Aleksandar Milenkoski

Evaluation of Intrusion Detection Systems in Virtualized Environments

Aleksandar Milenkoski

Evaluation of Intrusion Detection Systems in Virtualized Environments

Dissertation, Julius-Maximilians-Universität Würzburg
Fakultät für Mathematik und Informatik, 2016
Erster Gutachter: Prof. Dr. Samuel Kounev
Zweiter Gutachter: Prof. Dr. Felix Freiling
Dritter Gutachter: Prof. Dr. Michael Meier

Contents

Publication List

Peer-Reviewed Journal Articles

[MVK+15] Aleksandar Milenkoski, Marco Vieira, Samuel Kounev, Alberto Avritzer, and Bryan D. Payne. Evaluating Computer Intrusion Detection Systems: A Survey of Common Practices. ACM Computing Surveys, 48(1):12:1–12:41, Sep 2015, ACM, New York, NY, USA. **5-year Impact Factor (2014): 5.949**.

Books

[KKMZ17] Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu. Self-Aware Computing Systems. Springer Verlag Berlin Heidelberg, Germany, 2016. To appear in 2017, number of pages: approximately 650.

Peer-Reviewed International Conference Contributions

Full Research Papers

[MJA+16] Aleksandar Milenkoski, K. R. Jayaram, Nuno Antunes, Marco Vieira, and Samuel Kounev. Quantifying the Attack Detection Accuracy of Intrusion Detection Systems in Virtualized Environments. In Proceedings of The 27th IEEE International Symposium on Software Reliability Engineering (ISSRE 2016), Washington DC, USA, October 2016. IEEE, IEEE Computer Society. To Appear.

[MPA+15] Aleksandar Milenkoski, Bryan D. Payne, Nuno Antunes, Marco Vieira, Samuel Kounev, Alberto Avritzer, and Matthias Luft. Evaluation of Intrusion Detection Systems in Virtualized Environments Using Attack Injection. In The 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID 2015). Springer, November 2015. **Acceptance Rate: 23%**.

[MPA+14] Aleksandar Milenkoski, Bryan D. Payne, Nuno Antunes, Marco Vieira, and Samuel Kounev. An Analysis of Hypercall Handler Vulnerabilities. In Proceedings of The 25th IEEE International Symposium on Software Reliability Engineering (ISSRE 2014). IEEE, 2014. **Acceptance Rate: 25%, Best Paper Award Nomination**.

Short/Work-in-progress Papers

[CAV+15] Diogo Carvalho, Nuno Antunes, Marco Vieira, Aleksandar Milenkoski, and Samuel Kounev. Challenges of Assessing the Hypercall Interface Robustness (Fast Abstract). In The 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2015). IEEE, June 2015.

[MPA+13] Aleksandar Milenkoski, Bryan D. Payne, Nuno Antunes, Marco Vieira, and Samuel Kounev. HInjector: Injecting Hypercall Attacks for Evaluating VMI-based Intrusion Detection Systems. In Poster Reception at the 2013 Annual Computer Security Applications Conference (ACSAC 2013), Maryland, USA, 2013. Applied Computer Security Associates (ACSA).

[MK12] Aleksandar Milenkoski and Samuel Kounev. Towards Benchmarking Intrusion Detection Systems for Virtualized Cloud Environments (Work-in-Progress Paper). In Proceedings of the 7th International Conference for Internet Technology and Secured Transactions (ICITST 2012), pages 562–563, New York, USA, December 2012. IEEE.

Research Project Grants

[evi16] EvIDencE: Testing Intrusion Detection Systems in Virtualized Environments; EvIDencE: Testen von Systemen zur Angriffserkennung in virtualisierten Umgebungen (orig., ger.), 2016. Awarded by the German Research Foundation; Deutsche Forschungsgemeinschaft (DFG).

The project proposal written for this grant, submitted with Prof. Dr. Samuel Kounev, is based on this thesis and captures future work directly related to the contributions of the thesis.

Peer-Reviewed Magazine Articles

[MIK+16] Aleksandar Milenkoski, Alexandru Iosup, Samuel Kounev, Kai Sachs, Diane E. Mularz, Jonathan A. Curtiss, Jason J. Ding, Florian Rosenberg, and Piotr Rygielski. CUP: A Formalism for Expressing Cloud Usage Patterns for Experts and Non-Experts. IEEE Cloud Computing, 2016. To Appear.

Book Chapters

[MJK17] Aleksandar Milenkoski, K. R. Jayaram, and Samuel Kounev. Benchmarking Intrusion Detection Systems with Adaptive Provisioning of Virtualized Resources. In Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu, editors, Self-Aware Computing Systems. Springer Verlag Berlin Heidelberg, Germany, 2017. To Appear.

[JMK17] K. R. Jayaram, Aleksandar Milenkoski, and Samuel Kounev. Software Architectures for Self-Protection in IaaS Clouds. In Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu, editors, Self-Aware Computing Systems. Springer Verlag Berlin Heidelberg, Germany, 2017. To Appear.

[HBK+17] Nikolas Herbst, Steffen Becker, Samuel Kounev, Heiko Koziolek, Martina Maggio, Aleksandar Milenkoski and Evgenia Smirni. Metrics and Benchmarks for Self-Aware Computing Systems. In Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu, editors, Self-Aware Computing Systems. Springer Verlag Berlin Heidelberg, Germany, 2017. To Appear.

Edited Dagstuhl Reports

[Sam15] Samuel Kounev, Xiaoyun Zhu, Jeffrey O. Kephart, and Marta Kwiatkowska, editors. Aleksandar Milenkoski, assistant editor. Model-driven Algorithms and Architectures for Self-Aware Computing Systems. Dagstuhl Reports, 2015. Dagstuhl, Germany.

Technical Reports

[MVP+14] Aleksandar Milenkoski, Marco Vieira, Bryan D. Payne, Nuno Antunes, and Samuel Kounev. Technical Information on Vulnerabilities of Hypercall Handlers. Technical Report SPEC-RG-2014-001 v.1.0, SPEC Research Group - IDS Benchmarking Working Group, Standard Performance Evaluation Corporation (SPEC), 7001 Heritage Village Plaza Suite 225, Gainesville, VA 20155, USA, August 2014.

[MKA+13] Aleksandar Milenkoski, Samuel Kounev, Alberto Avritzer, Nuno Antunes, and Marco Vieira. On Benchmarking Intrusion Detection Systems in Virtualized Environments. Technical Report SPEC-RG-2013-002 v.1.0, SPEC Research Group - IDS Benchmarking Working Group, Standard Performance Evaluation Corporation (SPEC), 7001 Heritage Village Plaza Suite 225, Gainesville, VA 20155, USA, June 2013.

[MIK+13] Aleksandar Milenkoski, Alexandru Iosup, Samuel Kounev, Kai Sachs, Piotr Rygielski, Jason Ding, Walfredo Cirne, and Florian Rosenberg. Cloud Usage Patterns: A Formalism for Description of Cloud Usage Scenarios. Technical Report SPEC-RG-2013-001 v.1.0.1, SPEC Research Group - Cloud Working Group, Standard Performance Evaluation Corporation (SPEC), 7001 Heritage Village Plaza Suite 225, Gainesville, VA 20155, USA, May 2013.

Industry White Papers

[Ale16] Aleksandar Milenkoski, Bernd Jaeger, Kapil Raina, Mason Harris, Saif Chaudhry, Sivadon Chasiri, Veronica David, and Wenmao Liu. Security Position Paper: Network Function Virtualization. Cloud Security Alliance - Virtualization Working Group,

March 2016.

[Abh15] Abhik Chaudhuri, Heberto Ferrer, Hemma Prafullchandra, JD Sherry, Kelvin Ng, Xiaoyu Ge, Yao Sing and Yiak Por (Main Contributors). Aleksandar Milenkoski (Minor Contributor). Best Practices for Mitigating Risks in Virtualized Environments. Cloud Security Alliance - Virtualization Working Group, April 2015.

Abstract

Virtualization allows the creation of virtual instances of physical devices, such as network and processing units. In a virtualized system, governed by a hypervisor, resources are shared among virtual machines (VMs). Virtualization has been receiving increasing interest as a way to reduce costs through server consolidation and to enhance the flexibility of physical infrastructures. Although virtualization provides many benefits, it introduces new security challenges; that is, the introduction of a hypervisor introduces threats since hypervisors expose new attack surfaces.

Intrusion detection is a common cyber security mechanism whose task is to detect malicious activities in host and/or network environments. This enables timely reaction in order to stop an on-going attack, or to mitigate the impact of a security breach. The wide adoption of virtualization has resulted in the increasingly common practice of deploying conventional intrusion detection systems (IDSs), for example, hardware IDS appliances or common software-based IDSs, in designated VMs as virtual network functions (VNFs). In addition, the research and industrial communities have developed IDSs specifically designed to operate in virtualized environments (i.e., hypervisor-based IDSs), with components both inside the hypervisor and in a designated VM. The latter are becoming increasingly common with the growing proliferation of virtualized data centers and the adoption of the cloud computing paradigm, for which virtualization is as a key enabling technology.

To minimize the risk of security breaches, methods and techniques for evaluating IDSs in an accurate manner are essential. For instance, one may compare different IDSs in terms of their attack detection accuracy in order to identify and deploy the IDS that operates optimally in a given environment, thereby reducing the risks of a security breach. However, methods and techniques for realistic and accurate evaluation of the attack detection accuracy of IDSs in virtualized environments (i.e., IDSs deployed as VNFs or hypervisor-based IDSs) are lacking. That is, workloads that exercise the sensors of an evaluated IDS and contain attacks targeting hypervisors are needed. Attacks targeting hypervisors are of high severity since they may result in, for example, altering the hypervisors's memory and thus enabling the execution of malicious code with hypervisor privileges. In addition, there are no metrics and measurement methodologies for accurately quantifying the attack detection accuracy of IDSs in virtualized environments with elastic resource provisioning (i.e., on-demand allocation or deallocation of virtualized hardware resources to VMs). Modern hypervisors allow for hotplugging virtual CPUs and memory on the designated VM where the intrusion detection engine of hypervisor-based IDSs, as well as of IDSs deployed as VNFs, typically operates. Resource hotplugging may have a significant impact on the attack detection accuracy of an evaluated IDS, which is not taken into account by existing metrics for quantifying

IDS attack detection accuracy. This may lead to inaccurate measurements, which, in turn, may result in the deployment of misconfigured or ill-performing IDSs, increasing the risk of security breaches.

This thesis presents contributions that span the standard components of any system evaluation scenario: workloads, metrics, and measurement methodologies. The scientific contributions of this thesis are:

- A comprehensive systematization of the common practices and the state-of-the-art on IDS evaluation. This includes: *(i)* a definition of an IDS evaluation design space allowing to put existing practical and theoretical work into a common context in a systematic manner; *(ii)* an overview of common practices in IDS evaluation reviewing evaluation approaches and methods related to each part of the design space; *(iii)* and a set of case studies demonstrating how different IDS evaluation approaches are applied in practice. Given the significant amount of existing practical and theoretical work related to IDS evaluation, the presented systematization is beneficial for improving the general understanding of the topic by providing an overview of the current state of the field. In addition, it is beneficial for identifying and contrasting advantages and disadvantages of different IDS evaluation methods and practices, while also helping to identify specific requirements and best practices for evaluating current and future IDSs.
- An in-depth analysis of common vulnerabilities of modern hypervisors as well as a set of attack models capturing the activities of attackers triggering these vulnerabilities. The analysis includes 35 representative vulnerabilities of hypercall handlers (i.e., hypercall vulnerabilities). Hypercalls are software traps from a kernel of a VM to the hypervisor. The hypercall interface of hypervisors, among device drivers and VM exit events, is one of the attack surfaces that hypervisors expose. Triggering a hypercall vulnerability may lead to a crash of the hypervisor or to altering the hypervisor's memory. We analyze the origins of the considered hypercall vulnerabilities, demonstrate and analyze possible attacks that trigger them (i.e., hypercall attacks), develop hypercall attack models (i.e., systematized activities of attackers targeting the hypercall interface), and discuss future research directions focusing on approaches for securing hypercall interfaces.
- A novel approach for evaluating IDSs enabling the generation of workloads that contain attacks targeting hypervisors, that is, hypercall attacks. We propose an approach for evaluating IDSs using attack injection (i.e., controlled execution of attacks during regular operation of the environment where an IDS under test is deployed). The injection of attacks is performed based on attack models that capture realistic attack scenarios. We use the hypercall attack models developed as part of this thesis for injecting hypercall attacks.
- A novel metric and measurement methodology for quantifying the attack detection accuracy of IDSs in virtualized environments that feature elastic resource provisioning. We demonstrate how the elasticity of resource allocations in such environments may impact the IDS attack detection accuracy and show that using

existing metrics in such environments may lead to practically challenging and inaccurate measurements. We also demonstrate the practical use of the metric we propose through a set of case studies, where we evaluate common conventional IDSs deployed as VNFs.

In summary, this thesis presents the first systematization of the state-of-the-art on IDS evaluation, considering workloads, metrics and measurement methodologies as integral parts of every IDS evaluation approach. In addition, we are the first to examine the hypercall attack surface of hypervisors in detail and to propose an approach using attack injection for evaluating IDSs in virtualized environments. Finally, this thesis presents the first metric and measurement methodology for quantifying the attack detection accuracy of IDSs in virtualized environments that feature elastic resource provisioning.

From a technical perspective, as part of the proposed approach for evaluating IDSs, this thesis presents hInjector, a tool for injecting hypercall attacks. We designed hInjector to enable the rigorous, representative, and practically feasible evaluation of IDSs using attack injection. We demonstrate the application and practical usefulness of hInjector, as well as of the proposed approach, by evaluating a representative hypervisor-based IDS designed to detect hypercall attacks. While we focus on evaluating the capabilities of IDSs to detect hypercall attacks, the proposed IDS evaluation approach can be generalized and applied in a broader context. For example, it may be directly used to also evaluate security mechanisms of hypervisors, such as hypercall access control (AC) mechanisms. It may also be applied to evaluate the capabilities of IDSs to detect attacks involving operations that are functionally similar to hypercalls, for example, the input/output control (ioctl) calls that the Kernel-based Virtual Machine (KVM) hypervisor supports.

For IDSs in virtualized environments featuring elastic resource provisioning, our approach for injecting hypercall attacks can be applied in combination with the attack detection accuracy metric and measurement methodology we propose. Our approach for injecting hypercall attacks, and our metric and measurement methodology, can also be applied independently beyond the scenarios considered in this thesis. The wide spectrum of security mechanisms in virtualized environments whose evaluation can directly benefit from the contributions of this thesis (e.g., hypervisor-based IDSs, IDSs deployed as VNFs, and AC mechanisms) reflects the practical implication of the thesis.

Zusammenfassung

Virtualisierung ermöglicht die Erstellung virtueller Instanzen physikalischer Geräte, wie z.B. Netzwerkgeräten und Prozessoren. In einem virtualisierten System (welches von einem Hypervisor kontrolliert wird), wird von virtuellen Maschinen (engl. virtual machine - VM) gemeinsam auf Ressourcen zugegriffen. Die Virtualisierung wird zunehmend als technische Möglichkeit in Betracht gezogen, um durch Serverkonsolidierung Kosten zu reduzieren und die Flexibilität physikalischer Infrastrukturen zu erhöhen. Auch wenn die Virtualisierung viele Vorteile bietet, so ergeben sich doch neue Herausforderungen im Bereich der IT-Sicherheit — ein Hypervisor bietet nämlich neuartige Angriffsflächen.

Bei der Angriffserkennung handelt es sich um einen weitverbreiteten IT-Sicherheitsmechanismus, mit welchem bosartige Aktivitäten in Rechnern oder Netzwerken identifiziert werden. So können Angriffe rechtzeitig gestoppt oder Sicherheitsverletzungen in ihrer Schwere gemindert werden. Als Folge der weiten Verbreitung von Virtualisierung ergibt sich der verstärkte Einsatz konventioneller, hard- oder softwarebasierter Angriffserkennungssysteme (engl. intrusion detection system - IDS) im Rahmen von dedizierten VMs als virtuelle Netzwerkfunktionen (engl. virtual network function - VNF). Zusätzlich wurden im Forschungs- und Industrieumfeld IDSs konkret für die Verwendung in virtualisierten Umgebungen entwickelt (d.h. hypervisor-basierte IDSs), die in Virtualisierungsebenen mit Komponenten innerhalb des Hypervisors bzw. innerhalb einer dedizierten VM eingesetzt werden. Letztere werden immer üblicher, weil sich die Anzahl der virtualisierten Rechenzentren kontinuierlich vermehrt und im Paradigma des Cloud-Computings die Virtualisierung eine Schlüsseltechnologie darstellt.

Um die Risiken durch Sicherheitsverletzungen zu minimieren, sind Methoden und Verfahren zur Bewertung eines IDS von zentraler Bedeutung. Zum Beispiel können unterschiedliche IDSs hinsichtlich ihrer Angriffserkennungsgenauigkeit verglichen werden. Dies hilft um das IDS zu identifizieren und einzusetzen, dessen Leistung als optimal zu bewerten ist. So vermindert sich das Risiko einer Sicherheitsverletzung. Jedoch fehlen Methoden bzw. Verfahren zur realistischen und präzisen Bewertung der Angriffserkennungsgenauigkeit von IDSs in virtualisierten Umgebungen (d.h. IDSs eingesetzt als VNFs oder hypervisor-basierte IDSs). Hierfür sind Arbeitslasten für die Sensoren von zu evaluierenden IDSs notwendig, die Angriffe auf den Hypervisor enthalten. Angriffe auf den Hypervisor sind sehr kritisch, weil sie z.B. Speicherinhalte eines Hypervisors so verändern können, dass dieser schädlichen Code mit erhöhten Privilegien ausführt. Ebenfalls existieren keine Metriken und Messmethodiken, mit denen die Angriffserkennungsgenauigkeit von IDSs in elastischen Umgebungen (d.h. bedarfsgerechte Zuweisungen von Hardware-Ressourcen zu VMs) präzise quantifiziert

werden kann. Bei modernen Hypervisoren können virtuelle CPUs sowie Speichereinheiten während des Betriebs an die dedizierte VM zugewiesen werden, in welcher die Angriffserkennung des IDSs ausgeführt wird. Die Zuweisung von Ressourcen im laufenden Betrieb ("Hotplugging") kann sich beträchtlich auf die Angriffserkennungsgenauigkeit von zu evaluierenden IDSs auswirken, was jedoch von existierenden Metriken nicht berücksichtigt wird. Dies hat ggf. ungenaue Messungen zur Folge, was sich entsprechend im Einsatz von fehlerhaft konfigurierten oder mängelbehafteten IDSs widerspiegelt und so das Risiko von Sicherheitsverletzungen erhöht.

Diese Arbeit präsentiert Beiträge, die die Standardkomponenten eines jeden Szenarios zur Systembewertung umfassen: Arbeitslasten, Metriken und Messmethodiken. Die wissenschaftlichen Beiträge dieser Arbeit sind:

- Eine umfassende Systematisierung der verwendeten Praktiken und des aktuelles Standes bei der Bewertung von IDSs. Die Systematisierung enthält: *(i)* die Definition eines Entwurfraumes für die IDS-Bewertung, welches praktische und theoretische Arbeiten im Bereich IDS-Bewertung systematisch in einen einheitlichen Kontext stellt; *(ii)* einen Überblick über verwendete Praktiken im Bereich IDS-Bewertung, der Ansätze und Methodiken jedes Teils des Entwurfraumes beinhaltet; *(iii)* und eine Sammlung an Fallstudien, die demonstriert, wie unterschiedliche IDS-Bewertungsansätze in der Praxis angewendet werden. Vor dem Hintergrund der beträchtlichen Menge bestehender praktischer und theoretischer Arbeiten im Bereich IDS-Bewertung erweist sich die Systematisierung als vorteilhaft zur Verbesserung des allgemeinen Themenverständnisses, indem ein Überblick zur aktuellen Sachlage des Themengebietes geliefert wird. Zusätzlich ist dies vorteilhaft bei der Identifizierung und Gegenüberstellung von Vor- und Nachteilen unterschiedlicher IDS-Bewertungsmethodiken und -praktiken. Es hilft ebenfalls Vorgaben und Empfehlungen für die Bewertung gegenwärtiger wie auch zukünftiger IDSs zu identifizieren.
- Eine detaillierte Analyse von Schwachstellen von Hypervisoren wird präsentiert, sowie eine Menge von Angriffsmodellen, die die Aktivitäten eines Angreifers umfassen, der diese Schwachstellen auslöst. Diese Analyse umfasst 35 Schwachstellen in Hypercall-Routinen, sogenannte Hypercall-Schwachstellen. Hypercalls sind an den Hypervisor gerichtete „Software-Traps" aus dem Betriebssystemkern einer VM. Die Hypercall-Schnittstelle von Hypervisoren ist — neben Gerätetreibern und „VM exit"-Ereignissen — eine ihrer Angriffsflächen. Wird die gegenüber einem Hypercall bestehende Schwachstelle ausgenutzt, kann dies zu einem Absturz des Hypervisors oder zu einer Änderung seines Speicherinhalts führen. Wir analysieren die Gründe der betrachteten Hypercall-Schwachstellen, demonstrieren und analysieren Angriffe, die solche Schwachstellen ausnutzen (d.h. Hypercall-Angriffe), entwickeln Hypercall-Angriffsmodelle (nämlich systematisierte, auf die Schnittstelle der Hypercalls gerichtete Aktivitäten der Angreifer) und diskutieren zukünftige Forschungsrichtungen, die Ansätze betrachten, um die Schnittstellen von Hypercalls abzusichern.
- Ein neuartiger Ansatz zur Bewertung von IDSs, der die Generierung von Arbeitslasten ermöglichen, die Hypercall-Angriffe enthalten. Wir schlagen einen

Ansatz zur Bewertung von IDSs durch die Injektion von Angriffen (d.h. Hypercall-Angriffen) vor. Es handelt sich hier um die kontrollierte Ausführung von Angriffen in einer regulären Systemumgebung, in welcher das betrachtete IDS eingesetzt wird. Die Injektion von Angriffen folgt Angriffsmodellen, die durch Analyse realistischer Angriffe erstellt wurden. Wir verwenden die als Teil dieser Arbeit dargestellten Hypercall-Angriffsmodelle zur Injektion von Hypercall-Angriffen.

- Eine neuartige Metrik und Messmethodik zur präzisen Quantifizierung der Angriffserkennungsgenauigkeit von IDSs in virtualisierten elastischen Umgebungen. Wir demonstrieren, wie die Elastizität virtualisierter Umgebungen sich auf die Angriffserkennungsgenauigkeit von IDSs auswirkt und zeigen, dass die Verwendung existierender Metriken zu schwierigen und ungenauen Messungen bei der Bewertung von IDSs in virtualisierten elastischen Umgebungen führen. Ausserdem zeigen wir den praktischen Nutzen der von uns vorgeschlagenen Metrik in mehreren Fallstudien.

Zusammenfassend präsentiert diese Arbeit die erste Systematisierung des Stands der Technik bei der Bewertung von IDSs unter Beachtung der Arbeitslasten, Metriken und Messmethodiken als integraler Teil eines jeden Ansatzes zur IDS Bewertung. Außerdem sind wir die ersten, die Hypercall-Angriffsflächen im Detail untersuchen und die einen Ansatz zur Bewertung von IDSs in virtualisierten Umgebungen durch die Injektion von Angriffen vorschlagen. Abschließend präsentiert diese Arbeit die erste Metrik und Messmethodik zur Quantifizierung der Angriffserkennungsgenauigkeit von IDSs in virtualisierten elastischen Umgebungen.

Aus technischer Sicht präsentieren wir in dieser Arbeit, als Teil des vorgeschlagenen Ansatzes zur Bewertung von IDSs, ein Werkzeug mit der Bezeichnung „hInjector“, welches zur Injektion von Hypercall-Angriffen dient. Dieses Werkzeug wurde entworfen, um die gründliche, repräsentative und praktisch umsetzbare Bewertung von IDSs per Injektion von Angriffen zu ermöglichen. Wir demonstrieren die Anwendung und den praktischen Wert sowohl von hInjector als auch des vorgeschlagenen Ansatzes durch die Bewertung eines repräsentativen, hypervisor-basierten IDS, das zur Erkennung von Hypercall-Angriffen konzipiert ist. Während wir uns auf die Bewertung der Fähigkeiten von IDSs zur Erkennung von Hypercall-Angriffen fokusieren, kann der vorgeschlagene Ansatz verallgemeinert und in einem breiteren Kontext angewendet werden. Zum Beispiel kann er direkt verwendet werden, um auch Hypervisor-Sicherheitsmechanismen, nämlich etwa Hypercall-Zugangskontrollmechanismen, zu bewerten. Der Ansatz kann auch angewendet werden für die Bewertung von IDSs, die der Erkennung von Angriffen basierend auf Operationen dienen, die eine funktionelle Ähnlichkeit zu Hypercalls aufweisen. Solche Operationen sind z.B. die “input/output control (ioctl)” Aufrufe, die vom “Kernel-based Virtual Machine (KVM)”-Hypervisor unterstützt werden.

