It is important to remember that injection should induce erroneous states similar to real attacks exploiting vulnerabilities, but the potential security violations observed may vary across different versions of the system (or different systems). In fact, different versions or configurations

[1]These are not really non-vulnerable versions (there may be unknown vulnerabilities), but are versions where our **use case** vulnerabilities were fixed.

may implement different security measures that handle the erroneous states in diverse ways and that may prevent totally or partially security violations.

Table III presents a summary of the security violations observed when injecting the erroneous states with our prototype implementation in the two non-vulnerable versions. We were able to cause violations in some cases, as discussed next:

1) XSA-212-crash: for this use case, we got the same crash report message in the Xen terminal for versions 4.8 and 4.13 as the one observed when executing the exploit in version 4.6.

2) XSA-212-priv: we observed the same security violation in Xen 4.8 and in the vulnerable version 4.6. However, for version 4.13, we could not induce the security violation of escalating the privilege as designed by the creators of the original exploit. The code terminates in an exception when accessing some memory areas. This limitation is a result of the security improvements applied to Xen [84]. Since there is no specification in the INTEL ABI [83], the Xen developers removed a 512GB RWX mapping of the linear page table, restricting some operations, particularly how L4 and L3 memory pages are accessed by guests, increasing the defense for some attacks strategies such as the one implemented by XSA-212-priv. For this, **despite being able to inject the erroneous state**, some assumptions of the exploit are not valid, specifically, the direct access to virtual addresses at the guest level for the range `0xffff804000000000` to `0xffff80403fffffff`, which were used by the exploit to install the malicious code. This is why we could not observe a security violation.

3) XSA-148-priv: in versions 4.8 and 4.13, we were able to connect the reverse shell on the remote host. The privilege escalation on `dom0` is triggered by a backdoor installation in an essential library. Since the erroneous state injection enables the installation to take place, the security violation also occurs.

4) XSA-182-test: also impacted by the security fixes mentioned above. The exploit creates an L4 writable self-mapping page that resides entirely in hardened memory address space. Thus, despite the injector could add the RW bit directly using its API, the self-mapping L4 page is not a valid guest space reference address anymore. Thus, when trying to update this invalid address in the 4.13 version an exception is triggered.

Analyzing the results in the Table III, we can see that the impact of the same injected erroneous state may vary depending on the target version. In fact, when evaluating Xen 4.8, the security violations observed are similar to the ones observed when attacking version 4.6 using the original exploits. However, Xen 4.13 is able to handle the erroneous states injected for the cases of XSA-212-priv and XSA-182-test. Such different behaviors can be mapped to a security hardening performed on the Xen 4.9 code [84], which we assume reflects a different security level.

We have shown that the injection of similar erroneous states in distinct versions leads to different violations, showing a distinct level of security of such systems that later was revealed as a hardening made on the system. The results presented suggest a positive **response to *RQ3***: intrusion injection can be applied to assess security attributes in virtualized systems.

IX. Discussion: Strengths and Limitations

Next, we discuss the strengths and limitations of our approach and of the prototype and experiments presented.

A. Strengths and Motivation

We propose intrusion injection motivated by the limitations of the techniques for the evaluation of the security attributes in virtualized systems and to overcome the limitations imposed by the need to know vulnerability and attacks of previous security evaluation works [18], [23]–[25], [85]–[87]. In fact, intrusion injection has several advantages compared with previous strategies. The first is the ability to consider several classes of vulnerabilities at once. This is achieved as an IM reflects a common abusive functionality induced by different attacks on top of different vulnerabilities. Although further studies are needed to address IMs definition and scope properly, a consequence is the reduction of complexity of experiments as there is no need for crafting specific attacks on a broad set of different types of vulnerabilities.