Für IDSs, die in elastischen virtualisierten Umgebungen eingesetzt werden, kann unser Ansatz zur Injektion von Hypercall-Angriffen in Verbindung mit der von uns vorgeschlagenen Metrik und Messmethodik angewendet werden. Beide können auch unabhängig von den in dieser Arbeit betrachteten Szenarien angewendet werden. Das

breite Spektrum von Sicherheitsmechanismen (z.B. hypervisor-basierte IDSs, IDSs eingesetzt als VNFs und Zugangskontrollmechanismen), deren Bewertung von den Beiträgen dieser Arbeit profitieren, spiegelt ihre Praktikabilität wider.

Chapter 1

Introduction

1.1 Motivation

Virtualization is a concept of the 1960's allowing the creation of logical ("virtual") instances of physical devices, such as networks, storage or processing units. In recent years, virtualization has received increasing interest, both from industry and academia, as a way to reduce costs through server consolidation and to enhance the flexibility of physical infrastructures. In a virtualized system, governed by a hypervisor, resources such as processor time, disk capacity, and network bandwidth are shared among virtual machines (VMs). Each VM accesses the physical resources of the infrastructure through the hypervisor and is entitled to a predefined fraction of capacity. Modern hypervisors also provide mechanisms for elastic resource provisioning allowing to adapt the system to workload variations such as load spikes. Under elastic resource provisioning (which we also refer to as *elasticity*), we understand on-demand provisioning (i.e., allocation or deallocation) of virtualized hardware resources to VMs.

While server consolidation through virtualization provides many benefits, it also introduces some new challenges; that is, the introduction of a hypervisor and the allocation of potentially multiple VMs on a single physical server are additional critical aspects introducing new potential threats and vulnerabilities [Abh15], [PBSL13]. For instance, Gens et al. [GMV+10] report that security is a major concern for users of modern virtualized service infrastructures, followed by availability and performance. Some critical security issues include data integrity, authentication, application security, and so on [SK11]. In addition, attackers are actively exploring virtualization-specific attack surfaces such as hypervisors.

A common defensive instrument against security threats are intrusion detection systems (IDSs). They monitor on-going activities in the protected network(s) and host(s), detecting potentially malicious activities. The detection of malicious activities enables the timely reaction in order to stop an on-going attack, or to mitigate the impact of a security breach.

The adoption of virtualization technology has lead to the emergence of novel IDSs specifically designed to operate in virtualized environments (i.e., *hypervisor-based IDSs*) such as AdjointVM [Kon11], VMFence [JXZ+11], and Advanced Cloud Protection System (ACPS) [LDP11]. Such IDSs typically perform host intrusion detection and are deployed in the virtualization layer, with components both inside the hypervisor and

in a designated VM, which has several benefits [MKA+13]. Hypervisor-based IDSs can monitor the network and/or host activities of all guest VMs at the same time.[1] Further, they are isolated from, and transparent to, malicious users of the guest VMs since they do not operate inside the guest VMs, but instead leverage functionalities of the underlying hypervisor. In addition, some hypervisor-based IDSs can also detect attacks specifically targeted at the hypervisor. Hypervisor-based IDSs are becoming increasingly common with the growing proliferation of virtualized data centers and the advent of the cloud computing paradigm, for which virtualization is as a key enabling technology. Intrusion detection in cloud environments has been recently receiving increasing attention, given that security concerns are still one of the greatest showstoppers for the wide adoption of cloud computing [GMV+10].

The increasing adoption of virtualization has resulted in the practice of deploying conventional IDSs (e.g., hardware IDS appliances or common software-based IDSs) in designated VMs as virtual network functions (VNFs). For instance, a network-based IDS (e.g., Snort [Roe99]) may be deployed in a designated VM and configured to tap into the physical network interface card used by all VMs. Thus, the IDS can monitor the network activities of all VMs at the same time while being isolated from, and transparent to, their users. Further, in comparison to deploying hardware IDS appliances, which are expensive and challenging to manage, deploying IDSs as VNFs is cost-effective and the management of such IDSs is easier. It is important to emphasize that the network function virtualization technology introduces new security risks that have not yet been investigated in detail. We refer the reader to [Ale16] for more information.

To minimize the risk of security breaches, methods and techniques for evaluating the performance of IDSs in a realistic and reliable manner are needed. The benefits of IDS evaluation are manyfold. For instance, one may compare different IDSs in terms of their attack detection accuracy in order to deploy an IDS that operates optimally in a given environment, thus reducing the risks of a security breach. Further, one may tune an already deployed IDS by varying its configuration parameters and investigating their influence through evaluation tests. This enables comparison of the evaluation results with respect to the configuration space of the IDS and can help to identify an optimal configuration. IDS evaluation is of interest to many different types of users and professionals in the field of communication systems and information security. This includes researchers, who typically evaluate novel IDS algorithms and architectures with respect to specific IDS properties that are subject of research; industrial software architects, who typically evaluate IDSs by carrying out internationally standardized large scale tests; and IT security officers, who evaluate IDSs in order to select an IDS that is optimal for protecting a given environment, or to optimize the configuration of an already deployed IDS. We discuss more on application scenarios of IDS evaluation in Section 2.2.1.

IDS evaluation, in general, has proven to be a challenging task riddled with many

[1] In this thesis, we use the terms *guest VM* and *VM* interchangeably. We use the term *host VM* to explicitly refer to a VM that has higher privileges than the other VMs co-located with it and is used for managing (i.e., administering) the virtualized environment where it resides.

difficulties, such as the lack of realistic evaluation data, flawed methodologies, and many more. Many of these challenges have been subject of existing work in the research community, e.g., [SYB04], [Ran01], [McH00]. However, to the best of our knowledge, no approaches, methods, and tools for the evaluation of IDSs in virtualized environments (i.e., hypervisor-based IDSs or IDSs deployed as VNFs) currently exist. We argue that conventional approaches to IDS evaluation do not satisfy the requirements to enable rigorous and representative evaluation of IDSs in virtualized environments.

In this thesis, we focus on IDS evaluation requirements in the context of virtualized environments, considering the following requirements with respect to the standard components — *workloads*, *metrics*, and *measurement methodologies* — that comprise any system evaluation scenario:

- the use of workloads that contain *virtualization-specific attacks*, that is, *attacks initiated from malicious guest VMs and targeting the underlying hypervisors* — we argue that such workloads are needed for testing IDSs that have the functionality to detect attacks targeting hypervisors;
- the use of metrics and measurement methodologies for measuring the attack detection accuracy of IDSs taking *elasticity*, a feature of modern virtualized infrastructures, into account — we argue that such metrics and methodologies are needed for the accurate measurement of the attack detection accuracy of IDSs in virtualized environments (i.e., hypervisor-based IDSs or IDSs deployed as VNFs).

1.2 Problem Statement: Shortcomings of Existing Approaches

1.2.1 Workloads

As we mentioned in Section 1.1, workloads that contain virtualization-specific attacks, that is, attacks initiated from malicious VMs and targeting the underlying hypervisors, are needed. Such workloads are used to exercise the sensors of evaluated IDSs that have the functionality to detect attacks targeting hypervisors. Many hypervisor-based IDSs, such as Collabra [BSNS11a] and Xenini [MM11], and some conventional IDSs deployed as VNFs, such as Open Source Security (OSSEC) [oss], have the functionality to detect attacks targeting hypervisors; that is, OSSEC can be configured to analyze log files produced by a hypervisor (e.g., by Xen [xena]) and detect attacks targeting the hypervisor.

Attacks targeting hypervisors are of high severity since they may result in crashing a hypervisor including all VMs running on top of it. They may also result in altering the hypervisors's memory, which enables the execution of malicious code with hypervisor privileges. The lack of appropriate workloads for the evaluation of IDSs that have the functionality to detect attacks targeting hypervisors can lead to the deployment of

IDSs that do not operate optimally (e.g., exhibit low attack detection accuracy). This increases the risk of severe security breaches in virtualized environments.

When it comes to evaluating an IDS, one needs *malicious* workloads (i.e., workloads that contain attacks) and *benign* (regular, normal) workloads (i.e., workloads that do not contain attacks). Malicious workloads are used to subject an IDS under test to attack scenarios (as done by Reeves et al. [RRL+12] and Gornitz et al. [GKRB09]). Benign workloads are used, for example, to evaluate the monitoring performance overhead or the capacity of an IDS (as done by Bharadwaja et al. [BSNS11a]). Workloads normally take an *executable form* or a *trace form* (traces generated by recording the execution of activities for later replay). We now review approaches for the evaluation of IDSs using malicious workloads, which are in the focus of this thesis (see Section 1.1).

Malicious workloads in executable form can be obtained from *exploit databases* containing attack scripts. One has a choice of assembling an exploit database by himself or using a readily available one. A major disadvantage of the manual assembly is the high cost of the attack script collection process. Locating the attack scripts needed for triggering specific vulnerabilities and obtaining the required vulnerable software is normally time-consuming. In addition, once the needed attack scripts are found, they typically have to be customized for the specific target environment. To assemble an exploit database, IDS evaluators normally obtain attack scripts from public exploit repositories, such as 1337day [inj], Exploit database [exp], Packetstorm [pst], SecuriTeam [seca], and Securityfocus [secb]. Alternatively, IDS evaluators may employ a penetration testing tool as a readily available exploit database. The Metasploit framework [PtsM] has been extensively used for evaluating IDSs (see, for example, Gornitz et al. [GKRB09] and Nasr et al. [NKF12]). At the time of writing, the popular exploit repositories and penetration testing tools do not contain, or contain a very small number of, attacks targeting hypervisors. This makes them unsuitable for evaluating IDSs that have the functionality to detect such attacks.

As an alternative approach to using an exploit database, one can use the *attack injection* technique to generate malicious workloads in executable form. Attack injection is the controlled execution of attacks during regular operation of the environment where an IDS under test is deployed. This technique enables IDS testing by attacking the target platform with respect to attack models or by executing vulnerable code injected in the platform (see, for example, Fonseca et al. [FVM09]). Attack injection is typically used in cases where the collection of attack scripts is unfeasible. The application of this technique for generating workloads that contain attacks targeting hypervisors has not been investigated.

Malicious workloads in trace form can be generated by *acquiring* or *generating* traces. Real-world production traces can be acquired from proprietary organizations. Such traces subject an IDS under test to a workload as observed during operation in a real production environment. However, they are usually very difficult to obtain mainly due to the unwillingness of industrial organizations to share operational traces.

In contrast to proprietary traces, one can acquire publicly available traces without any legal constraints. However, publicly available traces often contain errors and

they quickly become outdated after their public release since the recorded attacks have limited shelf-life. The most frequently used publicly available traces (see, for example, Alserhani et al. [AAA+10], Yu et al. [YD11], and Raja et al. [RAR12]) are the Defense Advanced Research Projects Agency (DARPA) [LHF+00] [IDE] and the derived Knowledge Discovery and Data Mining (KDD) Cup 99 [UoC] datasets. IDS evaluators may also use publicly available traces from trace repositories such as Cooperative Association for Internet Data Analysis (CAIDA) [cai], Defense Readiness Condition (DEFCON) [CtCtF], Internet Traffic Archive (ITA) [ita], and Lawrence Berkeley National Laboratory/International Computer Science Institute (LBNL/ISCI) [lbn]. None of the commonly used publicly available traces contain attacks targeting hypervisors.

IDS evaluators may generate traces in a testbed environment or deploy a honeypot in order to capture malicious activities. Generating traces in a testbed environment may be done by using the previously mentioned methods to generate workloads in executable form whereby the executed workloads are captured and stored in trace files. The generation of traces in a testbed environment is challenging since the costs of building a testbed that scales to realistic production environments may be high and the used trace generation method may produce faulty workloads.

Honeypots enable the recording of host and/or network malicious activities performed by an attacker without revealing their purpose. By mimicking real systems and vulnerable services, honeypots record the interaction between the attack target and the attack itself. Security researchers often use the honeyd [hon] honeypot, which is equipped with many logging and log processing utilities. We are not aware of honeypots that are able to record the interaction between a VM and the underlying hypervisor, that is, of honeypots that are able to record attacks targeting hypervisors.

To summarize, the application of the above approaches for the generation of workloads that contain attacks targeting hypervisors has not been investigated. Given the potential severity of these attacks, the lack of appropriate workloads for the evaluation of IDSs that have the functionality to detect attacks targeting hypervisors is a critical issue — it can lead to the deployment of IDSs that do not operate optimally, which increases the risk of severe security breaches in virtualized environments.

1.2.2 Metrics and Measurement Methodologies

We distinguish between two categories of IDS evaluation metrics: *performance-related* and *security-related* metrics. Performance-related metrics are metrics that quantify non-functional properties of an IDS under test, such as capacity, performance overhead, and resource consumption. For instance, Meng et al. [ML12] consider workload processing throughput, Lombardi et al. [LDP11] consider performance overhead, Mohammed et al. [MOL+11] consider power consumption, and Sinha et al. [SJP06] consider memory consumption. Security-related metrics are metrics that quantify attack detection properties of an IDS under test (e.g., attack detection accuracy). Next, we focus on security-related metrics, which are in the focus of this thesis (see Section 1.1).

A common aspect of all existing security-related metrics is that they are defined with

respect to a *fixed* set of hardware resources available to the IDS under test [MK12]. Mell et al. [MHL+03] and Hall et al. [HW02] confirm that the values of existing IDS evaluation metrics express the attack detection accuracy of an IDS only for a specific hardware environment in which the IDS is expected to reside during operation. However, many virtualized infrastructures support *elastic* resource provisioning; that is, resources can be provisioned and used by the IDS *on-demand* during operation [HKR13]. For instance, the Xen and VMware virtualization platforms allow for hotplugging virtual CPUs (vCPUs) and memory on the designated VM where the intrusion detection engine of hypervisor-based IDSs, or conventional IDSs deployed as VNFs, typically operates (see Section 1.1). This may have a significant impact on many properties of the evaluated IDS, including its attack detection accuracy.

Based on the above, we argue that the use of conventional metrics (i.e., existing IDS attack detection accuracy metrics that do not take elasticity of virtualized environments into account) may lead to inaccurate measurements in cases where the on-demand provisioning of resources to an IDS under test has significant impact on its attack detection accuracy. This, in turn, may result in the deployment of misconfigured or ill-performing IDSs in production environments, increasing the risk of security breaches. We argue that novel metrics and measurement methodologies for measuring the attack detection accuracy of IDSs in virtualized environments featuring elasticity are needed. Such metrics and methodologies should take into account the behavior of the IDS under test as its operational environment changes. Next, we provide a compact overview of conventional security-related metrics commonly used in practice.

We distinguish between *basic* and *composite* security-related metrics. Basic metrics are the true positive rate, the false positive rate, the positive predictive value, and the negative predictive value. These metrics quantify various individual attack detection properties. The true positive rate quantifies the probability that an alert generated by an IDS is really an intrusion. The false positive rate quantifies the probability that an alert generated by an IDS is not an intrusion, but a regular benign activity. The respective complementary metrics (i.e., the true negative rate and the false negative rate) are also relevant. The positive predictive value (PPV) quantifies the probability that there is an intrusion when an IDS generates an alert. The negative predictive value (NPV) quantifies the probability that there is no intrusion when an IDS does not generate an alert.

Although the above IDS properties are quantified individually, they need to be analyzed together in order to accurately characterize the attack detection accuracy of a given IDS. In order to analyze relationships between basic metrics, IDS evaluators typically combine basic metrics into composite metrics. Such an analysis is typically used to discover an optimal IDS operating point — an IDS configuration that yields optimal values of both the true and false positive detection rate — or to compare multiple IDSs. It is a common practice to use Receiver Operating Characteristic (ROC) curve to investigate the relationship between the measured true positive and false positive detection rates of an IDS. A ROC curve is a two-dimensional depiction of the accuracy of a detector as it plots the true positive rate against the corresponding false positive rate [MR04].

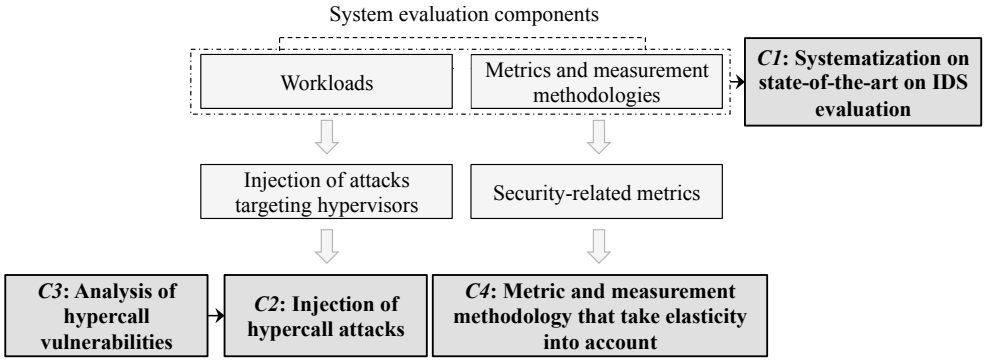


Figure 1.1: Focus and contributions of this thesis.

Security researchers have proposed metrics that are more accurate and expressive than ROC curves. They can be classified into two main categories: metrics that use *cost-based* measurement methodologies and metrics that use *information-theory* measurement methodologies. Two of the most prominent metrics that belong to these categories are the expected cost metric proposed by Gaffney et al. [GU01] and the intrusion detection capability metric proposed by Gu et al. [GFD+06]. The expected cost metric uses a cost model to quantify the cost of the operation of an IDS under test. The intrusion detection capability metric aims to quantify the attack detection accuracy of an IDS in an objective manner by modeling the input to, and output of, the evaluated IDS as a stream of random variables. The latter enables the quantification of the attack detection accuracy as the reduction of the uncertainty of the IDS input after the IDS output is known.

To summarize, all of the basic and composite security-related metrics discussed above are defined with respect to a fixed set of hardware resources available to the IDS under test, that is, they do not take elastic resource provisioning into account [MHL+03], [HW02]. Therefore, using these metrics for evaluating IDSs deployed in virtualized environments may lead to inaccurate measurements. We argue that novel metrics and measurement methodologies, which take into account the behavior of an IDS as its operational environment changes, are needed. Such metrics and methodologies would allow to quantify the attack detection accuracy of IDSs deployed in virtualized environments in an accurate manner. In addition, they would allow to quantify the ability of the IDS to scale its attack detection efficiency as resources are allocated to it, or deallocated from it, during operation.

1.3 Contributions of this Thesis

In Figure 1.1, we depict an overview of the focus and contributions of this thesis. The contributions (marked in bold in Figure 1.1) can be classified according to the standard components of any system evaluation scenario: workloads, metrics, and measurement

methodologies. They can be further divided into scientific and technical contributions. The scientific contributions are:

(i) ***Contribution 1*** *(C1, see Figure 1.1):* a comprehensive systematization of the common practices and the state-of-the-art on IDS evaluation including:

- *definition of an IDS evaluation design space* allowing to put existing practical and theoretical work into a common context in a systematic manner;
- *overview of common practices* in IDS evaluation reviewing existing evaluation approaches and methods related to each part of the design space;
- a set of *case studies* demonstrating how different IDS evaluation approaches are applied in practice.

Given the significant amount of existing practical and theoretical work related to IDS evaluation, the presented systematization is benefitial for improving the general understanding of the topic by providing an overview of the current state of the field. In addition, it is beneficial for identifying and contrasting advantages and disadvantages of different IDS evaluation methods and practices, while also helping to identify specific requirements and best practices for evaluating current and future IDSs.

(ii) ***Contribution 2*** *(C2, see Figure 1.1):* a novel approach for evaluating IDSs enabling the generation of workloads that contain attacks targeting hypervisors, that is, hypercall attacks. Hypercalls are software traps from a kernel of a semi- or fully paravirtualized guest VM to the hypervisor. They can, for example, enable intrusion into a vulnerable hypervisor, initiated from a malicious VM kernel, through the hypervisor's hypercall interface. The triggering of a vulnerability of a hypercall handler (i.e., a hypercall vulnerability) may crash the hypervisor or lead to altering the hypervisor's memory. This enables the execution of malicious code with hypervisor privileges (see the work of Rutkowska et al. [RW]). In the context of this thesis, under hypercall attack, we understand any malicious hypercall activity.

In this thesis, we propose an approach for evaluating IDSs using attack injection (i.e., injection of hypercall attacks, see Section 1.2.1). We focus on attack injection as an approach for generating malicious workloads that contain hypercall attacks since the collection of attack scripts that demonstrate such attacks is unfeasible, that is, publicly available scripts that demonstrate hypercall attacks are very rare [MPA+14], [HL09].

Workloads that contain hypercall attacks are a key requirement for evaluating the attack detection accuracy of IDSs designed to detect hypercall attacks. Such workloads are needed to exercise the sensors of an IDS monitoring the hypercall interface of a hypervisor. The research and industrial communities have developed security mechanisms that can detect hypercall attacks. These include IDSs that can be configured to analyze in real-time log files with information on executed hypercalls, such as Xenini [MM11] and the de-facto standard host-based IDS OSSEC [oss], as well as access control (AC) systems, such as Xen Security Modules - Flux Advanced Security Kernel (XSM-FLASK) distributed with the Xen hypervisor. Given the potential sever-

ity of hypercall attacks, the rigorous evaluation of IDSs designed to detect hypercall attacks using workloads that contain such attacks is crucial for preventing high-impact breaches in virtualized environments.

The approach we propose may be applied conceptually not only for evaluating IDSs designed to detect hypercall attacks, but also attacks involving the execution of operations that are functionally similar to hypercalls. Such operations are, for example, the input/output control (ioctl) calls that the Kernel-based Virtual Machine (KVM) hypervisor supports. By enabling the generation of workloads that contain hypercall attacks, this thesis contributes towards addressing the issue of the lack of IDS evaluation workloads that contain virtualization-specific attacks (see Section 1.2.1).

For the injection of realistic hypercall attacks, representative hypercall attack models are required (see Section 1.2.1). Note that the injection of attacks is performed with respect to attack models constructed by analysing realistic attacks. Attack models are systematized activities of attackers targeting a given attack surface. Publicly disclosed reports describing hypercall vulnerabilities (e.g., CVE-2013-4494, CVE-2013-3898) are currently the only available source of information, however, they only provide high-level descriptions. As a result, the characterization of hypercall vulnerabilities and hypercall attacks is challenging. It warrants a detailed investigation of existing vulnerabilities, which can only be done by reverse-engineering released patches fixing the vulnerabilities. The latter is crucial for the construction of representative attack models, which, in turn, are a prerequisite for the injection of realistic hypercall attacks. This brings us to the next contribution of this thesis.

(iii) ***Contribution 3*** *(C3, see Figure 1.1):* an in-depth analysis of common vulnerabilities of modern hypervisors, as well as a set of attack models capturing the activities of attackers triggering these vulnerabilities. The analysis includes 35 representative vulnerabilities of hypercall handlers discovered by searching major vulnerability report databases (e.g., cvedetails [CVEj]). We discuss issues, challenges, and gaps that apply specifically to securing hypercall interfaces. Our analysis is based on information obtained by reverse engineering released patches fixing the considered vulnerabilities. More specifically, this thesis contributes:

- a comprehensive analysis and systematization of the origins of the considered hypercall vulnerabilities,
- a demonstration of possible attacks triggering the hypercall vulnerabilities and evaluation of their effects,
- a set of hypercall attack models based on an in-depth analysis of the activities for executing hypercall attacks, and
- a discussion of future research directions focusing on both proactive and reactive approaches for securing hypercall interfaces.

To the best of our knowledge, there is no previous work examining the hypercall attack surface in detail.

(iv) ***Contribution 4*** *(C4, see Figure 1.1):* a novel metric and measurement methodology

for quantifying the attack detection accuracy of IDSs in virtualized environments that feature elastic resource provisioning; that is, a metric and measurement methodology that take the elasticity aspects of virtualized environments into account. More specifically, this thesis:

- demonstrates how the elasticity of resource allocations in virtualized environments may impact the IDS attack detection accuracy;
- shows that using conventional IDS evaluation metrics in such environments may lead to practically challenging and inaccurate measurements;
- proposes and demonstrates the practical use of a novel metric and measurement methodology that allow for quantifying the impact of elasticity on the IDS attack detection accuracy.

We designed the new metric with respect to a set of specific criteria for accurate and practically feasible IDS evaluation. Our metric is meant to complement conventional metrics — it is specifically designed for evaluating IDSs that perform run-time monitoring and operate in virtualized environments with elastic resource provisioning. The metric can be used to quantify the attack detection accuracy of such IDSs. This enables the identification and deployment of optimally performing IDSs, thus reducing the risk of security breaches in virtualized environments.

From a technical perspective, as part of the proposed approach for evaluating IDSs in virtualized environments, this thesis presents *hInjector*, an open-source tool for injecting hypercall attacks. We designed hInjector to achieve the challenging goal of satisfying the following key requirements for the rigorous, representative, and practically feasible evaluation of IDSs in virtualized environments: injection of realistic attacks, during regular system operation, and in a non-disruptive manner (e.g., prevention of potential crashes due to the injected attacks).

We demonstrate in this thesis the application of the proposed approach for evaluating IDSs and the practical usefulness of hInjector by evaluating Xenini [MM11], a representative IDS designed to detect hypercall attacks. We inject realistic attacks triggering publicly disclosed hypercall vulnerabilities (e.g., CVE-2012-5525 [CVEg], CVE-2012-3495 [CVEb], and CVE-2012-5513 [CVEf]) as well as specifically crafted evasive attacks — attacks specifically crafted to not be easily detectable by an IDS. We extensively evaluate Xenini considering multiple alternative configurations of the IDS, that is, varying the sensitiveness of the IDS for labeling a given activity as malicious. We calculate values of attack detection accuracy metrics, such as true and false positive rate, and plot ROC curves (see Section 1.2.2). The obtained results match the expected behavior of Xenini (e.g., the more sensitive the IDS, the higher true and false positive rates it exhibits), which shows the practical usefulness of the approach we propose. Our approach is the first to enable an extensive evaluation at this level of detail and accuracy.

While we focus on evaluating the capabilities of IDSs to detect hypercall attacks, the proposed IDS evaluation approach can be generalized and applied in a broader context. For example, it may be directly used to also evaluate security mechanisms of

hypervisors, such as hypercall access control (AC) mechanisms. It may also be applied to evaluate the capabilities of IDSs to detect attacks involving operations that are functionally similar to hypercalls, for example, the ioctl calls that the KVM hypervisor supports.

We also demonstrate in this thesis the practical use of the metric and measurement methodology we propose by evaluating two conventional IDSs (i.e., Snort [Roe99] and Suricata [sur]) deployed as VNFs running on top of the Xen hypervisor. We consider 15 different configurations of the IDSs and the hypervisor performing elastic resource provisioning. The obtained metric values match the expected behavior of the metric with respect to the criteria according to which we designed the metric. This shows the accuracy and practical usefulness of the metric and measurement methodology we propose.

For IDSs in virtualized environments featuring elastic resource provisioning, our approach for injecting hypercall attacks can be applied in combination with the attack detection accuracy metric and measurement methodology we propose. Our approach for injecting hypercall attacks, and our metric and measurement methodology, can also be applied independently beyond the scenarios considered in this thesis. The wide spectrum of security mechanisms in virtualized environments whose evaluation can directly benefit from the contributions of this thesis (e.g., hypervisor-based IDSs, IDSs deployed as VNFs, and AC mechanisms) reflects the practical implication of the thesis.

1.4 Outline

This thesis is structured into seven main chapters, including the introductory chapter (Chapter 1), and one appendix chapter.

In Chapter 2, we provide the background that is essential for understanding the topic of IDS evaluation. We discuss several types of attacks that we refer to throughout the thesis, and the role of intrusion detection in relation to other security mechanisms. In addition, we define and discuss different types of IDSs. Further, we provide an overview of major developments in the area of IDS evaluation in a chronological manner. Finally, we demonstrate the wide applicability and relevance of IDS evaluation by discussing various application scenarios.

In Chapter 3, we analyze the current state of IDS evaluation. To this end, we define an IDS evaluation design space structured into three parts — workloads, metrics, and measurement methodology — which are considered as the standard components of any evaluation experiment. We systematize and review different approaches for generating or obtaining workloads for evaluating IDSs. Further, we systematize and review IDS evaluation metrics, and we demonstrate the use of these metrics for comparing IDSs. In addition, we systematize IDS properties that are typically evaluated in practice. We also discuss and practically demonstrate the respective measurement methodologies. Finally, we discuss challenges that apply to evaluating IDSs specifically designed for deployment and operation in virtualized environments (i.e., hypervisor-based IDSs)

and we present guidelines for planning IDS evaluation studies based on the lessons learned.

In Chapter 4, we characterize the hypercall attack surface based on analyzing a set of vulnerabilities of hypercall handlers. We systematize and discuss the errors that caused the considered vulnerabilities, and activities for executing attacks triggering them. We also demonstrate attacks triggering the considered vulnerabilities and analyze their effects. Finally, we suggest an action plan for improving the security of hypercall interfaces.

In Chapter 5, we propose a novel approach for the rigorous evaluation of IDSs in virtualized environments, with a focus on IDSs designed to detect attacks leveraging or targeting the hypervisor via its hypercall interface. We present hInjector, a tool for generating IDS evaluation workloads by injecting such attacks during regular operation. We demonstrate the application of our approach and show its practical usefulness by evaluating a representative hypervisor-based IDS designed to detect hypercall attacks. The virtualized environment of the industry-standard benchmark SPECvirt_sc2013 [spea] is used as a testbed, whose drivers generate workloads representative of real-life workloads.

In Chapter 6, we demonstrate the impact of elasticity on IDS attack detection accuracy. In addition, we propose a novel metric and measurement methodology for accurately quantifying the accuracy of IDSs in virtualized environments featuring elasticity. We demonstrate their practical use through case studies involving commonly used IDSs.

Chapter 7 concludes this thesis by summarizing its contributions and gives an outlook on future research.

In Chapter A, we provide in-depth technical information on publicly disclosed vulnerabilities of hypercall handlers, that is, on a selected representative subset of the vulnerabilities we consider in Chapter 4. Our vulnerability analysis is based on reverse engineering the released patches that fix the considered vulnerabilities. For each analyzed vulnerability, we provide background information essential for understanding the vulnerability, and information on the vulnerable hypercall handler and the error causing the vulnerability. We also show how the vulnerability can be triggered and discuss the state of the targeted hypervisor after the vulnerability has been triggered.

Chapter 2

Foundations

2.1 Intrusion Detection Systems

2.1.1 Attacks and Common Security Mechanisms

A given system (i.e., a host) is considered as secure if it has the properties of confidentiality, integrity, and availability of its data and services [Sta02], commonly known as the CIA triad. Under confidentiality, we understand the protection of data against its release to unauthorized parties. Under integrity, we understand the protection of data and/or services against modifications by unauthorized parties. Under availability, we understand the protection of services such that they are ready to be used when needed.

Attacks are deliberate attempts to violate the previously mentioned security properties [Shi99]. There are many different types of attacks with respect to various attack properties. Security researchers have developed many attack categorization schemes (i.e., attack taxonomies) designed for different purposes. For instance, Nasr et al. [NAEKF11] propose an attack taxonomy useful for classifying attacks used in intrusion detection system (IDS) evaluation studies, while Hansman et al. [HH05] define an attack taxonomy for general use.

In this section, for the sake of completeness, we discuss only the types of attacks that we refer to in the rest of the thesis when discussing IDS evaluation approaches. We stress that we do not aim to provide an extensive coverage of attack types as that is out of the scope of this thesis. We also stress that some attack types, although relevant, are out of the scope of IDSs, as confirmed by international standards (see National Institute of Standards and Technology (NIST)'s guide to IDSs [SM07]). Such are, for example, spamming and information fishing attacks.

According to the source of execution of an attack from the perspective of the targeted system, we distinguish between *remote* and *local* attacks. A *remote attack* is an attack that targets a service of a system available over a network and is carried out over a network connection, i.e., the Internet or a local area network (LAN) connection, between the attacker and the targeted service. An example of a remote attack is an Standard Query Language (SQL) injection attack. When executing such an attack, an attacker normally inserts a malicious SQL query into an entry field of a database-driven web application and executes it. This leads to, for example, obtaining access to sensitive data stored

in the database (e.g., passwords), deleting or adding database records, and so on. For detailed information on SQL injection attacks, we refer the reader to the work of Halfond et al. [HVO06]. In Figure 2.1a, we depict a scenario where an SQL injection attack is executed against a database-driven web application with the goal of obtaining sensitive data.

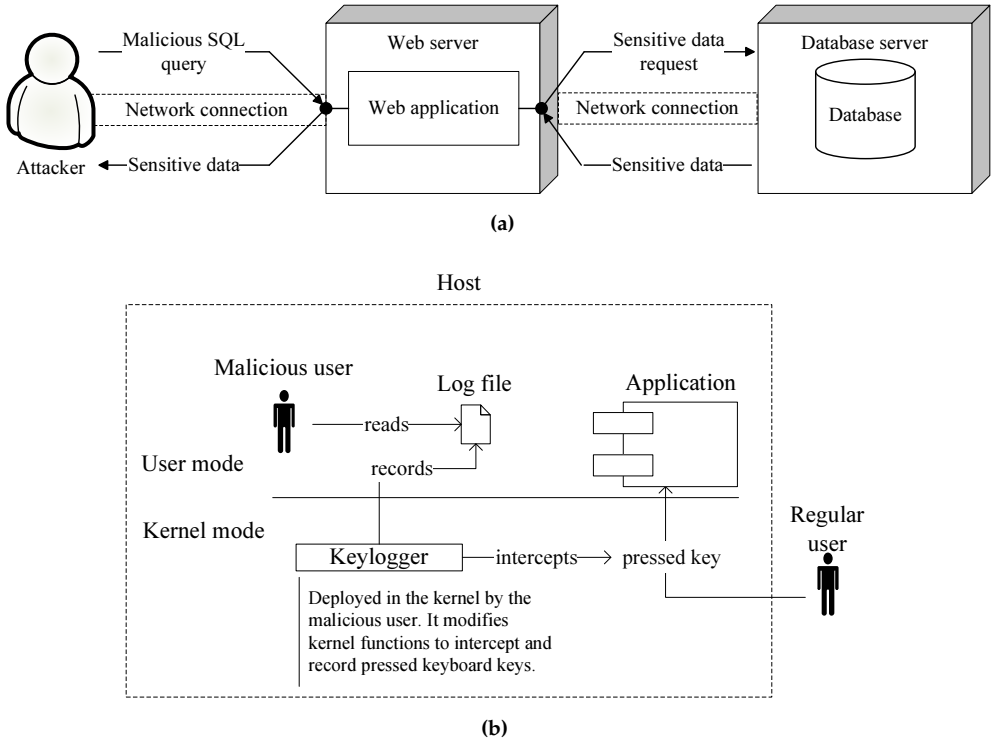


Figure 2.1: An example of a (a) remote attack - SQL injection attack, and (b) local attack - deployment of a keylogger.

Local attacks are executed by users of the targeted system itself, which results, for example, in privilege escalation or unauthorized access to sensitive files. An example of a local attack is the kernel attack. Such an attack violates the integrity of the targeted system's kernel by altering its regular behavior to the benefit of the attacker. For instance, specific kernel functions may be modified in order to record activities of the users of a given system, such as the pressing of keyboard keys. The real-time recording of pressed keyboard keys is a feature of a specific class of malicious software known as keyloggers. In Figure 2.1b, we depict a scenario where a malicious user deploys a keylogger in a host's kernel, an action that violates its integrity by modifying specific kernel functions in order to intercept and record pressed keys.

Besides intrusion detection, there are many other security mechanisms used to enforce the properties of confidentiality, integrity, and availability of system data and

services. Kruegel et al. [KVV05] classify security mechanisms by taking an attack-centric approach distinguishing between *attack prevention*, *attack avoidance*, and *attack detection* mechanisms. Based on this classification, we put intrusion detection into a common context with other security mechanisms, as depicted in Figure 2.2.

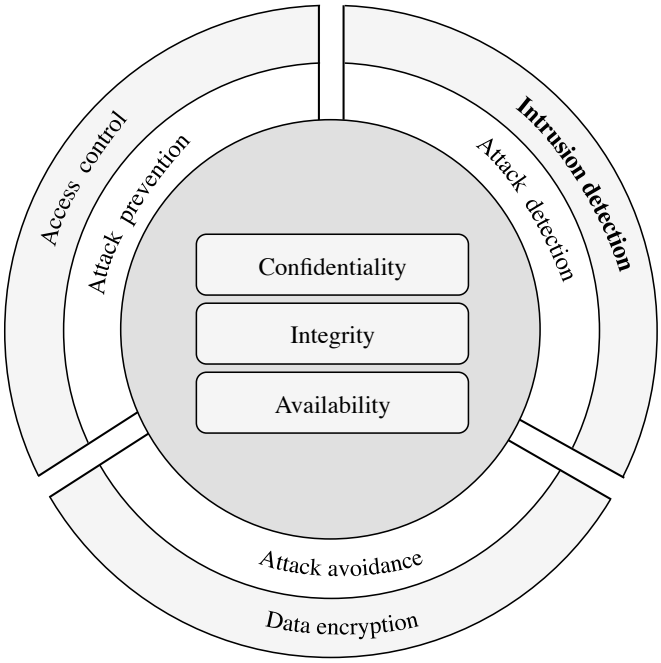


Figure 2.2: Intrusion detection in relation to other common security mechanisms.

The attack prevention class includes security mechanisms that prevent attackers from reaching, or gaining access to, the targeted system. A representative mechanism that belongs to this class is access control, which uses the concept of identity to distinguish between authorized and unauthorized parties. For instance, firewalls distinguish between different parties trying to reach a given system over a network connection based, for example, on their Internet Protocol (IP) addresses. According to access control policies, firewalls may allow or deny access to the system.

The attack avoidance class includes security mechanisms that modify the data stored in the targeted system such that it would be of no use to an attacker in case of an intrusion. A representative mechanism that belongs to this class is data encryption, which is typically implemented using encryption algorithms such as Rivest Shamir Adleman (RSA), Data Encryption Standard (DES), and so on.

The attack detection class includes security mechanisms that detect on-going attacks under the assumption that an attacker can reach, or gain access to, the targeted system and interact with it. A representative security mechanism that belongs to this class is **intrusion detection**.

2.1.2 Intrusion Detection Systems: A Systematization

According to Scarfone et al. [SM07] from NIST, intrusion detection is *"the process of monitoring the events occurring in a computer system or network and analyzing them for signs of possible incidents, which are violations or imminent threats of violation of computer security policies, acceptable use policies, or standard security practices."* Given the above definition, an IDS can be defined as the software that automates the intrusion detection process.

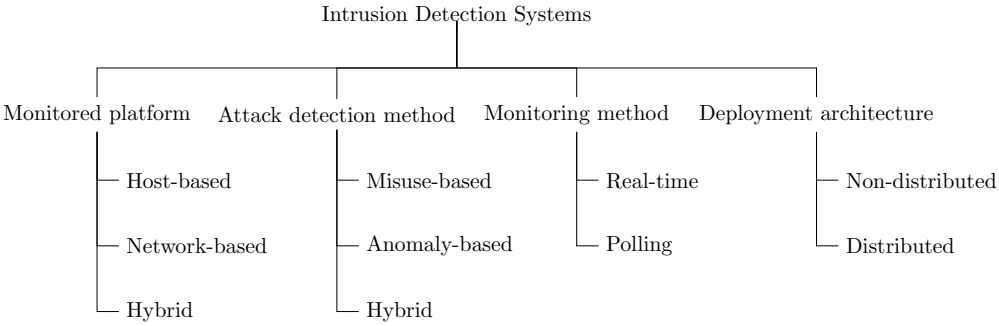


Figure 2.3: Different types of IDSs.

IDSs can be categorized according to many different properties. In the rest of this section, we present a categorization of IDSs with respect to the properties that we consider relevant for evaluating and comparing different systems. We refer the reader to [DDW99] for an in-depth categorization of IDSs. In Figure 2.3, we depict an IDS categorization scheme that we constructed by considering the following properties of IDSs:

(i) Monitored platform: According to the target platform that IDSs monitor, they can be categorized into host-based, network-based, or hybrid IDSs.

Host-based IDSs monitor the activities of the users of the host where they are deployed, which often includes the behavior of applications running on the host, in order to detect local attacks. Host-based IDSs are typically able to detect a variety of local attacks such as unauthorized modifications of sensitive files or abnormal execution behaviors of applications. Open Source Security (OSSEC) [oss] and Tripwire [tri] are among the most popular host-based IDSs at the time of writing.

Network-based IDSs monitor the network traffic that is destined for, and/or originates from, a single host or a set of hosts that constitute a network environment, in order to detect remote attacks. A network-based IDS is typically deployed at the perimeter of a network such that it can monitor all incoming and/or outgoing network traffic. Snort [Roe99], Bro [TBNSM], and Suricata [sur] are among the most popular network-based IDSs at the time of writing.

Hybrid IDSs are a combination of host-based and network-based IDSs. A typical hybrid IDS is deployed in the host that it monitors as a host-based IDS. An example of a hybrid IDS is VMFence [JXZ+09].

In Figure 2.4, we depict a typical IDS deployment scenario where a network-based IDS is deployed at the perimeter of a LAN (i.e., in the LAN's gateway used to connect it with a public network) in order to monitor all incoming and outgoing traffic. Further, a host-based IDS, monitoring filesystem changes and application behavior, is deployed in each of the hosts connected to the LAN.

(ii) Attack detection method: According to the employed attack detection method, IDSs can be categorized into misuse-based, anomaly-based, or hybrid IDSs.

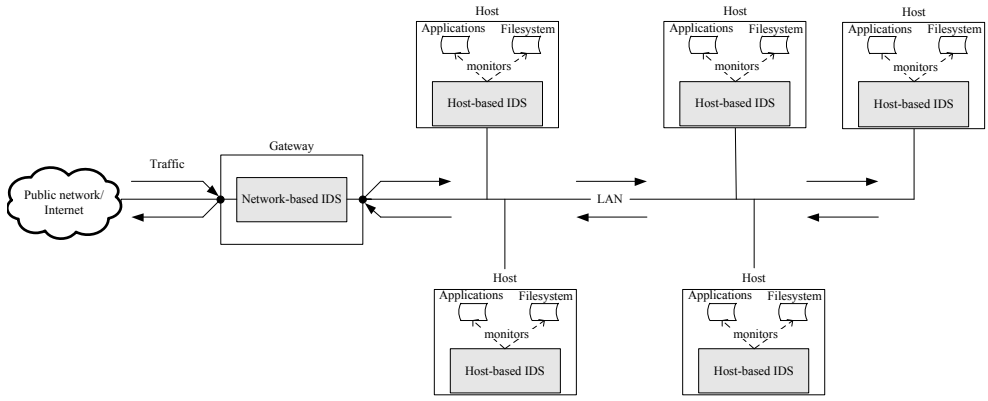


Figure 2.4: Deployment scenario of a network-based IDS and multiple host-based IDSs.

Misuse-based IDSs evaluate system and/or network activities against a set of signatures of known attacks. An attack signature is a unique arrangement of information used to identify attack attempts exploiting known vulnerabilities. For instance, the popular network-based IDS Snort [Roe99] is a misuse-based IDS that matches the content of network packets against a set of signatures and issues an alert if it finds a match. Given that misuse-based IDSs use signatures of known attacks for detecting attacks, they are not able to detect zero-day attacks. Under zero-day attack, we understand an attack that exploits a vulnerability that has not been publicly disclosed before the execution of the attack. The notion "zero-day" indicates that such an attack occurs on "day zero" of public awareness of the exploited vulnerability.

Anomaly-based IDSs use a profile of normal (i.e., regular) network and/or system activities as a reference to distinguish between regular activities and anomalous activities, the latter being treated as attacks. Anomaly-based IDSs must be initially trained by monitoring regular activities in order to construct regular activity profiles. In contrast to misuse-based IDSs, anomaly-based IDSs are able to detect zero-day attacks as well as known attacks, since both typically manifest themselves through anomalies. However, depending on the quality of their training and the sensitivity of the employed attack detection algorithms, anomaly-based IDSs may often mislabel regular activities as anomalous, which is their major disadvantage. Under sensitivity of an attack detection algorithm, we understand the smallest deviation of monitored system or network

activities from a regular activity profile, for which the algorithm labels the activities as anomalous.

Hybrid IDSs use both misuse-based and anomaly-based attack detection methods.

(iii) Monitoring method: According to the employed monitoring method, IDSs can be categorized into real-time monitoring or polling IDSs.

Real-time monitoring IDSs, also known as *event-driven IDSs*, analyze system and/or network activities as they occur. An example of a real-time monitoring IDS is the network-based IDS Snort [Roe99], which intercepts and analyzes network packets in order to detect attacks. Another example is the file integrity monitoring component of the host-based IDS OSSEC [oss], which performs real-time monitoring in order to inspect occuring filesystem activities (e.g., file write and read operations).

In contrast to real-time monitoring IDSs, *polling IDSs* do not intercept executed activities in order to obtain input data for analysis, but rather obtain such data periodically in an asynchronous manner. Some polling IDSs, known as log analysis IDSs, obtain input data in the form of log files where system and/or network activities that have already occurred are stored. Alternatively, polling IDSs periodically inspect relevant system components, for example, content of memory regions allocated to the kernel in order to detect kernel attacks. The vast majority of the IDSs that use polling as a monitoring method are host-based IDSs, such as Wizard [GR03] and OSSEC [oss].

(iv) Deployment architecture: According to their deployment architecture, IDSs can be categorized into non-distributed or distributed IDSs.

Non-distributed IDSs, also known as *centralized IDSs*, perform the same functions and are deployed in an identical manner to hybrid, host-, or network-based IDSs, which we discussed in detail earlier. In the following, we focus on distributed IDSs, which have distinct characteristics due to their specific deployment architecture.

Distributed IDSs consist of multiple intrusion detection sub-systems (also known as nodes or agents) that communicate and/or exchange intrusion detection-relevant data (e.g., attack alerts, records of monitored activities). The communication is between the agents themselves or with a centralized server that aggregates information obtained from the agents and/or performs tasks such as management of the agents, analysis of the data provided by them, and so on.

Because of its architecture, a distributed IDS maintains a global view of the network and/or host activities occuring at multiple sites which may even be sparsely geographically distributed. In addition to attacks targeting individual hosts, the latter also enables the detection of coordinated attacks. Under coordinated attack, we understand carefully orchestrated attack that targets multiple hosts at specific moments in time towards achieving a given malicious goal. However, the benefits of using a distributed IDS come at the cost of network overhead caused by the communication required for its operation. Example of a distributed IDS is the host-based IDS OSSEC [oss], which may be configured to operate in a distributed manner.

In Table 2.1, we summarize the categorization of IDSs presented above.

Table 2.1: Categorization of intrusion detection systems.

Property	IDS type	Description
Monitored platform	Host-based	An IDS that monitors the activities on the system (i.e., the host) where it is deployed in order to detect local attacks — attacks executed by users of the targeted system itself (e.g., OSSEC [oss]).
	Network-based	An IDS that monitors network traffic in order to detect remote attacks — attacks carried out over a network connection (e.g., Snort [Roe99]).
	Hybrid	An IDS that is a combination of host and network-based IDSs (see [JXZ+09]).
Attack detection method	Misuse-based	An IDS that evaluates system and/or network activities against a set of signatures of known attacks (e.g., Snort [Roe99]); therefore, it is not able to detect zero-day attacks — attacks that exploit vulnerabilities that have not been publicly disclosed before the execution of the attacks.
	Anomaly-based	An IDS that uses a baseline profile of regular network and/or system activities as a reference to distinguish between regular and anomalous activities, the latter being treated as attacks (see [AT]+10]); therefore, it is able to detect zero-day as well as known attacks, however, it may often mislabel regular activities as anomalous, which is its major disadvantage. An anomaly-based IDS must be trained by monitoring regular activities in order to construct baseline activity profiles.
	Hybrid	An IDS that uses both misuse-based and anomaly-based attack detection methods (see [MP13]).
Monitoring method	Real-time	An IDS that analyzes system and/or network activities as they occur (e.g., Snort [Roe99]).
	Polling	An IDS that does not analyze system and/or network activities as they occur, but obtains input data for analysis periodically in an asynchronous manner (e.g., OSSEC [oss]).
Deployment architecture	Non-distributed	A non-compound IDS that can be deployed only at a single location (e.g., Snort [Roe99]).
	Distributed	A compound IDS that consists of multiple intrusion detection subsystems that can be deployed at different locations and communicate to exchange intrusion detection-relevant data, for example, attack alerts (e.g., OSSEC [oss], which can be configured to operate in a distributed manner). Distributed IDSs can detect coordinated attacks targeting multiple sites in a given time order.