Another advantage is the ability to conduct testing campaigns without assuming any knowledge about existing vulnerabilities in the system under evaluation, possibly anticipating the effects of unknown vulnerabilities. This can be achieved by modeling vulnerabilities seen in other systems in the field. For instance, defining IMs based on KVM Hypervisor environments and then attempting to inject the corresponding erroneous states in Xen deployments. The design and run-time differences are a challenge when translating the erroneous states, but this is mostly a technical issue. In fact, to trigger similar erroneous states in different systems, we envision each system having its own injector, providing abusive functionality interfaces that handle the design and run-time differences.

B. Intrusion Models

The definition of representative Intrusion Models is a topic yet to be carefully explored and that is crucial to address the *reachability* problem in the context of erroneous states. The corruption of memory protected by a hardware virtualization extension is a clear example of an unreachable state whose injection should be avoided, while the unauthorized access of memory based on models from use-after-free vulnerabilities seems quite reasonable. The examples presented in this work are still limited, serving only for the purpose of demonstrating the viability of the intrusion injection approach. A key aspect to consider is that the definition of Intrusion Models should take into account portability aspects: if models are system-specific then they would not be useful to assess and compare different systems (or different versions of the same system).

Attackers exploit vulnerabilities and weaknesses to subvert the system in multiple steps. Each step towards a system breach can be modeled as an abusive functionality, i.e., an injection of erroneous states through a given interface. While the difficulty of modeling an abusive functionality can be debated, especially considering reachability, our concept provides a reasonable representation of each step of an attack. Although our technique is not directly associated with the act

of attacking the system (but instead, with the corresponding intrusions), conceptually, a set of intrusion injectors can emulate the outcomes of the tools that attackers use to perform complex attacks (e.g., advanced persistent threats (APTs)).

As discussed before, the proposed approach can potentially be used to assess the impact of any unknown vulnerability as far as it is covered by an Intrusion Model. However, the ability to cover and generalize unknown vulnerabilities in a system (and, indirectly, attacks on them) by intrusion injection is highly correlated to the definition of Intrusion Models. In other words, we can only state that intrusion injection may be capable of foreseen consequences of unknown vulnerabilities if they have a IM (i.e., abusive functionalities, etc.) similar to a previously studied and classified vulnerability. There is always a chance that an unpredictable type of abusive functionality might appear that would not relate to previously discovered vulnerabilities. Still, we believe its frequency would be less than the already observable ones.

IMs are crucial to tackle the *reachability* problem and distinguish intrusion injection from arbitrary erroneous state injection and from "accidental faults/errors". In fact, such models should include multiple properties like the abusive capability achieved, the target components, the attack interfaces, and the type of erroneous states, which will help distinguish reachable and unreachable states in advance. At the same time, this is not straightforward to abstract once multiple functionalities, architectures, and technologies are involved in a complex system like a hypervisor. For this, we must carefully (and continuously) learn from actual intrusions trying to reach a common set of properties that establish at least a minimum useful intrusion model to allow a trustworthy assessment.

C. Prototype and Experiments

The prototype presented is limited as it is just a proof-of-concept to shows the viability of Intrusion Injection. The case study was done with memory attacks since more memory-related vulnerabilities are more prevalent [44], [45], [47], allowing us to show feasibility. Nevertheless, the approach is threat vector agnostic and can be mapped to other components, e.g., interruptions, device drivers, IO. We are expanding our prototype to cover IMs related with malicious interrupts and activities originating from the management interface.

In the experiments, we showed that we could determine the erroneous state observed in an exploited vulnerable system and inject it into a non-vulnerable version. We also showed that, when injecting the same erroneous state in similar systems, we may observe security violations in one (Xen 4.8) and not in the other (Xen 4.13). Finally, we demonstrated that intrusion injection can be useful to assess security attributes in virtualized systems. All this was done by keeping both environmental aspects (hardware, OS, VMs, building systems, etc.) completely the same. The only difference was the Xen version. However, **the aim was to show viability, not validity, or full utility of the technique**, and thus a complete validation must be conducted as a future step in this research.

D. Scope and Limits

The precise scope and limits of the intrusion injection approach are yet unknown. Any attacks besides memory-corruption bugs (on the hypervisor) and the so-called "logic flaws" (design flaws, side-channel attacks, etc) are not covered by our intrusion models. However, memory corruption bugs on the hypervisor may translate into non-memory components in the virtualized environments (e.g., interruptions, IO) since virtualization enables that (e.g., interruptions are implemented using event channel data structures in Xen). Our focus is security vulnerabilities introduced through programming mistakes. Despite the promising results presented in this paper, there is a chance that, for complex IMs, one may not be able to find viable solutions to expose an interface that enables injection.

Intrusiveness is another aspect can be seen as a drawback since the injection of erroneous states may require modifying the system, limiting the applicability. We are convinced that this can be handled by choosing adequate injection solutions. Still, trading some intrusiveness for flexibility and increased assessment capabilities may be an acceptable compromise.

X. Conclusions

In this paper, we introduced intrusion injection, a novel approach for the injection of erroneous states that can be used to assess how virtualized systems handle intrusions. The proposed approach brings sound advantages, such as enabling security evaluation without requiring knowledge about the existing vulnerabilities or attacks. We believe that intrusion injection opens the door for the future development of techniques for security assessment and benchmarking that, instead of relying on real vulnerabilities and attacks (or on their emulation), are based on the injection of the consequences of intrusions, which can be applied to different systems similarly.

We expect to apply it in assessing the security attributes of hypervisors and establish a security benchmark for virtualized infrastructures in the future. To that goal, future work includes consolidating Intrusion Models to define the scope and requirements for new injectors. We also plan to implement different injectors and an open-source list of tests and experiments covering various Intrusion Models, fostering community involvement and broader applicability.

Acknowledgments

This work is funded by Project "Agenda Mobilizadora Sines Nexus". ref. No. 7113), supported by the Recovery and Resilience Plan (PRR) and by the European Funds Next Generation EU, following Notice No. 02/C05-i01/2022, Component 5 - Capitalization and Business Innovation - Mobilizing Agendas for Business Innovation, by national funds through the FCT - Foundation for Science and Technology, I.P., within the scope of the project CISUC - UID/CEC/00326/2020, grant SFRH/BD/144839/2019, by European Social Fund, through the Regional Operational Program Centro 2020, and by CEFET-MG. It was also supported SPEC RG Security Benchmarking (Standard Performance Evaluation Corporation; http://www.spec.org, http://research.spec.org).

REFERENCES

[1] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, no. 7, p. 412421, Jul. 1974.

[2] C. H. Kao, "Survey on evaluation of iot services leveraging virtualization technology," in *Proceedings of the 2020 5th International Conference on Cloud Computing and Internet of Things*. Association for Computing Machinery, 2020, p. 2634.

[3] A. Tellabi, M. Parekh, C. Ruland, and M. Ezziyyani, "A case study of virtualization used in mixed criticality systems," in *International Conference on Advanced Intelligent Systems for Sustainable Development*. Springer, 2019, pp. 1–11.

[4] V. Bandur, G. Selim, V. Pantelic, and M. Lawford, "Making the Case for Centralized Automotive E/E Architectures," *IEEE Transactions on Vehicular Technology*, vol. 70, no. 2, pp. 1230–1245, feb 2021.

[5] R. T. Tiburski, C. R. Moratelli, S. F. Johann, E. de Matos, and F. Hessel, "A lightweight virtualization model to enable edge computing in deeply embedded systems," in *Software - Practice and Experience*, 2021.

[6] L. Hatton, "The chimera of software quality," *Computer*, vol. 40, no. 8, pp. 104–103, 2007.

[7] S. M. Bellovin, "On the brittleness of software and the infeasibility of security metrics," *IEEE Security and Privacy*, vol. 4, no. 4, p. 96, jul 2006.

[8] H. F. Lipson, "Tracking and Tracing Cyber-Attacks : Technical Challenges and Global Policy," Carnegie Mellon University, Tech. Rep. November, 2002.