2.2 Evaluation of Intrusion Detection Systems

2.2.1 Application Scenarios

As we mentioned in Section 1.1, IDS evaluation helps in answering two main high-level questions: *How well an IDS performs?*, and *How well an IDS performs when compared to other IDSs?* The answers to these questions are of interest to many different types of professionals in the field of communications and information security. This includes

IDS designers, both researchers and industrial software architects, as well as IDS users, such as IT security officers. In this section, we demonstrate the broad relevance of IDS evaluation by discussing its relevance in the context of the mentioned professions.

Researchers advance the field of intrusion detection by designing novel intrusion detection methods and/or IDS architectures. They typically focus on designing IDSs that are superior in terms of given IDS properties that are subject of research, for example, attack detection accuracy or workload processing capacity. To demonstrate the value of the research outcome, researchers typically perform small-scale evaluation studies comparing the proposed IDS with other IDSs in terms of the considered IDS properties. For instance, Meng et al. [ML12] measure workload processing throughput, Mohammed et al. [MOL+11] measure power consumption, and Sinha et al. [SJP06] measure memory consumption. Further, in order to demonstrate that the proposed IDSs are practically useful, researchers also evaluate IDS properties that are not necessarily in the focus of their research, but are relevant from a practical perspective. For example, Lombardi et al. [LDP11] measure the performance overhead incurred by the IDS they propose.

Industrial software architects design IDSs with an extensive set of features according to their demand on the market. IDSs, in this context, are typically evaluated by carrying out tests of a large scale. The latter are part of regular quality assurance procedures before releasing a product for sale. They normally use internationally standardized tests for evaluating IDSs in a standard and comprehensive manner. For instance, Microsoft's Internet Security and Acceleration (ISA) Server 2006 [ISA], which features intrusion detection, has been evaluated according to the international standard Common Criteria framework for evaluating IT security products [mic]. Standardized IDS tests are performed in strictly controlled environments and normally by independent testing laboratories, such as NSS Labs [nssa], to ensure credibility of the results.

In contrast to IDS evaluation studies performed by researchers, evaluation studies in industry normally include the evaluation of IDS properties that are relevant from a marketing perspective. An example of such a property is the financial cost of deploying and maintaining an IDS, which is evaluated as part of the IDS tests performed by NSS Labs.

IT security officers use IDSs to protect environments they are in charge of from malicious activities. They may evaluate IDSs, for example, when designing security architectures, in order to select an IDS that is considered as optimal for protecting a given environment. Further, if a security architecture is already in place, an IT security officer may evaluate the performance of the selected IDS for different configurations in order to find an optimal configuration. The performance is typically very sensitive to the way the IDS is configured.

In addition to security and performance-related aspects, as part of IDS evaluation studies, further usability-related aspects may also be considered. This is to be expected since IT security officers deal with IDSs on a daily basis. For instance, security officers in charge of protecting large-scale environments may be cognitively overloaded by the output produced by the deployed IDS(s), an issue acknowledged by many researchers

(e.g., Komlodi et al. [KGL04]). Thus, the ability to produce structured output that can be analyzed in an efficient manner is an important property often considered when evaluating IDSs.

2.2.2 Historical Overview

In Figure 2.5, we depict chronologically ordered dates that mark major developments in the area of IDS evaluation from its inception until the present date.

The earliest effort on evaluating IDSs in a systematic manner is the work of Puketza et al. [PZC+96], [PCOM97]. They presented an approach for evaluating IDSs based on principles of the field of software systems testing. Puketza et al. were the first to develop a framework for evaluating IDSs, which they describe in detail in their work from 1997 [PCOM97]. They used the framework to evaluate a network-based IDS in terms of attack detection accuracy, resource consumption, and performance under stress.

The years of 1998, 1999, and 2000 mark a major accomplishment in the area of IDS evaluation. The Lincoln Laboratory at Massachusetts Institute of Technology (MIT), sponsored by Defense Advanced Research Projects Agency (DARPA), evaluated multiple IDSs using generated trace files that contain host and network activities of benign and malicious nature. The latter are commonly known as the DARPA datasets. Cunningham et al. [CLF+99] describe in detail the approach taken to generate the DARPA datasets. The DARPA datasets are still extensively used in IDS evaluation studies.

In 1998, Debar et al. [DDWL98] from the International Business Machines Corporation (IBM) Zurich Research Laboratory developed a workbench for evaluating IDSs. The workbench enabled the execution of attack scripts stored in a database maintained internally at IBM and the generation of regular workloads for training anomaly-based IDSs. Debar et al. demonstrated the use of the workbench by evaluating multiple host-based IDSs.

A recent effort to support the rigorous evaluation of IDSs is being driven by Symantec. Dumitras et al. [DS11] presented the Symantec's Worldwide Intelligence Network Environment (WINE) datasets [Wor], which contain local and remote attacks (see Table 2.1). They also presented an evaluation platform that makes use of the datasets and is available for use by researchers for evaluating security mechanisms. However, since the datasets are captured from real network infrastructures and systems, and therefore contain private user data, they can only be accessed on-site at Symantec in order to avoid legal issues. The large scale of this project is indicated by the fact that Symantec continuously monitors and records malicious activities using more than 240,000 sensors deployed in 200 countries.

In addition to attacks, which can be used for evaluating IDSs, the WINE datasets contain samples of malware (i.e., malicious software, such as trojans or viruses), which can be used for evaluating malware detection systems (e.g., anti-virus systems). In contrast to IDSs, which are designed to detect on-going attacks, malware detection systems are

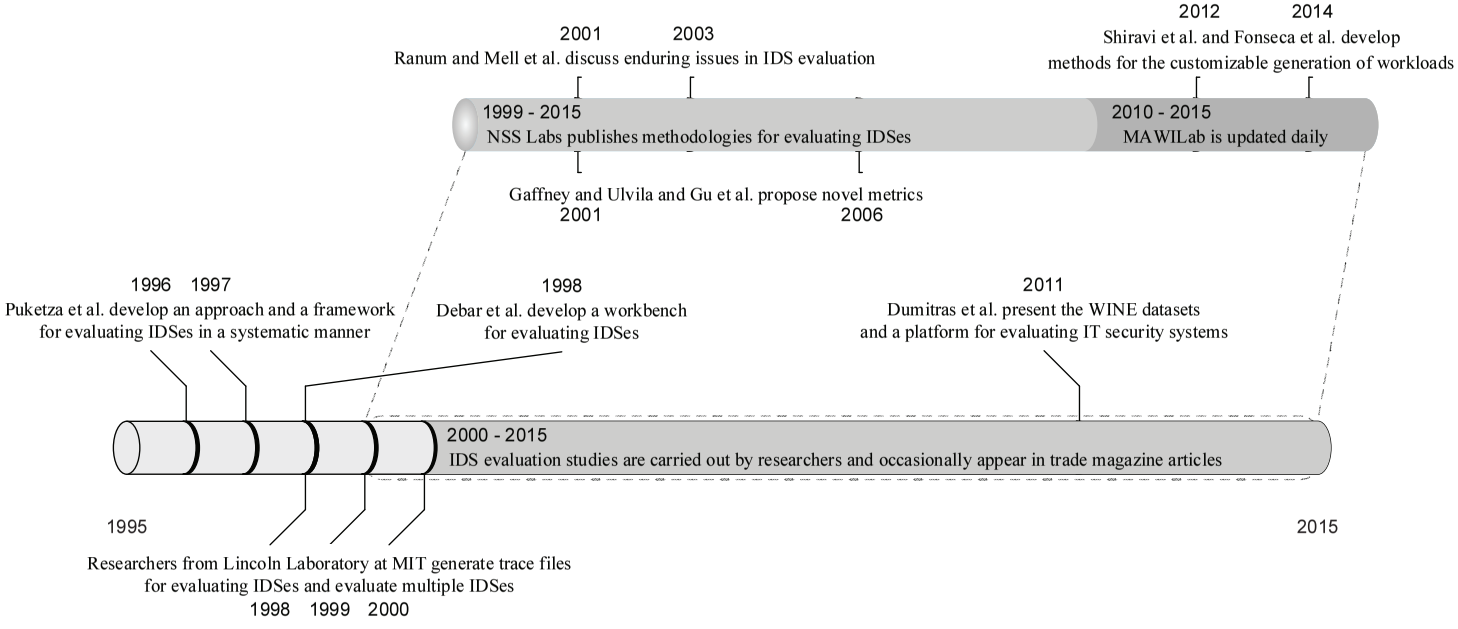


Figure 2.5: Timelime showing dates that mark major developments in the area of IDS evaluation.

designed to detect malware running on a given host, whose installation normally takes place after an intrusion (i.e., a successful attack) has occurred. Evaluation of malware detection systems is outside of the scope of this work.

There have been many small-scale IDS evaluation efforts between 2000 and today. Articles reviewing and comparing IDSs occasionally appear in trade magazines, such as the IDS evaluation studies presented in the SC magazine in 2011 [SC]. Following the rising interest of researchers in intrusion detection since 2000, many IDS evaluation studies have been presented as part of publications proposing novel intrusion detection techniques or IDS evaluation methods.

Several works published between 2000 and today have had long-term impact on the IDS evaluation area: Ranum [Ran01] and Mell et al. [MHL+03] proposed approaches and gave recommendations towards addressing enduring issues in IDS evaluation (e.g., use of faulty or unrepresentative workloads, inaccurate interpretation of results from IDS evaluation studies); Gaffney et al. [GU01] and Gu et al. [GFD+06] were the first to propose metrics for quantifying IDS attack detection accuracy that use specific measurement methods in order to address issues in using the conventional metrics at the time, such as the Receiver Operating Characteristic (ROC) curve; focusing on the issue of using unrepresentative workloads, Shiravi et al. [SSTG12] and Fonesca et al. [FVM14] developed methods for the customizable generation of IDS evaluation workloads that closely resemble real-world workloads at the time they are generated.

In 2010, the Measurement and Analysis on the WIDE Internet (MAWI) Working Group of the Widely Integrated Distributed Environment (WIDE) project announced MAWILab, a repository of publicly available traces intended for use in IDS evaluation studies [FBAF10], [MAW]. This is a significant effort to enable the representative evaluation of modern network-based IDSs. The trace files in MAWILab contain network traffic captured from a trans-Pacific 150 Mbps link between Japan and the United States. They contain regular network traffic as well as attacks, which are labeled before the public release of the traces using a variety of attack labeling methods. MAWILab has been updated daily since its release until the present date.

In 1999, NSS Labs, an information security research and testing organization, pioneered third party testing of IDSs with the publication of the first systematic, criteria-driven methodology for IDS testing. From 1999 until the present date, NSS Labs has been continuously supplying methodologies for testing IDSs to the public following trends in IDS design. These methodologies may serve as guidelines for the rigorous testing of IDSs. For instance, in 2015, NSS Labs published a methodology for testing next-generation IDSs [NGI], that is, IDSs designed to detect novel threats, such as advanced persistent threats (APTs) and social media threats.

2.3 Summary

In this chapter, we provided the background knowledge essential for understanding the topic of intrusion detection and IDS evaluation. We discussed different types of attacks and put intrusion detection into a common context with other security

mechanisms. We also defined different types of IDSs. We systematized the latter according to the monitored platform, the attack detection method, the monitoring method, and the deployment architecture. In addition, we demonstrated the wide applicability of IDS evaluation by discussing its relevance to researchers, industrial software architects, and IT security officers. Finally, we provided a historical overview of major developments in the area of IDS evaluation ordering them chronologically.

Chapter 3

IDS Evaluation Design Space: A Survey of Common Practices

In this chapter, we present an IDS evaluation design space structured into three parts — workloads (Section 3.2), metrics (Section 3.3), and measurement methodology (Section 3.4), which are in the focus of this thesis (see Section 1.1). The systematization presented in this chapter includes 124 references out of which 65 are peer-reviewed research publications and technical reports, and 59 are links to specific tool information sites, relevant data, and similar.

Although they have a lot in common when it comes to evaluation, different types of IDSs also pose challenges and requirements that apply specifically for each IDS type. When a given category or part of the design space relates closely to a particular IDS type, we stress such a relation in our discussions. We do so especially when considering evaluation methodologies in Section 3.4, where we round up and finalize the IDS evaluation design space.

The work presented in this chapter has been published in [MVK+15].

3.1 Related Work

There are only a few previous efforts that provide an overview of the existing work on IDS evaluation. However, they do not cover developments until the current date and/or are focusing on specific aspects, as opposed to providing a broad overview of the field as presented in this chapter.

Athanasiades et al. [AAL+03] focus on IDS evaluation methodologies analyzing several existing IDS evaluation tools and environments at the time of writing (i.e., 2003). In particular, they analyze the Defense Advanced Research Projects Agency (DARPA) environment [IDE] and the Lincoln Adaptable Real-time Information Assurance Testbed (LARIAT) environment [RCF+01]. Further, they evaluate the usability of multiple tools used in IDS evaluation experiments such as the test suite Nidsbench [nid]. Finally, they describe several conducted IDS evaluation studies, including IDS evaluation studies performed by trade magazines.

Zanero [Zan06] identifies IDS evaluation requirements with respect to different types of IDSs and intrusion detection techniques. He also provides a brief overview

of the employed IDS evaluation workloads and environments at the time of writing (i.e., 2006). This includes the Neohapsis Open Security Evaluation Criteria (OSEC) environment [neo], the DARPA datasets [LHF+00], and similar. Finally, he concludes that the evaluation of IDSs is an open research area riddled with many challenges.

Some researchers are focusing on specific aspects in the context of their own work analyzing, for example, the existing work on IDS evaluation related to one particular evaluation component, that is, either workloads, metrics or methodology. For instance, Debar et al. [DM02] provide an overview of several IDS evaluation methodologies considering both approaches followed by research institutes (e.g., the IDS evaluation platform at University of California (UC) Davis [PCOM97]) and by commercial organizations [MS01]. Further, Gu et al. [GFD+06] provide a brief overview of various IDS evaluation metrics.

In summary, to the best of our knowledge, no global and broad overview of the IDS evaluation field has been published so far, systematizing existing knowledge, best practices, and experiences in a comprehensive and up-to-date manner. To the best of our knowledge, we are the first to systematically consider workloads, metrics and measurement methodologies as integral parts of every IDS evaluation approach. In this chapter, we track the development of relevant practical and research work in the field of IDS evaluation until the current date and we identify milestone accomplishments. We mention commonly used tools and we provide references to technical manuals and related information sources. We also discuss topics that have been, and still are, actively debated in the IDS evaluation community.

3.2 Workloads

In Figure 3.1, we depict the workload part of the IDS evaluation design space. In order to evaluate an IDS, we need both malicious and benign workloads. These can be used separately (e.g., as *pure malicious* or *pure benign* workloads) to measure the capacity of an IDS as in [BSNS11a] and [JXZ+11], or its attack coverage as in [RRL+12]. Pure benign workloads are workloads that do not contain attacks, whereas pure malicious workloads are workloads that contain only attacks. Alternatively, one can use *mixed* workloads (i.e., workloads that are a mixture of pure benign and pure malicious workloads) to subject an IDS under test to realistic attack scenarios as in [YD11], [ATJ+10], and [SJP06].

IDS evaluation workloads normally take an *executable* form for live testing of an IDS, or a *trace* form, generated by recording a live execution of workloads for later replay. The trace replay is performed with tools designed to process trace files — a common combination is the use of the tool tcpdump [PR] for capturing network traces for subsequent replay by tcpreplay [Tcpb]. A major advantage of using workloads in executable form is that they closely resemble a real workload as monitored by an IDS during operation. However, a malicious workload in executable form requires a specific victim environment which can be expensive and time-consuming to setup (see [DDWL98]). In contrast, such an environment is not always required for replaying

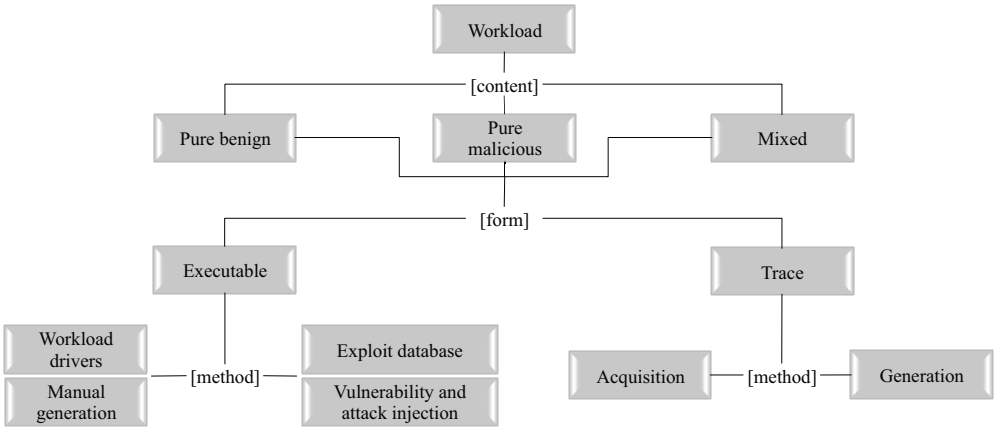


Figure 3.1: IDS evaluation design space: Workloads [There are three types of workloads with respect to workload content: *pure benign* (workloads that do not contain attacks), *pure malicious* (workloads that contain only attacks), and *mixed* • There are two types of pure benign, pure malicious, or mixed workloads with respect to their form: *executable* and *trace* • There are two methods for generating pure benign executable workloads: use of *workload drivers* (Section 3.2.1) and *manual generation* (Section 3.2.2) • There are two methods for generating pure malicious executable workloads: use of an *exploit database* (Section 3.2.3), and *vulnerability and attack injection* (Section 3.2.4) • There are two methods for generating pure benign, pure malicious, or mixed workloads in trace form: *acquisition* (Section 3.2.5) and *generation* (Section 3.2.6)].

workload traces. Further, multiple evaluation runs are typically required to ensure statistical significance of the observed system behavior. However, replicating evaluation experiments when using executable malicious workloads is usually challenging since the execution of attacks might crash the victim environment or render it in an unstable state. Moreover, the process of restoring the environment to an identical state as before the execution of the attacks may be very time-consuming.

In the following, we first discuss different methods for the generation of benign and malicious workloads in executable form (see Figure 3.1). We discuss the use of *workload drivers* and *manual generation* approaches for generating pure benign workloads. We also discuss the use of an *exploit database* and *vulnerability and attack injection* techniques for generating pure malicious workloads. We note that mixed workloads in executable form can be generated by using in combination the previously mentioned methods for generating pure benign and pure malicious workloads. Finally, we discuss methods for obtaining pure benign, pure malicious or mixed workloads in a trace form, distinguishing between trace *acquisition* and trace *generation*.

3.2.1 Pure Benign → Executable Form → Workload Drivers

For the purpose of live IDS testing, a common practice is to use benign workload drivers to generate artificial pure benign workloads with different characteristics.

Some of the commonly used workload drivers are: SPEC CPU2000 [SPEb] for CPU-intensive workloads; iozone [IFB] and Postmark [Kat97] for file input/output (I/O)-intensive workloads; httpbench [Htt], dkftpbench [Dkf], and ApacheBench [aAHSBT] for network-intensive workloads; and UnixBench [Uni] for system-wide workloads that exercise not only the hardware, but also the operating system. Experiments using the mentioned tools were performed in [GPB+03], [PKSZ04], [CM06], [RJX08], [ZWGW08], [JXZ+09], [LDP11], [JXZ+11], and [RRL+12]. As expected, the CPU- and file I/O-intensive drivers have been employed mainly for evaluating host-based IDSs, while the network-intensive drivers for evaluating network-based IDSs. We look at the IDS properties typically quantified using these drivers when we discuss IDS evaluation methodologies in Section 3.4.

A major advantage of using benign workload drivers is the ability to customize the workload in terms of its temporal and intensity characteristics. For instance, one may configure a workload driver to gradually increase the workload intensity over time, as typically done when evaluating the workload processing capacity of an IDS. A disadvantage is that the workloads generated by such drivers often do not closely resemble real-life workloads. In the case when realistic benign workloads are needed (e.g., to be used as background activities mixed with attacks), a reasonable alternative is the manual generation of benign workloads.

3.2.2 Pure Benign → Executable Form → Manual Generation

Under manual generation of workloads, we understand the execution of real system users' tasks known to exercise specific system resources, which is typically applied in the context of evaluating host-based IDSs. For example, a common approach is to use: file encoding or tracing tasks to emulate CPU-intensive tasks (e.g., Dunlap et al. [DKC+02] perform ray-tracing, Srivastava et al. [SSG08] perform video transcoding, Lombardi et al. [LDP11] perform mp3 file encoding); file conversion and copying of large files to emulate file I/O-intensive tasks (e.g., Lombardi et al. [LDP11] and Allalouf et al. [ABYSS10] use the UNIX command *dd* to perform file copy operations); kernel compilation to emulate mixed (i.e., both CPU- and file I/O-intensive) tasks (e.g., performed by Wright et al. [WCS+02], Dunlap et al. [DKC+02], Riley et al. [RJX08], Lombard et al. [LDP11], and Reeves et al. [RRL+12]).

Provided that it is based on a realistic activity model, this approach of benign workload generation enables the generation of workloads with a behavior similar to the one observed by an IDS during regular system operation. Thus, it is suitable when realistic benign workloads are required (e.g., for training and evaluation of anomaly-based IDSs). Also, it is suitable for generation of workload traces capturing realistic workloads executed in a recording testbed, a topic that we discuss later in Section 3.2.6. However, the manual benign workload generation does not support workload customization as workload drivers do, and might require a substantial amount of manpower.

In Table 3.1, we provide an overview of the use of the discussed methods for generating pure benign workloads in practice. We also provide stepwise guidelines (see

Table 3.1, section 'Selection guidelines') for selecting an approach from those presented in Table 3.1 to apply in a given IDS evaluation study, that is, to evaluate a given IDS property (e.g., workload processing capacity or performance overhead, see Section 3.4).

3.2.3 Pure Malicious → Executable Form → Exploit Database

Pure malicious workloads in executable form are used for evaluating the attack detection coverage of IDSs (see Section 3.4). As pure malicious workloads in executable form, security researchers typically use an exploit (i.e., an attack script) database. They can assemble an exploit database by themselves, or use a readily available one.

Exploit Database → Manual Assembly

A major disadvantage of the manual assembly is the high cost of the attack script collection process. Locating the attack scripts needed for exploiting specific vulnerabilities and obtaining the required vulnerable software is typically time-consuming. In addition, once the needed attack scripts are found, they typically have to be adapted to exploit the vulnerabilities of the victim environments, especially when the attack scripts exploit local system vulnerabilities for evaluating host-based IDSs. This includes, for example, time-consuming adaptation of employed exploitation techniques.

Depending on the size of a manually assembled exploit database, the previously mentioned activities might require a considerable amount of manpower in order to be completed in a reasonable time frame. For instance, Mell et al. [MHL+03] report that based on previous experiences, a single attack script requires approximately one person-week to modify the script's code, to test it, and to integrate it in an IDS evaluation environment. Mell et al. [MHL+03] also report that in 2001 the average number of attack scripts used for evaluating IDSs was in the range of 9 to 66. We observe that some recent works, such as [LDP11], use as low as 4 attack scripts.