[9] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz, "Comparing operating systems using robustness benchmarks," in *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, 1997, pp. 72–79.

[10] W. R. Blischke and D. P. Murthy, *Reliability: modeling, prediction, and optimization*. John Wiley & Sons, 2011, vol. 767.

[11] C. F. Gonçalves, N. Antunes, and M. Vieira, "Evaluating the applicability of robustness testing in virtualized environments," in *8th Latin-American Symposium on Dependable Computing, LADC 2018,*. IEEE, 2018, pp. 161–166.

[12] R. Buhren, H. N. Jacob, T. Krachenfels, and J. Seifert, "One glitch to rule them all: Fault injection attacks against amd's secure encrypted virtualization," in *SIGSAC Conference on Computer and Communications Security, Virtual Event*. ACM, 2021.

[13] D. Cotroneo, F. Frattini, R. Pietrantuono, and S. Russo, "State-based robustness testing of iaas cloud platforms," in *Proceedings of the 5th International Workshop on Cloud Data and Platforms,*. ACM, 2015, pp. 3:1–3:6.

[14] H. Madeira, D. Costa, and M. Vieira, "On the emulation of software faults by software fault injection," in *International Conference on Dependable Systems and Networks. DSN 2000*, 2000, pp. 417–426.

[15] K. Pattabiraman, N. Nakka, Z. T. Kalbarczyk, and R. K. Iyer, "Symplfied: Symbolic program-level fault injection and error detection framework," *IEEE Trans. Computers*, vol. 62, no. 11, pp. 2292–2307, 2013.

[16] K. Kanoun and L. Spainhower, *Dependability Benchmarking for Computer Systems*. Wiley-IEEE Computer Society Pr, 2008.

[17] J. Fonseca, M. Vieira, and H. Madeira, "Evaluation of Web Security Mechanisms Using Vulnerability & Attack Injection," *IEEE Transactions on Dependable and Secure Computing*, vol. 11, no. 5, pp. 440–453, 2014.

[18] N. Neves, J. Antunes, M. Correia, P. Veríssimo, and R. Neves, "Using attack injection to discover new vulnerabilities," *Proceedings of the International Conference on Dependable Systems and Networks*, vol. 2006, pp. 457–466, 2006.

[19] R. V. Bhor and H. K. Khanuja, "Analysis of web application security mechanism and attack detection using vulnerability injection technique," in *2016 International Conference on Computing Communication Control and automation (ICCUBEA)*, 2016, pp. 1–6.

[20] C. F. Goncalves and N. Antunes, "Vulnerability Analysis as Trustworthiness Evidence in Security Benchmarking: A Case Study on Xen." in *Proceedings - 2020 IEEE 31st International Symposium on Software Reliability Engineering Workshops, ISSREW 2020*, 2020.

[21] A. Adams, "Adventures in xen exploitation," https://research.nccgroup.com/2015/02/27/adventures-in-xen-exploitation/, 2015, accessed: 2021-03-27.

[22] G. Canfora, A. D. Sorbo, S. Forootani, A. Pirozzi, and C. A. Visaggio, "Investigating the vulnerability fixing process in OSS projects: Peculiarities and challenges," *Comput. Secur.*, vol. 99, p. 102067, 2020.

[23] J. Fonseca, M. Vieira, and H. Madeira, "Vulnerability & attack injection for web applications," in *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, 2009, pp. 93–102.

[24] C. Mainka, J. Somorovsky, and J. Schwenk, "Penetration testing tool for web services security," in *2012 IEEE Eighth World Congress on Services*, 2012, pp. 163–170.

[25] R. V. Bhor and H. K. Khanuja, "Analysis of web application security mechanism and attack detection using vulnerability injection technique," in *2016 International Conference on Computing Communication Control and automation (ICCUBEA)*, 2016, pp. 1–6.

[26] 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 - SOSP '03*, vol. 37, no. 5. ACM, 2003, p. 164.