To assemble an exploit database, IDS evaluators normally obtain attack scripts from public exploit repositories. In Table 3.2, we list popular exploit repositories characterized according to the criteria 'exploit verification', 'vulnerable software', and 'vulnerability identifiers' (see Table 3.2, section 'Categorization criteria'). Given that an exploit repository hosts a limited number of attack scripts, an IDS evaluator normally does not search only a single repository, but as many as it takes until the desired number of attack scripts is obtained. In this process, we recommend that an IDS evaluator prioritizes the exploit repository 'Exploit database' (marked in bold in Table 3.2) since it fulfills more criteria than any other repository presented in Table 3.2 (see Table 3.2, section 'Categorization criteria', for an overview of the benefits of a repository fulfilling the criteria 'exploit verification', 'vulnerable software', and 'vulnerability identifiers').

Publicly available attack scripts normally do not feature techniques for evaluating the ability of an IDS to detect evasive attacks. Adapting publicly available attack scripts to feature techniques for evaluating the ability of an IDS to detect evasive attacks normally requires an in-depth knowledge of the architecture and inner working mechanisms

Table 3.1: Practices for generating pure benign workloads in executable form [W: workload drivers, M: manual generation].

Reference	Method / Workload type / [Approach / Workload driver]
[ABYSS10]	M / I/O-intensive / Creating, deleting and truncating files, appending data to files; M / Mixed / Compilation of libraries
[CPX+13]	M / CPU-intensive / Compiling Java code; **M / Network-intensive / Web surfing**, Telnet sessions; M / I/O-intensive / Reading PDF files; **W / CPU-intensive / SPEC CPU2000**
[DKC+02]	M / CPU-intensive / Ray tracing; **M / Mixed / Kernel compilation**; W / Network-intensive / SPECweb99 [SPEc]
[GPB+03]	M / CPU-intensive / Building SSH server; **W / I/O-intensive / Postmark**
[JXZ+11]	**W / I/O-intensive / iozone**; **W / Network-intensive / Apachebench, dkftp-bench**
[FJGS+00]	M / Network-intensive / Executing `traceroute`
[LMJ07]	M / CPU-intensive / Executing Linux commands (`ps`, `who`); M / Mixed / Executing Linux commands (`find`, `ls`); M / Network-intensive / Downloading files
[ML12]	M / Network-intensive / Web surfing, transmitting files
[PKSZ04]	M / I/O-intensive / File read operations; W / CPU-intensive / Am-utils [TBA-SoU]; W / I/O-intensive / Postmark
[RRL+12]	M / Mixed / Server and kernel compilation; W / CPU-intensive / SPEC CPU2000; W / Mixed / lmbench [LTfPA]
[RJX08]	M / Mixed / Kernel compilation, executing `insmod`; **W / Mixed / Unixbench**; W / Network-intensive / Apachebench
[SSG08]	**M / CPU-intensive / Encoding files**; M / I/O-intensive / Copying files; M / Mixed / Video file compression and decompression, kernel compilation
[WCS+02]	M / Mixed / Kernel compilation; W / Network-intensive / Webstone [MWBI]; W / Mixed / lmbench
[ZWGW08]	W / I/O-intensive / iozone

Selection guidelines

1) Select a method for generating workloads, that is, use of workload drivers ('workload drivers' in Table 3.1) or manual generation, by taking the advantages and disadvantages of the different methods into account (see Section 3.2.1 and Section 3.2.2). In Section 3.4, in the context of IDS evaluation methodologies, we present IDS evaluation scenarios where the different methods are applied for evaluating various IDS properties.

2) Select the type of workloads (e.g., CPU- or I/O-intensive) that is required for evaluating the considered IDS property. For instance, CPU- and/or I/O-intensive workloads are required for evaluating the performance overhead of a host-based IDS, and network-intensive workloads are required for evaluating any property of a network-based IDS. In Section 3.4, we present IDS evaluation scenarios where workloads of the different types are used for evaluating IDS properties.

3) Depending on the selection made in step *1)*, select an approach for manually generating workloads or a workload driver. This is normally done based on subjective criteria (e.g., prior experience with using a given workload driver). In an effort to provide general recommendations for selecting an approach for manually generating workloads / a workload driver, we mark the most popular approaches / workload drivers in bold. Based on what is reported in the surveyed work, we argue that the popularity of the workload drivers marked in bold is due to high configurability, representativeness of the workloads they generate, and ease of use.

Table 3.2: Popular exploit repositories.

Exploit database	Exploit verification	Vulnerable software	Vulnerability identifiers		
			CVE	OSVDB	BugTraq
1337day (`http://0day.today/`)	x				
Exploit database (`http://www.exploit-db.com/`)	x	x	x	x	
Packetstorm (`http://packetstormsecurity.com/`)			x	x	
SecuriTeam (`http://www.securiteam.com/exploits/`)			x	x	
Securityfocus (`http://www.securityfocus.com/`)		○	x		x

Categorization criteria

Criteria	Description
Exploit verification	An exploit repository that fulfills this criterion maintains a record for each hosted attack script indicating whether the script has been empirically verified to successfully exploit a specific vulnerability. This helps IDS evaluators to identify attack scripts that they can easily adapt to their specific requirements.
Vulnerable software	An exploit repository that fulfills this criterion provides a download link to the specific vulnerable software that can be exploited by different attack scripts. This helps IDS evaluators to quickly obtain this software and use it to experiment with the scripts. ○ marks partial fulfillment of this criteiron — an exploit repository that partially fulfills this criterion provides a link to the website of the vendor of the vulnerable software instead of a download link to the software, due to which it takes more time for an IDS evaluator to obtain the vulnerable software.
Vulnerability identifiers	An exploit repository that fulfills this criterion can be searched based on standard vulnerability identifiers. This enables IDS evaluators to quickly locate an attack script that exploits a given vulnerability. Common Vulnerabilities and Exposures (CVE) [CVEC], Open Sourced Vulnerability Database (OSVDB) [OSVDO], and BugTraq [Bug] are the de-facto standard vulnerability enumeration systems.

of the IDS, a topic that we discuss in detail in Section 3.4.1. Such a knowledge may be challenging to obtain if the evaluated IDS is closed source. Thus, IDS evaluators use third-party tools when executing attack scripts, such as Nikto [Nik]. Cheng et al. [CLLL12] provide an overview of IDS evasion techniques and discuss the use of the previously mentioned and similar tools for evaluating IDSs.

Exploit Database → Readily Available Exploit Database

To alleviate the above mentioned issues, many researchers employ penetration testing tools as a readily available exploit database. We discuss in detail the Metasploit

framework [PtsM], since it is the most popular penetration testing tool used extensively in both past and recent IDS evaluation experiments (e.g., by Nasr et al. [NKF12]). Some other penetration testing tools are Nikto, w3af [w3a], and Nessus [Nes]. The interest in Metasploit is not surprising, given that Metasploit enables a customizable and automated platform exploitation by using an exploit database that is maintained up-to-date and is freely available. However, although convenient, penetration testing frameworks have some critical limitations. Gadelrab [GER08] analyzes the Metasploit's database showing that most of the exploits are executed from remote sources, and therefore, they are most useful when evaluating network-based IDSs and are of limited use for evaluating host-based IDSs.

In order to provide an up-to-date characterization of Metasploit's exploit database, we analyzed the exploit database of Metasploit version 4.7, the most recent release at the time of writing. In Table 3.3, we categorize Metasploit's exploits according to the criteria 'execution source', 'target platforms', and 'exploit rank' [MFER] (see Table 3.3, section 'Categorization criteria'). Similarly to Gadelrab [GER08], we observe that Metasploit's exploit database contains mostly remote exploits, which makes it most useful for evaluating network-based IDSs. We also observe that a big portion of the exploits in the Metasploit's database have a 'great' and 'excellent' rank. This indicates that an IDS evaluator can use many attack scripts from Metasploit's database without crashing the victim platform(s). Finally, as we can see in Table 3.3, most of the remote and local exploits exploit vulnerabilities of Windows platforms.

3.2.4 Pure Malicious → Executable Form → Vulnerability and Attack Injection

An alternative approach to the use of an exploit database is the use of the vulnerability and attack injection technique. Vulnerability and attack injection enables live IDS testing by first artificially injecting exploitable vulnerable code in a target platform and then attacking the platform. Although not yet mature, this technique is useful in cases where collection of attack scripts that exploit vulnerabilities is unfeasible. As the injected vulnerable code may be exploitable remotely or locally, vulnerability and attack injection is useful for evaluating both host-based and network-based IDSs. However, injecting attacks such that the sensors of an IDS under test are exercised may require in-depth knowledge of the architecure and inner working mechanisms of the IDS.

Vulnerability and attack injection relies on the principles of the more general research area of fault injection. Fault injection is an approach for validating specific fault handling mechanisms and assessing the impact of faults in actual systems. In recent years, the interest in software fault injection has increased providing a basis for many research works on emulation of software faults. A specific application of software fault injection is injection of software faults that represent security vulnerabilities. Fonesca et al. [FVM14] proposed an approach that enables the automated vulnerability and attack injection of web applications, which is suitable for evaluating network-based IDSs.

Table 3.3: Characterization of Metasploit's exploit database.

	Execution source																			
	Remote															**Local**				
Target platforms / Exploit rank	AIX	IOS	BSDI	Unix	FreeBSD	HP-UX	Irix	Linux	Netware	OS X	Solaris	Windows	Multi-platform	**Total (per rank)**		Linux	OS X	Windows	Multi-platform	**Total (per rank)**
Manual				4				3		1		5	2	**15**						**0**
Low												10		**10**						**0**
Average				1	1			4	1	8	1	120	3	**139**				2		**2**
Normal				2				7		2		251	6	**268**		2				**2**
Good		2						14	1			159	9	**185**						**0**
Great	2		1	1	3			8		1	3	147	17	**183**		2		2		**4**
Excellent		1		70		1	1	30		2	5	89	84	**283**			2	8	1	**11**
Total (per platform)	**2**	**3**	**1**	**78**	**4**	**1**	**1**	**66**	**2**	**14**	**9**	**781**	**121**			**4**	**2**	**12**	**1**	
Total	**1083**															**19**				

Categorization criteria

Criteria	**Description**
Execution source	An exploit can be executed from a remote (i.e., a remote exploit), or a local source (i.e., a local exploit). Remote exploits are used for evaluating network-based IDSs, whereas local exploits are used for evaluating host-based IDSs. The amount of remote and local exploits in an exploit database indicates its suitability for evaluating network- and host-based IDSs.
Target platforms	Each exploit is designed to exploit a vulnerability of a single or multiple target platforms (i.e., multi-platform exploits), such as Linux, Solaris, or Windows. An exploit database covering a wide range of platforms is beneficial since, for example, it can be used to evaluate a variety of IDSs for different target platforms.
Exploit rank	Each exploit in Metasploit's database is ranked according to its impact on the target platform as shown below. The use of attack scripts that do not crash the target platform (e.g., scripts ranked as 'excellent') significantly reduces the time spent on restoring it (see Section 3.2). *Manual:* An exploit of this rank almost never successfully exploits a vulnerability and nearly always crashes the target platform. *Low:* An exploit of this rank almost never successfully exploits a vulnerability or successfully exploits a vulnerability in under 50% of the cases if the target platform is popular. *Average:* An exploit of this rank is unreliable and rarely successfully exploits a vulnerability. *Normal:* An exploit of this rank successfully exploits a vulnerability, but only a vulnerability of a specific version of the target platform and cannot reliably auto-detect a vulnerable platform. *Good:* An exploit of this rank has a default target platform (i.e., a platform that the exploit almost always successfully exploits), which is widely used. *Great:* An exploit of this rank has a default target platform and detects a vulnerable platform. *Excellent:* An exploit of this rank almost always successfully exploits a vulnerability and never crashes the target platform.

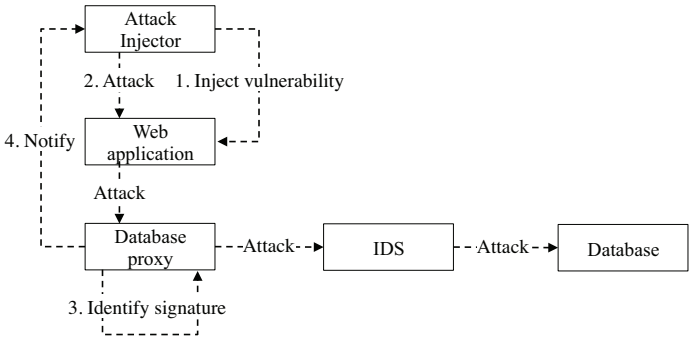


Figure 3.2: Use of vulnerability and attack injection to evaluate a network-based IDS.

We discuss a scenario where the approach of Fonesca et al. [FVM14] is applied for evaluating a network-based IDS that monitors network traffic to a database, which communicates with a web application, in order to detect Standard Query Language (SQL) injection attacks. Fonesca et al. [FVM14] built a Vulnerability Injector, a mechanism that injects vulnerabilities in the source code of web applications, and an Attack Injector, a mechanism that exploits the injected vulnerabilities. In order to inject vulnerabilities, the Vulnerability Injector first analyzes the application source code searching for locations where realistic vulnerabilities can be injected by code mutation. The Attack Injector then interacts with the web application in order to deliver attack payloads.

In Figure 3.2, we depict the approach of Fonesca et al. [FVM14] First, the Vulnerability Injector injects a vulnerability in the web application ("1. Inject vulnerability" in Figure 3.2), followed by the Attack Injector which delivers an attack payload with a given signature, that is, an attack identifier ("2. Attack" in Figure 3.2). The attack payload is targeted at the database ("Attack" in Figure 3.2). Fonesca et al. [FVM14] developed a database proxy that monitors the communication between the application and the database in order to identify the presence of attack signatures. When the proxy identifies the signature of the delivered attack payload ("3. Identify signature" in Figure 3.2), it notifies the Attack Injector that the attack payload has reached the database ("4. Notify" in Figure 3.2). In this way, the Attack Injector builds a "ground truth" knowledge. "Ground truth" is information about the attacks used as malicious workloads in a given IDS evaluation study (e.g., time of execution of the attacks). The output of an IDS under test is compared with "ground truth" information in order to quantify the attack detection accuracy of the IDS, a topic that we discuss in detail in Section 3.3.

3.2.5 Pure Malicious/Pure Benign/Mixed → Trace Form → Trace Acquisition

Under trace acquisition, we understand the process of obtaining real-world production traces from an organization (i.e., non-public, proprietary traces), or obtaining publicly

available traces that are intended for use in security research.

Trace Acquisition → Real-world Production Traces

Real-world production traces subject an IDS under test to a workload as observed during operation in a real production environment. However, they are usually very difficult to obtain mainly due to the unwilligness of industrial organizations to share operational traces because of privacy concerns. Thus, real-world traces are usually anonymized by using tools for that purpose. Such is the tool tcpmkpub [tcpa], which anonymizes network traces by modifying recorded network packets at multiple layers of the Transport Control Protocol/Internet Protocol (TCP/IP) network stack.

Some organizations are reluctant even towards trace anonymization due to the possibility of information leakages. For instance, trace files may be deanonymized to reveal sensitive internal information (e.g., Internet Protocol (IP) addresses, port numbers, network topologies). Coull et al. [CWM+07] demonstrate the severity of trace deanonymization by revealing anonymized information with 66% - 100% accuracy based on the traces provided by the Lawrence Berkeley National Laboratory (LBNL/ISCI) [lbn].

When it comes to the use of anonymization techniques on traces for IDS evaluation, Seeberg et al. [SP07] identify the following challenging requirements: *(i)* during anonymization, the smallest possible amount of intrusion detection relevant data should be removed, and *(ii)* assurance needs to be attained that no private and other sensitive information remains in the trace files after anonymization. For an IDS evaluator, the first requirement is of greatest concern since an anonymizer might remove data that is relevant for a given IDS under test. Therefore, the provisioning of extensive and accurate metadata on how a given trace file has been anonymized is crucial.

Another challenge is that attacks in real-world production traces are usually not labeled and traces may contain unknown attacks making the construction of the "ground truth" time-consuming since attacks have to be labeled manually. Lack of "ground truth" information severely limits the usability of trace files in IDS evaluation. For instance, it might be impossible to quantify the false negative detection rate of an IDS under test (see Section 3.3).

Trace Acquisition → Publicly Available Traces

In contrast to real-world traces, one can obtain publicly available traces without any legal constraints. However, the use of such traces has certain risks. For instance, publicly available traces often contain errors and quickly become outdated after their release since the recorded attacks have limited shelf-life. Consequently, claims on the generalizability of results from IDS evaluation studies based on publicly available traces can often be questioned. An in-depth knowledge about the characteristics of recorded activities in publicly available traces (e.g., types and distributions of recorded attacks) is a requirement for the accurate interpretation of results from IDS evaluation studies based on such traces.

The DARPA [IDE] and the derived Knowledge Discovery and Data Mining Cup 1999 (KDD'99) [UoC] datasets are the result of one of the most notable efforts up-to-date to provide publicly available data for security research. In three consecutive years (i.e., 1998, 1999, and 2000), three separate editions of the DARPA datasets have been made available. Since they contain local and remote attacks, the DARPA datasets are suitable for evaluating host- and network-based IDSs. They also contain training data, which makes them useful for evaluating anomaly-based IDSs. However, the DARPA and the KDD'99 datasets are currently considered outdated and have been often criticized (see [SP10] and [McH00]). Despite the criticism, these traces are still used in many recent IDS evaluation experiments (e.g., by Yu et al. [YD11] and Raja et al. [RAR12]).

In Table 3.4, we provide an overview of popular repositories of publicly available traces categorized according to multiple criteria (see Table 3.4, section 'Categorization criteria'): the Cooperative Association for Internet Data Analysis (CAIDA) [cai], the Defense Readiness Condition (DEFCON) [CtCtF], the DARPA/KDD'99 ([IDE], [UoC]), the Internet Traffic Archive (ITA) [ita], the Lawrence Berkeley National Laboratory (LBNL/ISCI) [lbn], and the Measurement and Analysis on the WIDE Internet (MAWILab) trace repositories [FBAF10]. We also provide stepwise guidelines (see Table 3.4, section 'Selection guidelines') for selecting a trace repository from those presented in Table 3.4 to use in a given IDS evaluation study, that is, to evaluate a given IDS property (see Section 3.4).

3.2.6 Pure Malicious/Pure Benign/Mixed → Trace Form → Trace Generation

Under trace generation, we understand the process of generating traces by the IDS evaluator himself. To avoid the issues with acquiring traces (see Section 3.2.5), researchers generate traces in a testbed environment, or deploy a honeypot in order to capture malicious activities.

Trace Generation → Testbed Environment

Different ways to generate traces that contain benign and malicious workloads in a testbed environment include using the previously mentioned methods for generating workloads in executable form (e.g., use of workload drivers, manual generation) and capturing and storing the executed workloads in trace files. The generation of traces in a testbed environment is challenging due to several concerns. For instance, the costs of building a testbed that scales to realistic production environments may be high. Further, the method for trace generation may produce faulty or simplistic workloads. Sommer et al. [SP10] warn that activities captured in small testbed environments differ fundamentaly from activities in a real-life environment. Finally, the methods used to generate traces are not flexible enough to timely follow the attack and benign activity trends.

Table 3.4: Repositories of publicly available traces.

Trace repository	Content	Activities	Labeled	Realistic	Anonymized	Metadata	Access restrictions
CAIDA	Mixed	Network	No	Yes	Partially[(a)]	Yes	Partial[(b)]
DEFCON	Pure malicious	Network	No	No	No	No	No
DARPA/KDD'99	Mixed	Network/Host	Yes	No	No	Yes	No
ITA	Pure benign	Network	n/a	Yes	Partially[(c)]	No	No
LBNL	Pure benign	Network	n/a	Yes	Yes	Yes	No
MAWILab	Mixed	Network	Yes	Yes	Yes	Yes	No

Categorization criteria

Criteria	Description
Content	The traces hosted in a repository may contain only benign activities (pure benign), only attacks (pure malicious), or both benign activities and attacks (mixed, see Figure 3.1).
Activities	The traces hosted in a repository may contain network and/or host activities. The former are needed for evaluating network-based IDSs and the latter for evaluating host-based IDSs.
Labeled	The attacks recorded in the traces hosted in a repository may, or may not be, labeled. Labeled attacks enable an IDS evaluator to observe whether the IDS under test detects the recorded attacks.
Realistic	The traces hosted in a repository may, or may not be, realistic. A trace is considered to be realistic if it has been captured during regular operation of an environment and has not been modified [SSTG12].
Anonymized	The traces hosted in a repository may, or may not be, anonymized. Anonymized traces may lack information that is crucial for intrusion detection (see Section 3.2.5).
Metadata	Metadata on how activities stored in trace files have been recorded and anonymized may, or may not be, provided. Metadata is important for accurately interpreting results from IDS evaluation experiments (see Section 3.2.5).
Access restrictions	The traces hosted in a repository may be available to the general public or only to certain individuals that satisfy specific requirements, for example, employment by a research organization or a government agency.

Selection guidelines

1) Select the trace repositories with the appropriate value of the criterion 'activities' with respect to the type of the tested IDS (i.e., network- or host-based IDS); that is, 'network' for evaluating network-based IDSs, 'host' for evaluating host-based IDSs.
2) Select the trace repositories with the appropriate value of the criterion 'content' with respect to the evaluated IDS property, for example, 'pure malicious' for evaluating attack detection coverage, 'mixed' for evaluating attack detection accuracy, and so on. In Section 3.4, we provide details on the use of pure malicious, pure benign, or mixed workloads for evaluating different IDS properties (see Table 3.7).
3) Select the repository with values of the criteria 'labeled', 'realistic', 'anonymized', 'metadata', and 'access restrictions' such that the benefit of using the repository is maximal (see Table 3.4, section 'Categorization criteria', for an overview of the benefits of a trace repository fulfilling or not fulfilling the previously mentioned criteria).

(a) The IP addresses recorded in some trace files are anonymized.
(b) Some trace files are available only to members of CAIDA's membership program.
(c) Some trace files are anonymized such that IP addresses are modified and all packet contents are removed.

A common approach to alleviate the previously mentioned issue of generating faulty and unrealistic workloads is to observe the network and/or host activities in a real-life production environment in order to construct a realistic activity model. The latter can then be used as a basis for the generation of live workloads that closely resemble the real observed workloads for the purpose of recording trace files.

The previously mentioned DARPA datasets were generated according to the above approach. More specifically, Cunningham et al. [CLF+99] observed the network activities in an Air Force base by deploying network traffic sniffers to record types and amounts of used network services. They also identified representative workstation users (e.g., programmers, secretaries, system administrators) and associated host workloads, so that they could recreate the activity of these users. As malicious workloads, besides surveillance/probing attacks, they used scripted and real attackers to execute a set of exploits against the testbed environment. Overall, Cunningham et al. [CLF+99] demonstrated the use of manual benign workload generation (Section 3.2.2) based on a realistic activity model to generate and record benign workloads, and live execution of attacks from an exploit database (Section 3.2.3) to generate and record malicious workloads.

Although useful, the use of the above mentioned approach for the generation of realistic traces results in one-time datasets, that is, datasets that resemble the real-world only for a given (short) time period after the trace generation. Given that the characteristics of intrusions and of benign workloads are rapidly changing over time, one-time datasets are considered as inappropriate for a representative IDS evaluation.

The above issue has motivated a major current research direction focusing on the generation of traces in a testbed environment, in a customizable and scientifically rigorous manner. To this end, Shiravi et al. [SSTG12] in 2012 proposed the use of workload profiles enabling the specification and customization of malicious and benign network traffic that can be captured in trace files for evaluating network-based IDSs. Shiravi et al. [SSTG12] introduced α-profiles for the specification of attack scenarios with attack description languages, and β-profiles for the specification of mathematical distributions or behaviors of certain entities (e.g., distribution of network packet sizes, payload sizes, and similar).

According to Shiravi et al. [SSTG12], the $\alpha-$ and $\beta-$profiles support the generation of datasets that can be modified, extended, and reproduced, so that they remain up-to-date as network usage trends change over time. Using the specifications defined in the profiles, one can generate both malicious and benign workloads in a testbed environment for recording. We refer the reader to [SSTG12] for further information on the practical use of such profiles.