[27] Xen, "Xen Security Advisory," https://xenbits.xen.org/xsa/, 2015, accessed: 2021-03-27.

[28] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Trans. Software Eng.*, vol. 47, no. 11, pp. 2312–2331, 2021.

[29] K. Serebryany, "OSS-Fuzz - google's continuous fuzzing service for open source software." Vancouver, BC: USENIX Association, Aug. 2017.

[30] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang, "ProFuzzer: On-the-fly input type probing for better zero-day vulnerability discovery," *Proceedings - IEEE Symposium on Security and Privacy*, vol. 2019-May, pp. 769–786, 2019.

[31] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "Krace: Data race fuzzing for kernel file systems," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1643–1660.

[32] R. Zhong, Y. Chen, H. Hu, H. Zhang, W. Lee, and D. Wu, "SQUIRREL: testing database management systems with language validity and coverage feedback," in *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. ACM, 2020, pp. 955–970.

[33] B. Cook, B. Dobel, D. Kroening, N. Manthey, M. Pohlack, E. Polgreen, M. Tautschnig, and P. Wieczorkiewicz, "Using model checking tools to triage the severity of security bugs in the Xen hypervisor," *Proceedings of the 20th Conference on Formal Methods in Computer-Aided Design, FMCAD 2020*, pp. 185–193, 2020.

[34] Y. She, H. Li, and H. Zhu, "UVHM: Model checking based formal analysis scheme for hypervisors," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7804 LNCS, pp. 300–305, 2013.

[35] J. Franklin, S. Chaki, A. Datta, J. M. McCune, and A. Vasudevan, "Parametric verification of address space separation," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7215 LNCS, no. i, pp. 51–68, 2012.

[36] S. W. Li, X. Li, R. Gu, J. Nieh, and J. Zhuang Hui, "A secure and formally verified linux KVM hypervisor," *Proceedings - IEEE Symposium on Security and Privacy*, vol. 2021-May, pp. 1782–1799, 2021.

[37] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta, "Design, implementation and verification of an extensible and modular hypervisor framework," in *Proceedings - IEEE Symposium on Security and Privacy*, 2013, pp. 430–444.

[38] H. Khosrowjerdi, K. Meinke, and A. Rasmusson, "Virtualized-fault injection testing: A machine learning approach," in *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*. IEEE Computer Society, 2018, pp. 297–308.

[39] R. Amarnath, S. N. Bhat, P. Munk, and E. Thaden, "A fault injection approach to evaluate soft-error dependability of system calls," in *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2018, pp. 71–76.

[40] M. Le and Y. Tamir, "Fault injection in virtualized systems—challenges and applications," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 3, pp. 284–297, 2015.

[41] F. Cerveira, R. Barbosa, H. Madeira, and F. Araujo, "The effects of soft errors and mitigation strategies for virtualization servers," *IEEE Transactions on Cloud Computing*, vol. 10, no. 2, pp. 1065–1081, 2022.

[42] F. Hauschild, K. Garb, L. Auer, B. Selmke, and J. Obermaier, "Archie: A qemu-based framework for architecture-independent evaluation of

faults," in *2021 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, 2021, pp. 20–30.

[43] A. A. Neto and M. Vieira, "Selecting Secure Web Applications Using Trustworthiness Benchmarking," *International Journal of Dependable and Trustworthy Information Systems*, vol. 2, no. 2, pp. 1–16, 2012.

[44] M. Compastié, R. Badonnel, O. Festor, and R. He, "From virtualization security issues to cloud protection opportunities: An in-depth analysis of system virtualization models," *Computers & Security*, vol. 97, p. 101905, oct 2020.

[45] R. Patil and C. Modi, "An exhaustive survey on security concerns and solutions at different components of virtualization," *ACM Computing Surveys*, vol. 52, no. 1, 2019.

[46] A. Litchfield and A. Shahzad, "A systematic review of vulnerabilities in hypervisors and their detection," in *AMCIS 2017 - America's Conference on Information Systems: A Tradition of Innovation*, vol. 2017-Augus, 2017.

[47] D. Sgandurra and E. Lupu, "Evolution of attacks, threat models, and solutions for virtualized systems," *ACM Computing Surveys*, vol. 48, no. 3, feb 2016.

[48] E. Yasasin, J. Prester, G. Wagner, and G. Schryen, "Forecasting IT security vulnerabilities An empirical analysis," *Computers and Security*, vol. 88, p. 101610, jan 2020.

[49] O. H. Alhazmi, Y. K. Malaiya, and I. Ray, "Measuring, analyzing and predicting security vulnerabilities in software systems," *Computers and Security*, vol. 26, no. 3, pp. 219–228, may 2007.

[50] A. Milenkoski, B. D. Payne, N. Antunes, M. Vieira, S. Kounev, A. Avritzer, and M. Luft, "Evaluation of intrusion detection systems in virtualized environments using attack injection," in *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses*, ser. RAID, 2015.

[51] M. Mostafavi and P. Kabiri, "Detection of repetitive and irregular hypercall attacks from guest virtual machines to xen hypervisor," *Iran journal of computer science*, vol. 1, no. 2, pp. 89–97, 2018.

[52] F. Cerveira, R. Barbosa, H. Madeira, and F. Araujo, "Recovery for Virtualized Environments," *Proceedings - 2015 11th European Dependable Computing Conference, EDCC 2015*, pp. 25–36, 2016.

[53] P. Mell, R. Lippmann, Chung, J. Haines, and M. Zissman, "An overview of issues in testing intrusion detection systems," 2003-07-11 2003.

[54] T. Dumitraş and D. Shou, "Toward a standard benchmark for computer security research: The worldwide intelligence network environment (wine)," in *Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, ser. BADGERS '11. New York, NY, USA: ACM, 2011, p. 8996.

[55] S. Kounev, K.-D. Lange, and J. von Kistowski, *Systems Benchmarking - For Scientists and Engineers*. Springer, 2020.

[56] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 01, no. 1, pp. 11–33, 2004.

[57] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira, "On fault representativeness of software fault injection," *IEEE Transactions on Software Engineering*, vol. 39, no. 1, pp. 80–96, 2013.

[58] J. Carreira, H. Madeira, and J. G. Silva, "Xception: A technique for the experimental evaluation of dependability in modern computers," *IEEE Transactions on Software Engineering*, vol. 24, no. 2, pp. 125–136, 1998.

[59] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: A methodology and some applications," *IEEE Transactions on software engineering*, vol. 16, no. 2, pp. 166–182, 1990.

[60] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. H. Leber, "Comparison of physical and software-implemented fault injection techniques," *IEEE Transactions on Computers*, vol. 52, no. 9, pp. 1115–1133, 2003.

[61] E. Clarke, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model Checking, second edition*, ser. Cyber Physical Systems Series. MIT Press, 2018. [Online]. Available: https://books.google.pt/books?id=OJV5DwAAQBAJ

[62] A. Johansson and N. Suri, "Error propagation profiling of operating systems," in *2005 International Conference on Dependable Systems and Networks (DSN'05)*, 2005, pp. 86–95.

[63] R. Chillarege, W.-L. Kao, and R. Condit, "Defect type and its impact on the growth curve (software development)," in *[1991 Proceedings] 13th International Conference on Software Engineering*, 1991, pp. 246–255.

[64] MITRE, "Common Weakness Enumeration," https://cwe.mitre.org/, accessed: 2021-06-09.

[65] A. Ghaleb, I. Traore, and K. Ganame, "A framework architecture for agentless cloud endpoint security monitoring," in *2019 IEEE Conference on Communications and Network Security (CNS)*, 2019, pp. 1–9.

[66] A. G. Ramirez, M. M. Pedreira, C. Grigoras, L. Betev, C. Lara, U. Kebschull, A. Collaboration *et al.*, "A security monitoring framework for virtualization based hep infrastructures," in *Journal of Physics: Conference Series*, vol. 898, no. 10. IOP Publishing, 2017, p. 102004.