In Figure 3.3a and Figure 3.3b, we depict distributions of Hyper Text Transfer Protocol (HTTP) requests made over a period of 24 hours by a real user browsing websites and by an agent using $\beta-$ profiles, respectively. The distributions were observed by Shiravi et al. [SSTG12] who showed that the measurement data can be best modeled using a Beta distribution as shown in Figure 3.3a. Given the great similarity between the histograms depicted in Figure 3.3a and Figure 3.3b, one can conclude that the

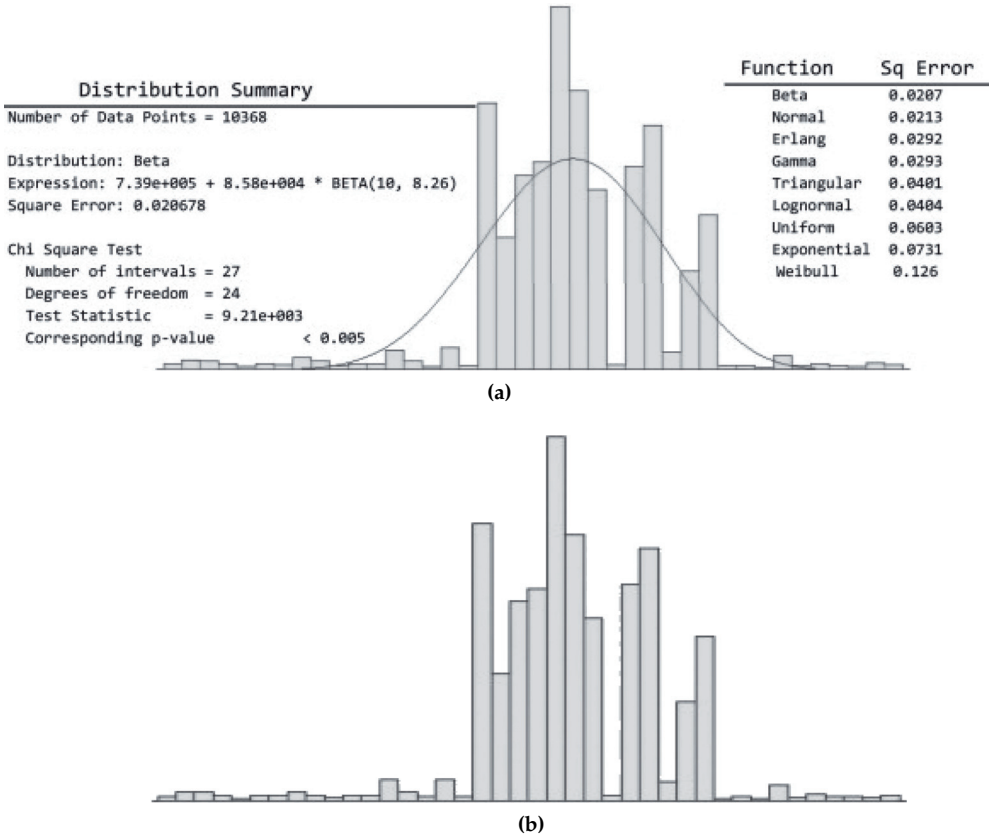


Figure 3.3: Histogram of HTTP requests made by (a) a real user, and (b) an agent using $\beta-$profiles [cf. [SSTG12]].

$\beta-$profiles proposed by Shiravi et al. [SSTG12] can be used for generating benign workloads that closely resemble real workloads.

Trace Generation → Honeypots

By mimicking real systems and/or vulnerable services, honeypots enable the interaction and recording of host and/or network malicious activities performed by an attacker without revealing their purpose. Since honeypots are usually isolated from production platforms, most of the interactions that they observe are malicious, making honeypots ideal for generation of pure malicious traces. However, given that honeypots interact with real attackers, the outcome of a trace generation campaign performed by using a honeypot (e.g., amount and types of recorded attacks) is uncertain since it cannot be planned in advance and controlled.

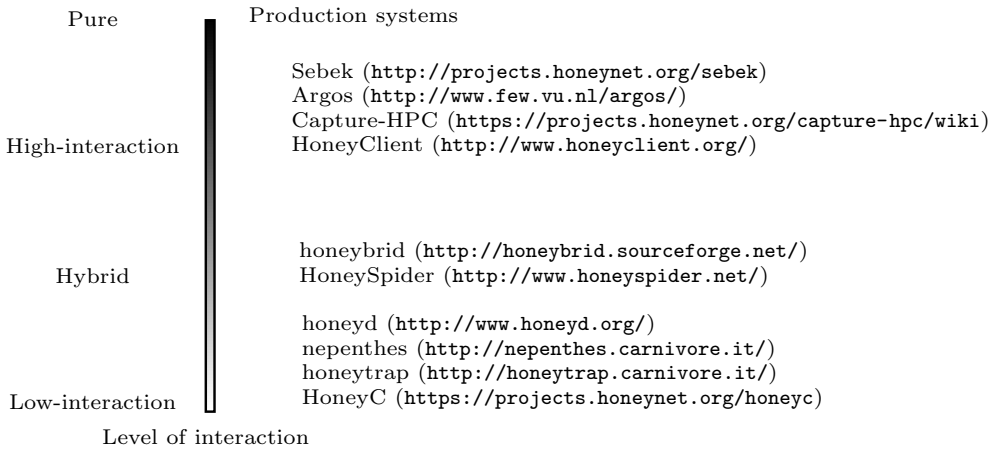


Figure 3.4: Honeypots of different levels of interaction.

Based on their level of interaction with attackers, honeypots can be categorized into low-interaction, high-interaction, and pure honeypots. Low-interaction honeypots mimic only specific vulnerable services by using scripts, high-interaction honeypots mimic production systems by using actual operating systems, and pure honeypots are full-fledged production systems equipped with logging tools.

Low-interaction honeypots are not flexible, can be easily detected by attackers, and cannot interact with zero-day attacks (see Section 2.1.2). However, they are not expensive to maintain and are useful for recording malicious activities that can be easily labeled. High-interaction and pure honeypots are flexible, can interact with zero-day attacks, and are not easily detectable, however, they are expensive to maintain and require time-consuming analysis of recorded activities for the purpose of labeling recorded attacks in order to construct "ground truth". There are also hybrid honeypots, which combine the advantages of low- and high-interaction honeypots. In Figure 3.4, we present commonly used honeypots of the previously mentioned levels of interaction.

3.3 Metrics

In Figure 3.5, we depict the metrics part of the IDS evaluation design space. We distinguish between two metric categories: *(i)* performance-related metrics, and *(ii)* security-related metrics (see Section 1.2.2). Under *performance-related* metrics, we consider metrics that quantify the non-functional properties of an IDS under test, such as capacity (see, for example, [ML12]) and resource consumption (see, for example, [SJP06]). In this chapter, we focus on security-related metrics. Under *security-related* metrics, we consider metrics that quantify the attack detection accuracy of an IDS.

We distinguish between *basic* and *composite* security-related metrics. We provide in

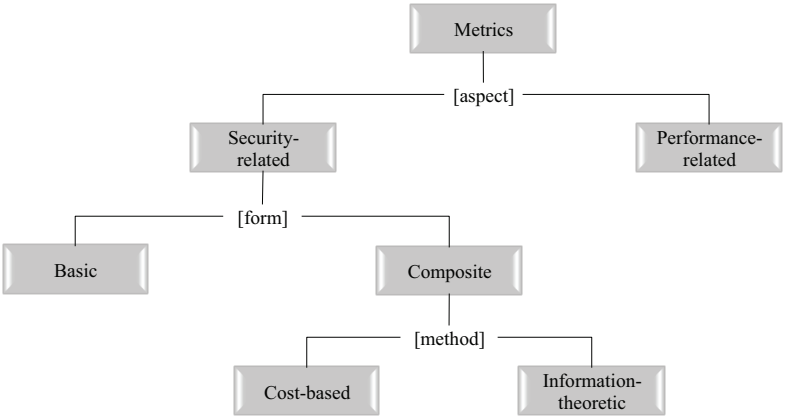


Figure 3.5: IDS evaluation design space: Metrics [There are two types of metrics with respect to the aspect of IDS behavior they quantify: *security-related* (quantify IDS attack detection accuracy) and *performance-related* (quantify non-functional IDS properties) • There are two types of security-related metrics with respect to metric form: *basic* (Section 3.3.1) and *composite* (metrics derived from basic metrics, Section 3.3.2) • There are two types of composite metrics with respect to used measurement method (Section 3.3.2): *cost-based* and *information-theoretic*].

Table 3.5 an overview of the most commonly used basic and composite security-related metrics. We also show the notation, formulas, and value domains of used symbols (including variables). In Table 3.5, P and p denote a probability, $\mathbb{R}$ denotes the set of real numbers, $\mathbb{R}^+$ denotes the set of positive real numbers excluding zero, and $\mathbb{R}_0^+$ denotes the set of positive real numbers including zero.

3.3.1 Security-related → Basic

The basic metrics quantify various individual attack detection properties. Although they are quantified individually, these properties need to be analyzed together in order to accurately characterize the attack detection efficiency of an IDS. For instance, the false negative rate $\beta = P(\neg A|I)$ quantifies the probability that an IDS does not generate an alert when an intrusion occurs; therefore, the true positive rate $1 - \beta = 1 - P(\neg A|I) = P(A|I)$ quantifies the probability that an alert generated by an IDS is really an intrusion. The false positive rate $\alpha = P(A|\neg I)$ quantifies the probability that an alert generated by an IDS is not an intrusion, but a regular benign activity; therefore, the true negative rate $1 - \alpha = 1 - P(A|\neg I) = P(\neg A|\neg I)$ quantifies the probability that an IDS does not generate an alert when an intrusion does not occur. In IDS evaluation experiments, the output of the IDS under test is compared with "ground truth" information in order to calculate the basic metrics (see Section 3.2.4).

Other basic metrics are the positive predictive value (PPV) and the negative predictive value (NPV). The first quantifies the probability that there is an intrusion when an IDS generates an alert whereas the latter quantifies the probability that there is no

Table 3.5: Common metrics for quantifying IDS attack detection accuracy.

Metric form	Metric	Annotation/Formula
Basic	False negative rate	$\beta = P(\neg A\|I)$
	True positive rate	$1 - \beta = 1 - P(\neg A\|I) = P(A\|I)$
	False positive rate	$\alpha = P(A\|\neg I)$
	True negative rate	$1 - \alpha = 1 - P(A\|\neg I) = P(\neg A\|\neg I)$
	Positive predictive value	$P(I\|A) = \frac{P(I)P(A\|I)}{P(I)P(A\|I)+P(\neg I)P(A\|\neg I)}$
	Negative predictive value	$P(\neg I\|\neg A) = \frac{P(\neg I)P(\neg A\|\neg I)}{P(\neg I)P(\neg A\|\neg I)+P(I)P(\neg A\|I)}$
Composite	Expected cost	$C_{exp} = Min(C\beta B, (1-\alpha)(1-B)) + Min(C(1-\beta)B, \alpha(1-B))$
	Intrusion detection capability	$C_{ID} = \frac{I(X;Y)}{H(X)}$

Notations and properties of used symbols

Symbol	Meaning	Formula/Value domain
A	*Alert event*: An IDS generates an attack alert	n/a
I	*Intrusion event*: An attack is performed	n/a
C_α	Cost of an IDS generating an alert when an intrusion has not occured	$C_\alpha \in \mathbb{R}_0^+$
C_β	Cost of an IDS failing to detect an intrusion	$C_\beta \in \mathbb{R}^+$
C	*Cost ratio*: The ratio between the costs C_α and C_β	$C_\beta/C_\alpha : C \in \mathbb{R}_0^+$
B	*Base rate*: Prior probability that an intrusion event occurs	$P(I) : B \in \mathbb{R} \rightarrow [0;1]$
X	*IDS input*: Discrete random variable used to model input to an IDS such that $X = 0$ represents a benign activity and $X = 1$ represents a malicious activity (i.e., an intrusion)	$X = 0 \vee X = 1$
Y	*IDS output*: Discrete random variable used to model the generation of alerts by an IDS such that $Y = 0$ represents no alert and $Y = 1$ represents an alert	$Y = 0 \vee Y = 1$
$H(X)$	*Uncertainty of* X: Entropy measure quantifying the uncertainty of the IDS input X	$-\sum_x p(x)logp(x) : x = 0 \vee x = 1, p(x) \in \mathbb{R} \rightarrow [0;1]$
$I(X;Y)$	*Mutual information*: The amount of information shared between the random variables X and Y (i.e., the amount of reduction of the uncertainty of the IDS input (X) after the IDS output (Y) is known)	$\sum_x \sum_y p(x,y)log\frac{p(x,y)}{p(x)p(y)} : x = 0 \vee x = 1, y = 0 \vee y = 1, p(x) \wedge p(y) \wedge p(x,y) \in \mathbb{R} \rightarrow [0;1], 0 \leq I(X;Y) \leq H(X)$

intrusion when an IDS does not generate an alert. These metrics are calculated by using the Bayesian theorem for calculating a conditional probability (see Table 3.5). PPV and

NPV are interesting from a usability perspective, for example, in situations when an intrusion alert triggers an attack response. In such situations, low values of PPV and NPV indicate that the considered IDS is not optimal for deployment. For example, a low value of PPV (therefore a high value of its complement $1 - P(I|A) = P(\neg I|A)$) indicates that the considered IDS may often cause the triggering of attack response actions when no real attacks have actually occured.

3.3.2 Security-related → Composite

Security researchers often combine the basic metrics in order to analyze relationships between them. Such an analysis is used to discover an optimal IDS *operating point* (i.e., an IDS configuration which yields optimal values of both the true and false positive detection rate) or to compare multiple IDSs. In this section, we focus on comparing the applicability of composite security-related metrics for the purpose of comparing IDSs, which includes the identification of optimal IDS operating points.

A Receiver Operating Characteristic (ROC) curve plots true positive rate against the corresponding false positive rate exhibited by a detector. In the context of IDSs, a ROC curve depicts multiple IDS operating points of an IDS under test and, as such, it is useful for identifying an optimal operating point or for comparing multiple IDSs.

An open issue is how to determine a proper unit and measurement granularity for the false positive and true positive rates based on which a ROC curve is plotted. Different units of measurement might yield different rates and therefore, the selection of a proper unit is considered as a task that needs to be performed with care. Gu et al. [GFD+06] acknowledge the importance and scope of the above issue by referring to it as: "*general problem for all the existing [IDS] evaluation metrics*". Gu et al. [GFD+06] discuss this issue in the context of the evaluation of network-based IDSs. They state that depending on the unit of analysis in a network-based IDS, at least two different units of measurement exist (i.e., a unit of packet and flow), which makes the comparison of IDSs with these units of analysis challenging. Gu et al. [GFD+06] recommend the conversion of different units of measurement to the same unit when possible for a fair and meaningful IDS comparison (e.g., conversion of a packet-level to a flow-level unit by defining a flow as malicious when it contains a malicious packet). Next, we analyze and demonstrate the use of ROC curves and related metrics through a case study scenario.

Case study #1: Let's consider the comparison of two IDSs, IDS_1 and IDS_2, and analyze the relationship between the true positive ($1 - \beta$) and the false positive (α) detection rate. We assume that for IDS_1, $1 - \beta$ is related to α with a power function (i.e., $1 - \beta = \alpha^k$) such that $k = 0.002182$. We assume that for IDS_2, $1 - \beta$ is related to α with an exponential function (i.e., $1 - \beta = 1 - 0.00765e^{-208.32\alpha}$). We obtain the values of k, α, and the coefficients of the exponential function from [GU01]. We calculate the values of $1 - \beta$ for IDS_1 and IDS_2 for $\alpha = \{0.005, 0.010, 0.015\}$. The respective values are shown in Table 3.6.

In Figure 3.6a, we depict the ROC curves that express the relationship between $1 - \beta$

Table 3.6: Values of $1-\beta$, PPV_{ID}, C_{exp}, C_{rec}, and C_{ID} for IDS_1 and IDS_2.

		IDS$_1$				**IDS$_2$**			
α	PPV_{ZRC}	$1-\beta$	PPV_{ID}	$C_{exp/rec}$	C_{ID}	$1-\beta$	PPV_{ID}	$C_{exp/rec}$	C_{ID}
0.005	0,9569	0.9885	0,9565	**0.016**	**0.9159**	0.973	0,9558	0.032	**0.8867**
0.010	0,9174	0.99	0,9167	0.019	0.8807	0.99047	0,9167	0.019	0.8817
0.015	0,8811	0.9909	0,8801	0.022	0.8509	0.99664	0,8807	**0.017**	0.8635

and α for IDS_1 and IDS_2. The ROC curves intersect approximately at $1-\beta = 0.99$ and $\alpha = 0.01$. Thus, the better IDS cannot be identified in a straightforward manner. An IDS is considered as better if it features a higher true positive detection rate ($1-\beta$) at all operating points along the ROC curve.

An intuitive solution to the above problem, as suggested by Durst et al. [DCW+99], is to compare the area under the ROC curves (i.e., $AUC_1 : \int_{\alpha=0.005}^{\alpha=0.015} \alpha^{0.002182} \mathrm{d}\alpha$ and $AUC_2 : \int_{\alpha=0.005}^{\alpha=0.015} (1-0.00765e^{-208.32\alpha}) \mathrm{d}\alpha$). However, Gu et al. [GFD+06] consider such a comparison as unfair, since it is based on all operating points of the compared IDSs, while in reality a given IDS is always configured according to a single operating point.

The ROC curves depicted in Figure 3.6a do not express the impact of the rate of occurence of intrusion events ($B = P(I)$), known as *base rate*, on α and $1-\beta$. The attack detection performance of an IDS should be assessed with respect to a base rate measure in order for such an assessment to be accurate (see [Axe00]). The error occurring when α and $1-\beta$ are assessed without taking the base rate into account is known as the *base rate fallacy*.

In order to address the above issue, Nasr et al. [NKF12] propose a metric called *intrusion detection effectiveness* (E_{ID}). E_{ID} is calculated based on comparing the ideal and actual performance of an IDS depicted in the form of IDS operation curves called zero reference curve (ZRC) and actual IDS operation curve, respectively. An IDS operation curve plots PPV, which contains measure of the base rate B (see Table 3.5), against α. Given a specific value of B, the ZRC plots PPV (denoted by PPV_{ZRC}) calculated assuming an ideal operation of the tested IDS; that is, the IDS does not miss attacks ($1-\beta = 1$). The actual IDS operation curve plots the actual PPV (denoted by PPV_{ID}) exhibited by the IDS. The value of E_{ID} is the normalized variance between the ZRC and the actual IDS operation curve over the interval [0, T_{FP}], where T_{FP} is the maximum acceptable α exhibited by the IDS; that is, $E_{ID} = \frac{1}{\int_0^{T_{FP}} PPV_{ZRC} \mathrm{d}\alpha} (\int_0^{T_{FP}} PPV_{ZRC} \mathrm{d}\alpha - \int_0^{T_{FP}} PPV_{ID} \mathrm{d}\alpha)$, $E_{ID} \in [0;1]$, such that the lesser E_{ID} the better the performance of the IDS under test.

In Table 3.6, we present PPV_{ZRC} and PPV_{ID} for IDS_1 and IDS_2, calculated assuming that $B = 0.1$. In Figure 3.6b and Figure 3.6c (the axes are in logarithmic scale), we depict the ZRC and the actual IDS operation curve for IDS_1 and IDS_2. These curves are very similar due to the high PPVs (i.e., close to the ideal PPV — PPV_{ZRC}) exhibited by IDS_1 and IDS_2. We calculate E_{ID} of 0.0004 for IDS_1 and E_{ID} of 0.0011

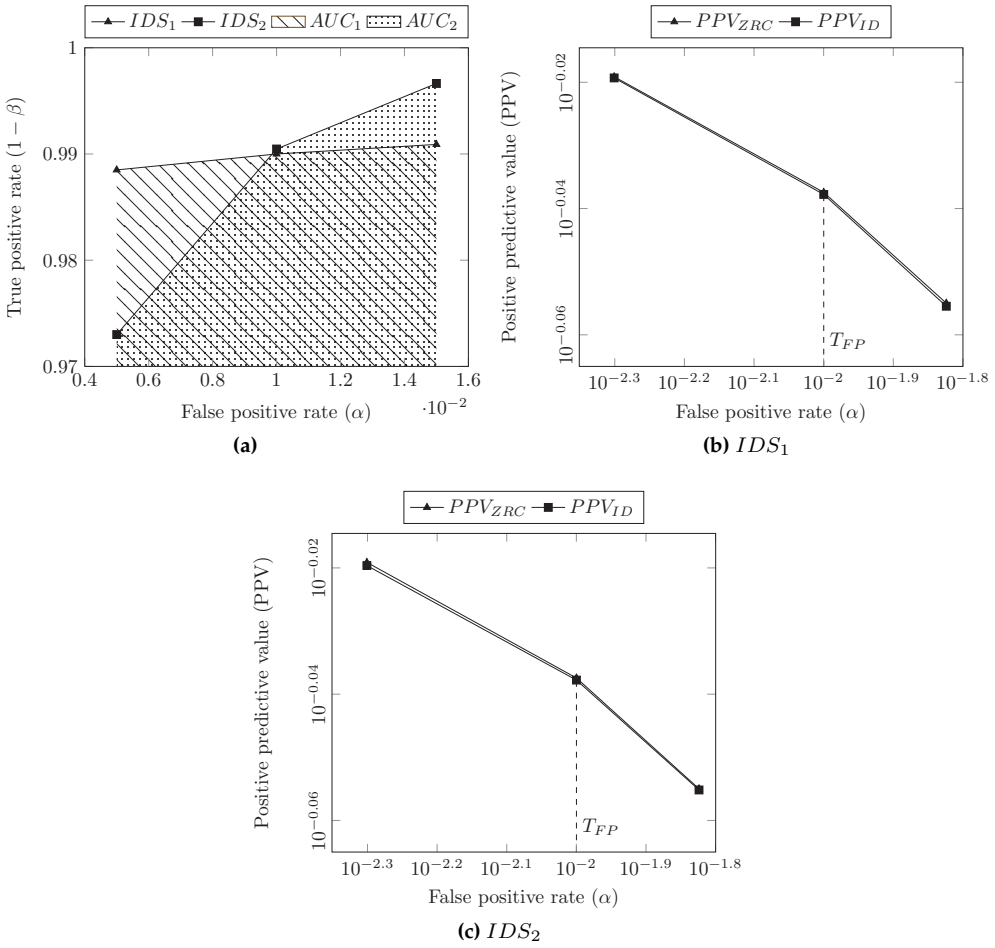


Figure 3.6: IDS comparison with (a) ROC curves, (b) – (c) intrusion detection effectiveness metric.

for IDS_2 ($T_{FP} = 0.01$), based on which we conclude that IDS_1 performs better.

Although E_{ID} expresses the impact of the base rate on α and $1 - \beta$, it suffers from the same issue as the metric proposed by Durst et al. [DCW+99]; that is, a comparison of IDSs based on E_{ID} may be misguiding, since it is based on multiple operating points of the compared IDSs (see [GFD+06]).

Cost-based and Information-theoretic Metrics

Due to the above mentioned issues, researchers have proposed novel metrics that can be classified into two main categories: *(i)* metrics that use cost-based measurement

methods, and *(ii)* metrics that use information-theory measurement methods (see Figure 3.5). In the following, we discuss metrics that belong to these categories focusing on the *expected cost* and *intrusion detection capability* metrics described in the seminal works of Gaffney et al. [GU01] and Gu et al. [GFD+06].

Cost-based metrics Gaffney et al. [GU01] propose the measure of cost as an IDS evaluation parameter. They combine ROC curve analysis with cost estimation by associating an estimated cost with each IDS operating point. The measure of cost is relevant in scenarios where a response that may be costly is taken (e.g., stopping a network service) when an IDS generates an attack alert. Gaffney et al. [GU01] introduce a cost ratio $C = C_\beta/C_\alpha$, where C_α is the cost of an IDS alert when an intrusion has not occured, and C_β is the cost of not detecting an intrusion when it has occurred. Gaffney et al. [GU01] use the cost ratio to calculate the expected cost C_{exp} of an IDS operating at a given operating point (see Table 3.5). Using C_{exp}, one can compare IDSs by comparing the estimated costs when each IDS operates at its optimal operating point. The IDS that has lower C_{exp} associated with its optimal operating point is considered as better. An IDS operating point is considered as optimal if it has the lowest C_{exp} associated with it compared to the other operating points.