[67] C. Pham, Z. Estrada, P. Cao, Z. Kalbarczyk, and R. K. Iyer, "Reliability and security monitoring of virtual machines using hardware architectural invariants," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 13–24.

[68] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*, 2008, pp. 233–247.

[69] R. Ragel and S. Parameswaran, "Impres: integrated monitoring for processor reliability and security," in *2006 43rd ACM/IEEE Design Automation Conference*, 2006, pp. 502–505.

[70] 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.

[71] X. Li, X. Chang, J. A. Board, and K. S. Trivedi, "A novel approach for software vulnerability classification," *Proceedings - Annual Reliability and Maintainability Symposium*, 2017.

[72] A. Gkortzis, S. Rizou, and D. Spinellis, "An empirical analysis of vulnerabilities in virtualization technologies," in *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*, vol. 0. IEEE Computer Society, jul 2016, pp. 533–538.

[73] J. Boutoille, "Xen exploitation part 3: Xsa-182, qubes escape," https://blog.quarkslab.com/xen-exploitation-part-3-xsa-182-qubes-escape.html, 2016, accessed: 2021-09-11.

[74] T. Dullien, "Weird Machines, Exploitability, and Provable Unexploitability," *IEEE Transactions on Emerging Topics in Computing*, vol. 8, no. 2, pp. 391–403, 2017.

[75] J. Bangert, S. Bratus, R. Shapiro, and S. W. Smith, "The page-fault weird machine: Lessons in instruction-less computation," *7th USENIX Workshop on Offensive Technologies, WOOT 2013*, 2013.

[76] S. Bratus, M. Locasto, and M. Patterson, "Exploit Programming: From Buffer Overflows to " Weird Machines " and Theory of Computation," *USENIX; login*, pp. 13–21, 2011.

[77] NIST, "National Vulnerability Database," https://nvd.nist.gov/, accessed: 2021-06-07.

[78] D. Chisnall, *The Definitive Guide to the Xen Hypervisor*, 1st ed. USA: Prentice Hall Press, 2013.

[79] X. P. Wiki, "X86 paravirtualised memory management," https://wiki.xenproject.org/wiki/X86_Paravirtualised_Memory_Managemen, 2015, accessed: 2021-02-27.

[80] J. Horn, "Issue 1184: Xen: Broken check in memory exchange permits pv guest breakout," https://bugs.chromium.org/p/project-zero/issues/detail?id=1184, 2017, accessed: 2021-03-30.

[81] J. Boutoille, "Xen exploitation part 2: Xsa-148, from guest to host," https://bugs.chromium.org/p/project-zero/issues/detail?id=1184, 2016, accessed: 2021-08-15.

[82] *vdso(7) — Linux manual page*, https://man7.org/linux/man-pages/man7/vdso.7.html, Linux Foundation, accessed: 2021-11-07.

[83] *Intel 64 and IA-32 Architectures Software Developer's Manual - Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*, Intel Corporation, November 2020.

[84] A. Cooper, "Xen-devel [PATCH 0/7] XSAs 213-315 followups," https://lists.xenproject.org/archives/html/xen-devel/2017-05/msg00149.html, 2017, accessed: 2021-04-27.

[85] R. A. Oliveira, M. M. Raga, N. Laranjeiro, and M. Vieira, "An approach for benchmarking the security of web service frameworks," *Future Generation Computer Systems*, vol. 110, pp. 833–848, nov 2020.

[86] A. A. Neto and M. Vieira, "Selecting secure web applications using trustworthiness benchmarking," *Int. Journal of Dependable and Trustworthy Information Systems (IJDTIS)*, vol. 2, no. 2, pp. 1–16, 2011.

[87] I. Medeiros, N. Neves, and M. Correia, "Detecting and removing web application vulnerabilities with static analysis and data mining," *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 54–69, 2016.