The formula of C_{exp} (see Table 3.5) can be obtained by analyzing the decision tree depicted in Figure 3.7a. The decision tree shows the costs that may be incurred by an IDS (e.g., C_α and C_β) with respect to the operation of the IDS (i.e., generation of alerts) and the responses that can be taken; uncertain events (e.g., the generation of an alert) are depicted by circles and actions are depicted by squares. In Figure 3.7a, we depict the probabilities $p_1 = P(A)$, $p_2 = P(I|A) = PPV$, and $p_3 = P(I|\neg A)$ (see Table 3.5). The formula of C_{exp} is obtained by "rolling back" the tree depicted in Figure 3.7a; that is, from right to left, the expected cost at an event node is the sum of products of probabilities and costs for each branch, and the expected cost at an action node is the minimum of expected costs on its branches. The formula of C_{exp} shown in Table 3.5 can be derived using the basic algebra of probability theory (see [GU01]).

A recent work proposing a cost-based IDS evaluation metric is [Men12]. Meng et al. [Men12] propose a metric called *relative expected cost* (C_{rec}). This metric is intended for comparing modern IDSs that use false alert filters (see, for example, [CLC+10]). A false alert filter detects false alerts generated by an IDS. The response taken when a false alert filter labels an alert as false is filtering out the alert before it is reported by the IDS. C_{rec} is based on the previously discussed expected cost metric. In contrast to C_{exp}, C_{rec} measures cost associated with the accuracy of an IDS's false alert filter at classifying alerts as true or false, which can be used as an IDS comparison parameter. C_{rec} can be associated with each IDS operating point on a ROC curve and can be used for comparing IDSs same as C_{exp}.

The formula of C_{rec} ($C_{rec} = C\beta B + \alpha(1 - B)$) can be obtained in a way similar to obtaining the formula of C_{exp}, that is, by "rolling back" the decision tree depicted in Figure 3.7b (see [Men12]). This tree is a modified version of the tree depicted in Figure 3.7a. In Figure 3.7b, p_1 denotes the probability that the false alert filter reports a true alert, p_2 denotes the conditional probability of true alert given that the filter

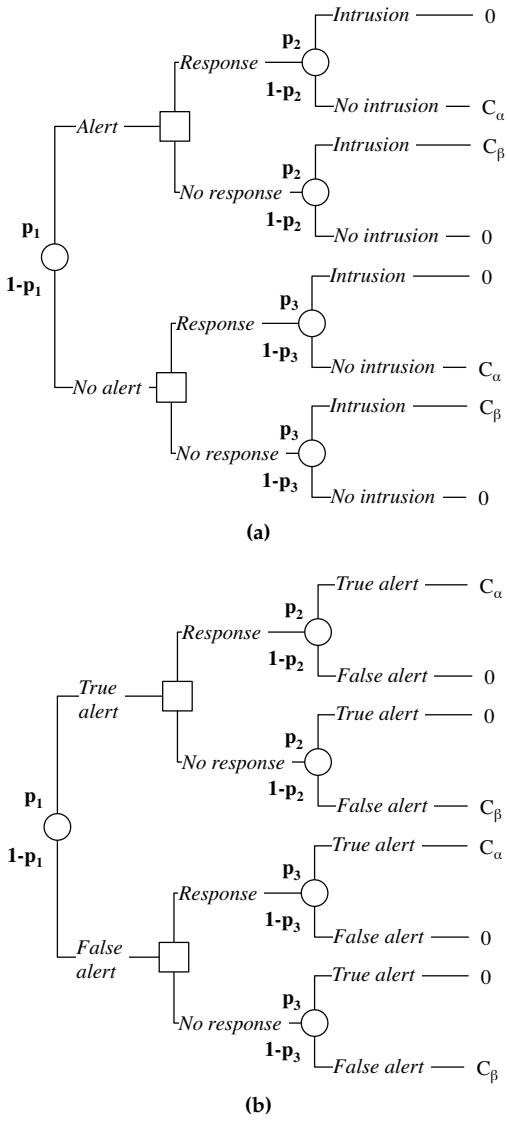


Figure 3.7: Decision tree for calculating (a) expected cost, and (b) relative expected cost.

reports a true alert, and p_3 denotes the conditional probability of true alert given that the filter reports a false alert. In the context of the work of Meng et al. [Men12], α and β denote the false positive and false negative rate exhibited by a false alert filter. Further, B denotes the prior probability of a false alert, and C_α and C_β denote the cost of classifying a false alert as a true alert, and the cost of classifying and filtering out a true alert as a false alert, respectively. C is the cost ratio C_β/C_α.

Although the discussed cost-based metrics enable straightforward comparison of IDSs, they depend on the cost ratio C. To calculate the cost ratio, one would need a cost-analysis model that can estimate C_α and C_β, which might be difficult to construct in reality. Cost-analysis models take parameters into consideration that might not be easy to measure, or might not be measurable at all (e.g., man-hours, system down-time). Further, C_{exp} and C_{rec} enable the comparison of IDSs based on a subjective measure making the metrics unsuitable for objective comparisons [GFD+06]. However, cost-based metrics may be of value when the relationships between the different attack detection costs (e.g., cost of missing an attack, cost of a false alert) can be estimated and when such estimations would be considered as sufficently accurate. For instance, given a statement such as "*a false alert is twice as costly as a missed attack*", a cost-based metric would be crucial to identify an optimal IDS operating point. Next, we demonstrate the use of the expected cost metric (C_{exp}) and the relative expected cost metric (C_{rec}) for comparing IDSs through a case study scenario.

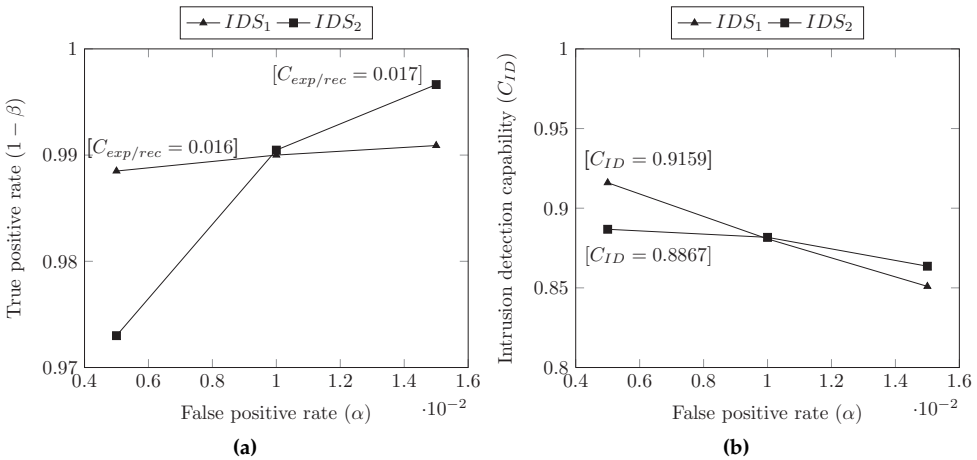


Figure 3.8: IDS comparison with (a) expected cost and relative expected cost metric, (b) intrusion detection capability metric.

Case study #2: First, we compare IDS_1 and IDS_2 (see Case study #1) using C_{exp}. The IDS that has lower C_{exp} associated with its optimal operating point (i.e., the point that has the lowest C_{exp} associated with it) is considered as better. To determine the optimal operating points of IDS_1 and IDS_2, we calculate C_{exp} for each operating point of the two IDSs. To calculate C_{exp}, we assume that $C = 10$ (i.e., the cost of not responding to an attack is 10 times higher than the cost of responding to a false alert) and $B = 0.10$. We present the values of C_{exp} in Table 3.6. The optimal operating point of IDS_1 is $(0.005, 0.9885)$, and of IDS_2 is $(0.015, 0.99664)$. Since the minimal C_{exp} of IDS_1 (0.016) is smaller than the minimal C_{exp} of IDS_2 (0.017), we conclude that IDS_1 performs better. In Figure 3.8a, we depict the ROC curves annotated with the minimal C_{exp} of IDS_1 and IDS_2.

We now compare IDS_1 and IDS_2 using C_{rec}. The IDS that has lower C_{rec} associated with its optimal operating point (i.e., the point that has the lowest C_{rec} associated with it) is considered as better. To calculate C_{rec}, we assume that $C = 10$, $B = 0.10$, and that IDS_1 and IDS_2 use false alert filters. We present the values of C_{rec} in Table 3.6. For the sake of simplicity, we assume that α and $1 - \beta$ exhibited by IDS_1 and IDS_2 (see Case study #1 and Table 3.6) correspond to α and $1 - \beta$ exhibited by the IDS's false alert filters. This results in identical values of C_{rec} and C_{exp}. Using the same approach for comparing IDSs as the one we used when comparing IDSs using C_{exp}, we conclude that IDS_1 performs better than IDS_2 in terms of incurred costs that are associated with the accuracy of the IDSs' false alert filters. In Figure 3.8a, we depict the ROC curves annotated with the minimal C_{rec} of IDS_1 and IDS_2.

Information-theoretic metrics Another approach for evaluating the attack detection accuracy of an IDS is the information-theoretic approach. Gu et al. [GFD+06] propose a metric called intrusion detection capability (C_{ID}, see Table 3.5). They model the input to an IDS as a stream of a random variable X ($X = 1$ denotes an intrusion, $X = 0$ denotes benign activity), and the IDS output as a stream of a random variable Y ($Y = 1$ denotes IDS alert, $Y = 0$ denotes no alert). The input and output stream have a certain degree of uncertainty reflected by the entropies $H(X)$ and $H(Y)$, respectively. Thus, Gu et al. [GFD+06] model the number of correct guesses of an IDS, that is, $I(X;Y)$, as mutual shared information between the random variables X and Y, that is, $I(X;Y) = H(X) - H(X|Y)$. C_{ID} is obtained by normalizing $I(X;Y)$ with $H(X)$ (see Table 3.5).

C_{ID} incorporates the uncertainty of the input stream $H(X)$ (i.e., the distribution of intrusions in the IDS input) and the accuracy of an IDS under test $I(X;Y)$; that is, C_{ID} incorporates the base rate B, the true positive rate $(1 - \beta)$, and the false positive rate (α). For the definition of the relationship between C_{ID}, on the one hand, and B, $1 - \beta$, and α, on the other hand, we refer the reader to [GFD+06]. Given this relationship, a value of C_{ID} may be assigned to any operating point of an IDS on the ROC curve. With this assignment, one obtains a new curve (i.e., a C_{ID} curve).

The information-theoretic approach is generic and enables the evaluation of IDSs of different designs. For instance, Meng et al. [YMLFK13] discuss the application of the intrusion detection capability metric for quantifying the attack detection accuracy of IDSs that use false alert filters. They fine-tune this metric and develop a metric called *false alarm reduction capability*, $RC_{FA} = \frac{I(X;Y)}{H(X)}$. In the context of the work of Meng et al. [YMLFK13], X denotes the input to a false alert filter (i.e., alerts generated by an IDS) and Y denotes the filter's output. Meng et al. [YMLFK13] show that the information-theoretic approach can be applied in practice for evaluating IDSs that use false alert filters by evaluating Snort [Roe99] configured to use a variety of false alert filters. Next, we show how one can compare IDSs using the intrusion detection capability metric (C_{ID}) through a case study scenario.

Case study #3: Assuming a base rate of $B = 0.10$, we calculated C_{ID} for the operating points of IDS_1 and IDS_2 (see Case study #1) presented in Table 3.6. In Figure 3.8b,

we depict the C_{ID} curves of IDS_1 and IDS_2. A C_{ID} curve provides a straightforward identification of the optimal operating point of an IDS (i.e., the point that marks the highest C_{ID}). One can compare IDSs by analyzing the maximum C_{ID} of each IDS and considering as better performing the IDS whose optimal operating point has higher C_{ID} associated with it. From Table 3.6, one would consider IDS_1 to perform better since it has greater maximum C_{ID} (0.9159) than the maximum C_{ID} of IDS_2 (0.8867). In contrast to the previously discussed expected cost metric, C_{ID} is not based on subjective measures such as cost, which makes it suitable for objective comparison of IDSs.

3.4 Measurement Methodology

Under measurement methodology, we understand the specification of the IDS properties of interest (e.g., attack detection accuracy, capacity) and of the employed workloads and metrics for evaluating the properties. After examining IDS evaluation experiments covering different types of IDSs, we identified nine IDS properties that are commonly considered in practice. In Table 3.7, we present these properties, some of which are grouped into categories. For the sake of clarity, we also present definitions of the IDS properties attack detection accuracy, attack coverage, performance overhead, and workload processing capacity. Further, in Table 3.7, we provide an overview of the workload and metric requirements (see Section 3.2 and Section 3.3) for evaluating the different properties. In Table 3.8, we list references to the surveyed publications where representative studies that investigate the respective properties can be found. Table 3.8 compares the surveyed work in terms of types of tested IDSs (see Table 2.1) and considered IDS properties. This enables the identification of common trends in evaluating IDS properties for different types of IDSs. Next, we discuss such trends and provide recommendations and key best practices, which we identified based on reported benefits of applying the practices. We also present observed quantitative values (e.g., acceptable performance overheads and attack detection speeds) and relevant observations (e.g., evasion techniques to which current IDSs are vulnerable) that may serve as reference points for designing and evaluating future IDSs.

Attack detection accuracy/attack coverage: As expected, these properties are evaluated for IDSs of all types. Due to the longevity of their presence on the IDS evaluation scene, the DARPA and the KDD'99 Cup datasets (see Section 3.2.5) represent at this time standard workloads for comparing novel anomaly-based IDSs with their past counterparts in terms of their attack detection accuracy. For the sake of representativeness, we recommend the evaluation of a single IDS using not the DARPA and the KDD'99 Cup datasets, but workloads that contain current attacks (see Section 3.2). We observed that the attack detection rates of IDSs reported in the surveyed work vary greatly, that is, between 8% and 97%, measures which largely depend on the configurations of the tested IDSs (see Section 3.3) and the applied evaluation methodologies.

Attack detection and reporting speed: This property is typically evaluated for dis-

Table 3.7: IDS evaluation design space: Measurement methodology.

IDS Property	Workloads	Metrics	
	[content]	[aspect]	[form]
Attack detection-related			
Attack detection accuracy	mixed	security-related	basic, composite
Attack coverage	pure malicious	security-related	basic
Resistance to evasion techniques	pure malicious, mixed	security-related	basic
Attack detection and reporting speed	mixed	performance-related	n/a
Resource consumption-related			
CPU consumption Memory consumption Network consumption	pure benign	performance-related	n/a
Performance overhead	pure benign	performance-related	n/a
Workload processing capacity	pure benign	performance-related	n/a

Definitions of IDS properties

IDS Property	Definition
Attack detection accuracy	The attack detection accuracy of an IDS in the presence of mixed workloads.
Attack coverage	The attack detection accuracy of an IDS in the presence of attacks without any background benign activity.
Performance overhead	The overhead incurred by an IDS on the system and/or network environment where it is deployed. Under overhead, we understand performance degradation of users' tasks/operations caused by: *(a)* consumption of system resources (e.g., CPU, memory) by the IDS, and/or *(b)* interception and analysis of the workloads of users' tasks/operations (e.g., network packets) by the IDS.
Workload processing capacity	The rate of arrival of workloads to an IDS for processing, in relation to the amount of workloads that the IDS discards (i.e., does not manage to process). For instance, in the context of network-based IDSs, capacity is normally measured as the rate of arrival of network packets to an IDS over time, in relation to the amount of discarded packets over time. The capacity of an IDS may also be defined as the maximum workload processing rate of the IDS such that there are no discarded workloads.

tributed IDSs. Each node of a distributed IDS typically reports an on-going attack to the rest of the nodes, or to a designated node, when it detects a malicious activity (see Section 2.1.2). Thus, the attack detection speed of a distributed IDS is best evaluated by measuring the time needed for the IDS to converge to a state in which all its nodes, or the designated nodes, are notified of an on-going attack (see the work of Hassanzadeh et al. [HS11] and Sen et al. [SUBP08], who consider delays up to 3 seconds as acceptable). The fast detection and reporting of an attack by an IDS node is important for the timely detection of coordinated attacks targeting multiple sites.

Table 3.8: Comparison of practices in evaluating IDS properties.

Reference	IDS Type			IDS Properties							
	Distributed (D) / Non-distributed (N)	Anomaly-based (A) / Misuse-based (M) / Hybrid (Hy)	Host-based (H) / Network-based (N) / Hybrid (Hy)	Attack detection accuracy	Attack coverage	Resistance to evasion techniques	Attack detection and reporting speed	CPU consumption	Memory consumption	Network consumption	Performance overhead	Workload processing capacity
[ATJ+10]	N	A	Hy	x								
[ALD11]	N	M	N	x								
[CM06]	N	A	H	x	x						x	
[CPX+13]	D	M	N	x			x				x	x
[GR03]	N	Hy	H	x							x	
[HS11]	D	M	N	x			x	x	x			
[JXZ+09], [JXZ+11]	N	Hy	Hy	x							x	x
[FJGS+00]	N	Hy	N	x	x							
[KZ05]	D	A	N	x							x	
[SJP06]	N	M	N	x					x			x
[LMJ07], [LDP11], [RRL+12], [Roe99]	N	A	H		x						x	
[Deh12], [RAR12], [RJX08]	N	A	H	x							x	
[SUBP08]	D	A	N	x			x			x		
[SSG08]	N	A	H	x		x					x	
[SSWX13]	D	A	N	x							x	
[ZWGW08]	N	A	H		x						x	

Resistance to evasion techniques: We observe that the evaluation of attack detection accuracy/attack coverage is prioritized over evaluation of resistance to evasion techniques. Sommer et al. [SP10] confirm this trend stating that resistance to evasion techniques is of limited importance from a practical perspective since most real-life attacks perform mass exploitation instead of targeting particular IDS flaws. However, a single successful IDS evasion attack poses the danger of a high-impact intrusion; therefore, it is a good practice to consider the resistance to evasion techniques in IDS evaluation studies.

We observed that Metasploit (see Section 3.2.3) is considered the optimal tool for executing IDS evasive attacks, which is required for evaluating resistance to evasion techniques. This is because Metasploit provides a freely available and regularly maintained attack execution environment supporting a wide range of IDS evasive techniques. By analyzing the decision-making processes of the IDSs proposed in the surveyed work for labeling an activity as benign or malicious, we observed that many IDSs are vulnerable to temporally crafted attacks (e.g., short-lived or multi-step attacks executed by delaying the execution of the attack steps, see [SSG08]). The execution of such attacks is supported by Metasploit.

Resource consumption-related: These properties are typically evaluated for IDSs deployed in resource constrained environments. An example is the evaluation of the resource consumption of a distributed IDS operating in wireless ad-hoc networks, which enables the measurement of the power consumption of its nodes. This is important since the computing nodes in a wireless ad-hoc network typically rely on a battery as a power source delivering a limited amount of power. Since the power consumption of software (i.e., of an IDS node) is difficult to measure, it can be best observed by using a model that estimates power consumption based on resource consumption measurements (see [HS11]).

The network consumption in particular is often evaluated for distributed IDSs (see [SUBP08]). The nodes of a typical distributed IDS exchange messages that contain information relevant for intrusion detection (see Section 2.1.2). This may consume a significant amount of the network bandwidth of the environment where the IDS is deployed. For instance, Sen et al. [SUBP08] consider the exchange of 120 messages over 160 seconds as a very high network consumption.

Performance overhead: This property is normally evaluated for host-based IDSs since they are known to cause performance degradation of the tasks running in the system where they are deployed. Performance overhead is evaluated by executing tasks twice, once with the tested IDS being inactive, and once with it being active (see [LMJ07]). The differences between the measured task execution speeds reveal the imposed performance overhead. Performance overhead is normally evaluated using workloads in executable form generated by workload drivers (see Section 3.2.1). Workload drivers enable the straighforward generation of live customized workloads in a repeatable manner. In general, overheads under 10%, relative to the execution time of tasks measured when the tested IDS is inactive, are considered acceptable.

Workload processing capacity: This property is normally evaluated for network-based IDSs that monitor workloads at high rates. For instance, some studies consider workload rates as high as 1 million packets per second and report percentage of discarded packets of around 50%. Workload processing capacity is best evaluated using traces or workloads generated by workload drivers since they allow for the generation of workloads at user-defined speeds. This enables the accurate measurement of the workload processing capacity of an IDS (e.g., of the particular network traffic speed such that a given network-based IDS under test does not discard packets, see Table 3.7). Further, it is a good practice to evaluate the workload processing capacity of an IDS together with its resource consumption (e.g., Sinha et al. [SJP06] observe the resource consumption of an IDS for various workload intensities). This enables an IDS evaluator to observe how resource consumption scales as workload intensity increases.

Next, we survey common approaches for evaluating the IDS properties presented in Table 3.7. In addition, we demonstrate through case studies how these approaches and the discussed best practices have been applied in the surveyed work; that is, we round-up the IDS evaluation design space by demonstrating the applicability of the different types of workloads and metrics with respect to their inherent characteristics (see Section 3.2 and Section 3.3).

3.4.1 Attack Detection-related Properties

We start by considering the properties attack coverage, attack detection accuracy, and resistance to evasion techniques. Note that we do not discuss the IDS property attack detection and reporting speed. The approach for evaluating this property is almost identical to that of attack detection accuracy with the only difference being in the used metrics (i.e., performance-related metrics that quantify time instead of security-related metrics).

Attack Coverage

The attack coverage of an IDS is typically evaluated with the goal of measuring the ability of the IDS to detect various attacks targeted at the specific system/network environment that it protects [MHL+03]. Given that the attack coverage of an IDS is its attack detection accuracy in the presence of attacks without any background benign activity (see Table 3.7), it is evaluated by using pure malicious workloads. The used pure malicious workloads should not have IDS evasive characteristics since such workloads are used for evaluating resistance to evasion techniques, which is a separate property. Further, only basic metrics that do not contain measures of false alerts are used (e.g., true positive rate). Note that an IDS might generate false alerts only in the presence of background benign activity.

Case study #4: We evaluate the attack coverage of the IDS Snort [Roe99], which uses misuse-based attack detection techniques. We consider a scenario where Snort is deployed in, and monitors the network traffic of, a server hosting web applications

Table 3.9: Attack coverage of Snort [✓ - detected / x - not detected].

Targeted vulnerability (CVE ID)	Platform	Detected
CVE-2011-3192	Apache	x
CVE-2010-1870	Apache Struts	✓
CVE-2012-0391	Apache Struts	x
CVE-2013-2251	Apache Struts	x
CVE-2013-2115/CVE-2013-1966	Apache Struts	✓
CVE-2009-0580	Apache Tomcat	x
CVE-2009-3843	Apache Tomcat	x
CVE-2010-2227	Apache Tomcat	x

using the Apache 2.2.16 server software, extended with the Tomcat 6.0.35 and Struts 1.3.10 frameworks. We tested Snort 2.9.22 using a database of signatures dated 11th July 2013. Given the architecture of the platform protected by Snort, attacks relevant for evaluating the attack coverage of Snort in this scenario are attacks targeting Apache, Tomcat, and Struts.

We used the Metasploit framework to generate pure malicious workloads (see Section 3.2.3). We use Metasploit's exploit database for the sake of convenience — Metasploit provides a readily-available exploit database that contains attack scripts targeting the platforms Apache, Tomcat, and Struts. We executed 8 attack scripts from a host that we refer to as the "attacking host". In order to eliminate benign network traffic destined for the web server, we used a firewall to isolate the web server in a way such that it was reachable only for the attacking host. In Table 3.9, we list the CVE identifiers of the vulnerabilities targeted by the executed attack scripts, and the respective target platforms. In Table 3.9, we present the results of the study. Snort detected 2 attacks, thus exhibiting a true positive rate of 0.25.

Resistance to Evasion Techniques

The evaluation of the IDS property resistance to evasion techniques involves the execution of attacks using techniques such that there is a strong possibility that a tested IDS does not detect the attacks. The decision about what evasion techniques should be used in a given study is based on knowledge about the IDS's decision-making process for labeling an activity as benign or malicious. For instance, one may evade an IDS that matches string content of network packets to signatures by modifying the content of malicious packets in a way such that the IDS cannot match them to signatures, a technique known as string obfuscation.

As workloads for evaluating resistance to evasion techniques, one may use pure malicious or mixed workloads. Pure malicious workloads are used to determine which evasion techniques can be used for successfully evading an IDS, not taking into account benign activity as a factor. Mixed workloads are used in scenarios where

Table 3.10: Resistance to evasion techniques of Snort [✓ - detected / x - not detected].

Evasion technique	Targeted vulnerability (CVE ID)	
	CVE 2010-1870	CVE 2013-2115
HTTP::uri_use_backslashes	✓	✓
HTTP::uri_fake_end	✓	✓
HTTP::pad_get_params	✓	x
HTTP::uri_fake_params_start	✓	✓
HTTP::uri_encode_mode (u-random; hex-random)	✓	x
HTTP::pad_method_uri_count	✓	✓
HTTP::method_random_valid	✓	x
HTTP::header_folding	✓	✓
HTTP::uri_full_url	✓	✓
HTTP::pad_post_params	✓	x
HTTP::uri_dir_fake_relative	✓	✓
HTTP::pad_uri_version_type (apache; tab)	✓	✓
HTTP::uri_dir_self_reference	✓	✓
HTTP::method_random_case	✓	✓

benign activities are important for evading an IDS. For instance, a network-based IDS designed to detect multi-step attacks (i.e., attacks that consist of several sequential attacks) may be constrained in the number of network packets that it can buffer for the purpose of attack tracking. Thus, one may evade the IDS by delaying the execution of the sequential attacks and generating benign network traffic between the executions.

The metrics used for quantifying resistance to evasion techniques are basic metrics that do not contain measures of false alerts (e.g., true positive rate) since the goal is to measure the accuracy of an IDS in detecting only evasive attacks.

Case study #5: Following up on the results of Case study #4 (see Table 3.9), we now investigate whether Snort is still able to detect the attacks targeting the vulnerabilities CVE 2010-1870 and CVE 2013-2115/CVE 2013-1966 when evasion techniques are applied. We used Metasploit to apply IDS evasive techniques to the considered attacks since it enables the convenient execution of a wide range of IDS evasive attacks. Given that Snort matches string content of network packets to signatures [Roe99], we used the string obfuscation evasion techniques provided by Metasploit. The applied IDS evasion techniques modify HTTP request strings stored in malicious network packets. In Table 3.10, we list the evasion techniques that we applied to each of the executed attacks (for details on the techniques see [Fos07]).

In Table 3.10, we present the detection score of Snort. It can be observed that Snort detected most of the executed evasive attacks and failed to detect the attack targeting the vulnerability CVE 2013-2115/CVE 2013-1966 when the evasive techniques HTTP::pad_get_params, HTTP::uri_encode_mode, HTTP::method_random_valid, and HTTP::pad_post_params were applied. Snort detected 24 out of 28 attack executions,

thus exhibiting a true positive rate of 0.85.

Attack Detection Accuracy

By evaluation of the attack detection accuracy of an IDS, we mean evaluation of the accuracy of the IDS in detecting attacks mixed with benign activities (see Table 3.7). Attack detection accuracy is quantified using security-related metrics that include measures of false alerts. This is important since an IDS under test might mislabel some benign activities as malicious. We demonstrated quantification of attack detection accuracy in Section 3.3.2.

Case study #6: We evaluate the attack detection accuracy of Snort 2.9.22 using a database of signatures dated 11th July 2013. We used the DARPA datasets as mixed workloads (see Section 3.2.5); we replayed a trace file from the 1998 DARPA datasets with tcpreplay [Tcpb].[1,2] In order to calculate values of security-related metrics that contain measures of false alerts, we used the "ground truth" files provided by the Lincoln Laboratory at Massachusetts Institute of Technology (MIT).[3] These files contain information useful for uniquely identifying each attack recorded in the trace file we replayed, such as time of execution. We compared the "ground truth" information with the alerts produced by Snort in order to calculate the number of detected and missed attacks as well as the number of false alerts. This is required for calculating values of security-related metrics.

With its default configuration enabled, Snort detected almost all attacks. However, Snort also issued false alerts — Snort's rule with ID 1417 led to mislabeling many benign Simple Network Management Protocol (SNMP) packets as malicious. Therefore, we examined the influence of the configuration parameter `threshold` on the attack detection accuracy of Snort. The parameter `threshold` is used for reducing the number of false alerts by suppressing signatures that often mislabel benign activities as malicious. A signature may be supressed such that it is configured to not generate an alert for a specific number of times (specified with the keyword `count`) during a given time interval (specified with the keyword `seconds`).

The measurement of the attack detection accuracy of an IDS for different configurations of the IDS enables the identification of an optimal operating point (see Section 3.3). We measured the attack detection accuracy of Snort for 5 different configurations where the signature with ID 1417 was suppressed by setting the value of `count` to 2, 3, 4, 5, and 6, while `seconds` was set to 120. We also measured the attack detection accuracy of Snort when its default configuration was used, according to which the signature with ID 1417 is not suppressed.

In Table 3.11, we present values of basic security-related metrics (see Section 3.3.1). One can observe that the values of α and $1-\beta$ decrease as the value of `count` increases

[1]The trace file we replayed is available at `https://www.ll.mit.edu/ideval/data/1998/testing/week1/monday/tcpdump.gz`

[2]We used tcpreplay in all IDS evaluation studies presented in this thesis where we replayed traces.

[3]We discussed the importance of "ground truth" information in Section 3.2.5. The "ground truth" files that we used are available at [dar].

— increasing the value of `count` leads to decreasing the number of generated false alerts, which is manifested by the decreasing values of α. However, increasing the value of `count` also leads to worsening of the true positive rate $1 - \beta$. This is a typical trade-off situation between the true and the false positive rate of an IDS. Next, we calculate values of composite security-related metrics.

Table 3.11: Attack detection accuracy of Snort — basic metrics [`seconds=120`].

Configuration	Metrics			
	α	$1 - \beta$	PPV	NPV
`count=6`	0.0008	0.333	0.9788	0.9310
`count=5`	0.0011	0.416	0.9768	0.9390
`count=4`	0.0013	0.5	0.9771	0.9473
`count=3`	0.0017	0.624	0.9761	0.9598
`count=2`	0.0024	0.833	0.9747	0.9817
Default configuration	0.0026	0.958	0.9762	0.9953

In Figure 3.9a, we depict an ROC curve providing an overview of the trade-off mentioned above. In addition, in Figure 3.9a, we annotate the depicted operating points with the associated estimated costs C_{exp}. The values of the estimated costs are values of the expected cost metric (see Section 3.3.2). To calculate values of the expected cost metric, we assumed a cost ratio C of 10. The base rate B is 0.10. Once we calculated the cost associated with each operating point, we were able to identify the optimal operating point (i.e., the operating point that has the lowest C_{exp} associated with it) — (0.0026, 0.958). Based on our findings, we conclude that Snort operates optimally in terms of cost when configured with its default settings.

In Figure 3.9b, we depict the values of the intrusion detection capability metric C_{ID} (see Section 3.3.2) for the considered operating points. The C_{ID} curve depicted in Figure 3.9b enables the identification of the optimal operating point of Snort in terms of intrusion detection capability (i.e., the point that marks the highest C_{ID}) — (0.0026, 0.958), which marks a C_{ID} of 0.883. We conclude that Snort operates optimally in terms of intrusion detection capability when configured with its default settings.

3.4.2 Resource Consumption-related Properties

The resource consumption-related properties are evaluated using pure benign workloads that: *(i)* are considered as regular for the environment where the IDS under test operates, or *(ii)* exhibit extreme behavior in terms of their intensity. The first are used for evaluating the resource consumption of an IDS under regular operating conditions, whereas the latter for evaluating the resource consumption of an IDS that processes high-rate workloads. The metrics used for quantifying IDS resource consumption are performance-related metrics that quantify resource utilization (e.g., CPU utilization).

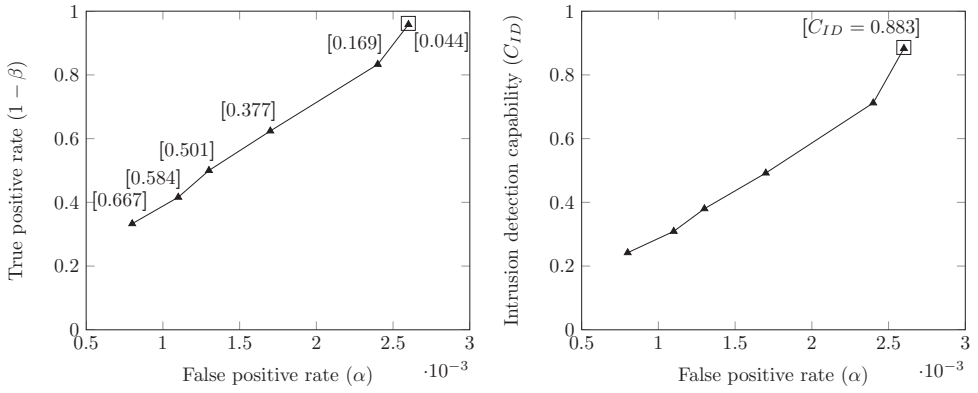


Figure 3.9: Attack detection accuracy of Snort — composite metrics: (a) ROC curve and estimated costs, and (b) C_{ID} curve [□ marks an optimal operating point].

There are mainly two approaches for evaluating IDS resource consumption: black-box and white-box testing. Black-box testing assumes the measuring of the resource consumption of an IDS as resource consumption of the IDS process that is active in the system where the IDS is deployed. This approach is commonly adopted in IDS evaluation experiments, however, it does not provide insight into the resource demands of the individual IDS components. Such insight is normally important for optimizing the IDS configuration. White-box IDS testing usually assumes the use of an IDS model that decomposes the IDS into its components and estimates their individual resource consumption. Dreger et al. [DFPS08] construct an IDS model shown to estimate CPU and memory consumption of an IDS with a relative error of 3.5%. An alternative to modeling is code instrumentation. However, this approach is unfeasible in case the tested IDS is not open-source.

Case study #7: We demonstrate the common black-box IDS resource consumption testing. We measure the CPU consumption of Snort 2.9.22 in two scenarios where we configured Snort to use the pattern matcher ac and ac-split, respectively. The use of a pattern matcher helps to speed up the process of evaluating network packets against Snort signatures by reducing the number of signatures against which each packet is evaluated. When a packet is intercepted by Snort, a pattern matcher evaluates the content of the packet against a set of patterns used to group multiple signatures. If a match is found, the packet is then evaluated only against the signatures that belong to the respective group of rules.

Although ac-split is known to consume less memory resources than ac (see [SM]), it may increase the CPU consumption of Snort. The goal of this study is to determine whether the use of ac-split over ac is advisable by taking CPU consumption into account. To this end, we deployed Snort in a host with a dual-core CPU, each core operating

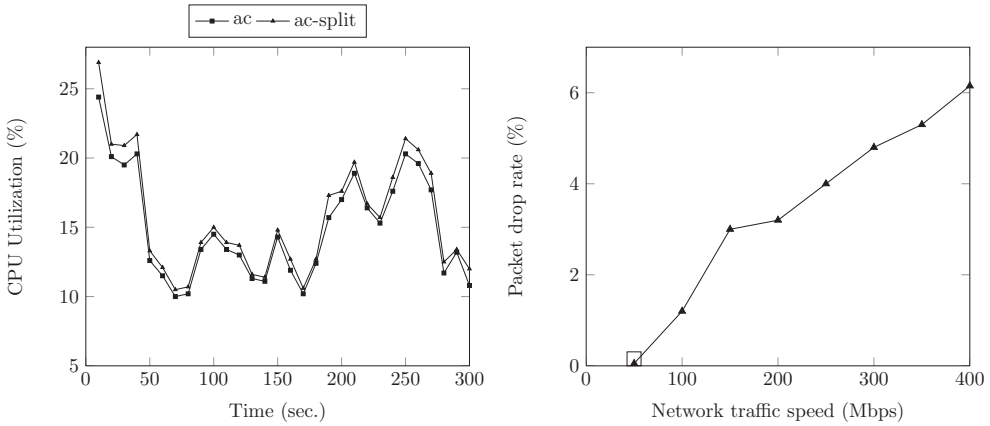


Figure 3.10: (a) CPU consumption of Snort, (b) packet drop rate of Snort [□ marks the data point whose x value is the network traffic speed that corresponds to the maximum workload processing rate of Snort such that there are no discarded workloads].

at 2 GHz, 3 GB of memory, and a Debian OS. To generate pure benign workloads, we replayed a trace file from the LBNL trace repository (see Table 3.4) at the speed of 6 Mbps.[4]

In Figure 3.10a, we depict the CPU utilization of Snort, which we measured with the tool top [tLmp]. This tool enables the measurement of the CPU utilization of any active system process and thus, it is an example of a typical tool used for black-box IDS resource consumption testing. We used top to sample the CPU utilization of the Snort process at every 10 seconds for 5 minutes. We repeated the measurements 30 times and we averaged the results. In Figure 3.10a, one may observe that in the considered scenario, the use of the ac-split pattern matcher does not cause significant increase in the CPU consumption of Snort when compared to the use of ac.

3.4.3 Workload Processing Capacity

The IDS workload processing capacity is evaluated using pure benign workloads that exhibit extreme behavior in terms of intensity. The goal is to observe the rate of arrival of workloads to an IDS under test for processing in relation to the amount of workloads that the IDS discards (see Table 3.7). The identification of a maximum workload processing rate of an IDS such that there are no discarded workloads may also be considered. Similar to evaluating resource consumption, an IDS capacity can be evaluated using a black-box or a white-box testing approach.

[4]The trace file we replayed is available at `ftp://ftp.bro-ids.org/enterprise-traces/hdr-traces05/lbl-internal.20041004-1313.port003.dump.anon`.

While black-box capacity testing does not pose significant challenges, white-box testing is more challenging given that multiple evaluation tests that target specific IDS components involved in processing workloads need to be defined and performed. This requires in-depth knowledge on the design of the IDS under test. White-box testing enables the identification of the particular component of an IDS that is a performance bottleneck. Hall et al. [HW02] propose an approach for white-box IDS capacity testing. They define a methodology consisting of individual tests for measuring the capacity of the components of workload processing mechanisms of network-based IDSs (i.e., the packet flow and capture, the state tracking, and the alert reporting component).

Case study #8: We demonstrate the common black-box IDS capacity testing. We measure the rate of dropped packets by Snort 2.9.22 when it monitors network traffic at speeds in the range of 50 Mbps to 400 Mbps, in steps of 50 Mbps. We deployed Snort in a host with a dual-core CPU, each core operating at the speed of 2 GHz, 3 GB of memory, and a Debian OS. We generated network traffic by replaying a trace file from the LBNL trace repository.[4] We used network workloads in trace form since the use of traces enables the straightforward generation of network traffic at customized speeds in a repeatable manner (see Section 3.2).

To measure the packet drop rate of Snort, we first started the Snort process, then replayed the network trace file, and finally, we stopped the Snort process. When a Snort process is stopped, it displays a set of statistics measured with Snort's built-in performance profiling tools, which includes the packet drop rate averaged over the lifetime of the process. We repeated the measurements 30 times and we averaged the results. In Figure 3.10b, we depict the packet drop rate of Snort in relation to the speed of the monitored network traffic. Starting at network traffic speed of 50 Mbps, the packet drop rate increases almost linearly as the network traffic speed increases. Note that the drop rate of Snort is 0 when it monitors network traffic at the speed of, and less than (not depicted in Figure 3.10b), 50 Mbps; that is, 50 Mbps is the maximum workload processing rate of Snort such that there are no dropped packets.

3.4.4 Performance Overhead

The IDS performance overhead is evaluated using pure benign workloads in executable form that do not exhibit extreme behavior in terms of intensity, but are extreme in terms of the exercised set of hardware resources. Depending on the type of workloads that the IDS under test monitors (e.g., network packets, file I/O operations), an overhead evaluation experiment may consist of five independent experiments, each with a workload that is CPU-intensive, memory-intensive, file I/O-intensive, network-intensive, or mixed. We provided an overview of such tasks in Section 3.2.1 and Section 3.2.2.

The execution of the tasks mentioned above is performed twice, once with the IDS under test being inactive, and once with it being active. The differences between the measured task execution speeds reveal the performance overhead imposed by the IDS.

Case study #9: We evaluate the performance overhead of the host-based IDS Open Source Security (OSSEC) 2.8.1. OSSEC performs file integrity monitoring at real-time

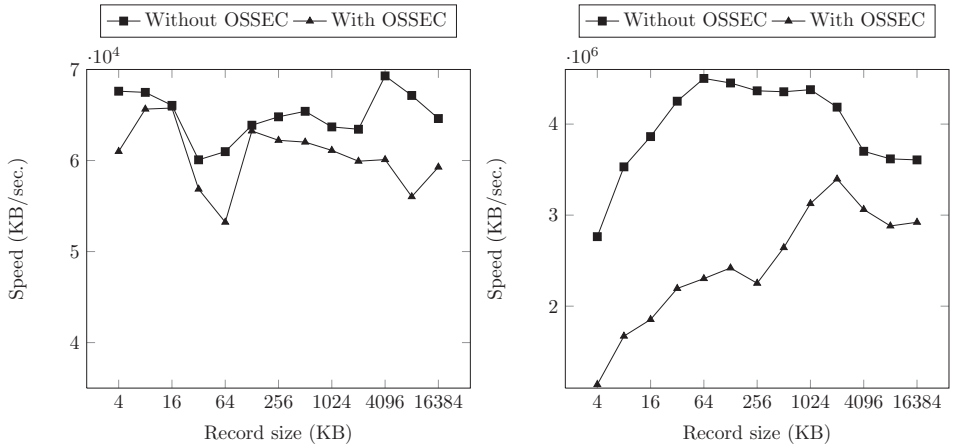


Figure 3.11: Performance overhead imposed by OSSEC on file (a) write, and (b) read operations.

by intercepting and analyzing file I/O operations (e.g., writing to a file). We measure the overhead imposed by OSSEC on file operations reading or writing data of various record sizes. We deployed OSSEC in a host with an ext3 filesystem. Given that OSSEC monitors file I/O operations, we generated pure benign workloads that are file I/O-intensive. We used the workload driver iozone [IFB] (see Section 3.2.1) to generate workloads consisting of file operations that write and read data of record sizes in the range of 4KB to 16MB. For the purpose of this study, we use iozone because it enables the fine-granular customization of file I/O-intensive workloads.

In Figure 3.11a and Figure 3.11b, we depict the execution speeds of the generated file write and read operations. We measured the execution speeds twice, once with OSSEC being inactive, and once with it being active. We repeated the measurements 30 times and we averaged the results. The difference in the speeds shown in Figure 3.11a and Figure 3.11b reveal the overhead imposed by OSSEC. For instance, the execution speed of the operation reading data of a record size 4 KB is 2763641 KB/sec. when OSSEC is not running and 1138420 KB/sec. when OSSEC is running.

3.5 Summary: Open Challenges and IDS Evaluation Guidelines

In this chapter, we systematized existing knowledge on IDS evaluation by defining an IDS evaluation design space that puts existing work into a common context. The IDS evaluation design space that we presented is structured into three parts, that is, workload, metrics, and measurement methodology. For each part of the design space,

we compared multiple approaches and methods that IDS evaluation practitioners can employ. Throughout our discussions on workloads, we identified, and provided links to, commonly used tools, including, for example, workload drivers and trace capturing and replay tools, exploit repositories, and trace repositories. Throughout our discussions on measurement methodologies, we demonstrated how different IDS evaluation approaches are applied in practice. We covered approaches for evaluating the IDS properties attack coverage, resistance to evasion techniques, attack detection accuracy, resource consumption, performance overhead, and workload processing capacity.

3.5.1 Open Challenges: Evaluating Hypervisor-based IDSs

In this section, we focus on the analysis of challenges that apply to evaluating IDSs specifically designed for deployment and operation in virtualized environments (i.e., hypervisor-based IDSs, see Section 1.1). The latter are becoming increasingly ubiquitous with the growing proliferation of virtualized data centers and cloud environments. In particular, intrusion detection in cloud environments has been recently receiving increasing attention, given that security concerns are still one of the greatest showstoppers for the wide adoption of cloud computing [GMV+10]. The evaluation of hypervisor-based IDSs is an emerging research area that is yet to be explored. Therefore, by providing an overview of the open challenges in this novel research area, we aim to contribute towards the establishment of a future research agenda.

In a virtualized environment, a hypervisor running on each physical machine is used to host multiple virtual machines (VMs). Hypervisor-based IDSs are deployed in the virtualization layer, usually with components inside the hypervisor and in a designed VM, which has several benefits. For instance, such IDSs can monitor the network and host activities of all VMs at the same time [JXZ+11], [LDP11], [Kon11]. Further, they are transparent to malicious users of the VMs. Finally, they are isolated from such users since they do not operate in the guest VMs, but leverage functionalities of the underlying hypervisor (see Section 1.1).

The hypervisor-based IDSs described above do not deploy monitoring agents inside the VMs, in order to achieve full transparency and isolation from attackers. However, the virtualization layer provides access only to low-level hardware information. Thus, hypervisor-based IDSs are unable to directly access rich OS-level information about the monitored VMs needed as input to intrusion detection logic (e.g., executed system calls, data files). This is known as the *semantic gap*. In order to alleviate this issue, hypervisor-based IDSs use virtual machine introspection (VMI) that provides them with access to OS-level information. An alternative approach towards closing the semantic gap is deployment of monitoring agents inside the VMs at the cost of decreased isolation and transparency. In this section, we focus on hypervisor-based IDSs with components inside the hypervisor, the latter being the most representative for virtualized environments.

We note that most existing evaluation approaches from over the last several decades can be applied to hypervisor-based IDSs as well. For instance, workload generation

methods such as benign workload drivers (Section 3.2.1), honeypots to capture malicious activities (Section 3.2.6), the DARPA datasets (Section 3.2.5), or the Metasploit framework (Section 3.2.3), may still be effectively used in the context of hypervisor-based IDSs. Further, the IDS properties discussed in Section 3.4, such as performance overhead, resource consumption, attack detection accuracy, and the associated trade-offs, are crucial for accurate evaluation of hypervisor-based IDSs. For instance, Lombardi et al. [LDP11] use benign workload drivers to measure the performance overhead of a hypervisor-based IDS, and Hai et al. [JXZ+09] use the DARPA dataset as a malicious workload to measure the attack detection accuracy.

Leveraging lessons learned from existing work in IDS evaluation, we derive a set of specific requirements and challenges that must be addressed in order to improve the current IDS evaluation practices for hypervisor-based IDSs. In the discussions that follow, we focus on the challenges and requirements that stem from the following aspects of hypervisor-based IDSs: *(i)* the use of VMI as an intrusion monitoring technique, and *(ii)* the virtualization-specific feature of supporting the operation of multiple VMs on top of a single shared hypervisor. We analyze the respective open research challenges related to the generation of workload traces that contain VMI information and the characterization of benign workloads from multiple VMs.

Generation of Workload Traces

Hypervisor-based IDSs use VMI in order to monitor VMs. VMI is a monitoring approach that was first introduced by Garfinkel et al. [GR03] in 2003. By obtaining VMI information about various system components (e.g., CPU register values, network interface memory content), a typical hypervisor-based IDS can monitor VMs from a trusted designated VM. Two VMI tools and libraries that have been used by hypervisor-based IDSs are AntFarm [JADAD06] and XenAccess [xend].

In this section, we investigate the challenge of recording and replaying VMI information for the purpose of generating IDS evaluation workloads in the form of traces. Nance et al. [NBH08] in 2008 briefly discussed the challenge of logging and replaying VMI information. They state that a logging VMI may be used to record relevant system events that enable an in-depth security analysis. They also state that a VM should record enough information to reconstruct a given relevant event. Finally, they close the discussion with the conclusion that the nature and the amount of recorded information varies significantly with respect to the goals of the replay. In this section, we look at the challenge of recording and replaying VMI information focusing on replaying security-relevant events for the purpose of evaluating hypervisor-based IDSs in particular.

Different hypervisor-based IDSs have different monitoring behavior in terms of which specific activities they monitor, as well as when they monitor such activities. For instance, some obtain VMI information about CPU registers, for example, the Extended Accumulator Register (EAX) register, in order to track the execution of

system calls,[5] while others use process structures stored in main memory to monitor the active processes of VMs. Finally, a hypervisor-based IDS might inspect additional VMI information than the one it normally inspects if it suspects the existence of an on-going attack.

As a result of the above observations, it is a significant challenge to capture trace files that contain the exact VMI information required at each point in time by a hypervisor-based IDS. The recording rate of VMI information would typically be constrained by many properties of the recording platform, such as the underlying I/O bandwidth of its file system. Thus, in case of highly intensive host and/or network activities, extensive logging of excessive VMI data over a longer period of time might impair the recording rate and, consequently, the overall quality of the generated trace files.

In addition to the above challenge, existing hypervisor-based IDSs differ significantly in terms of their architectural designs and the type of monitoring data that they require. Many VMI techniques leveraged by hypervisor-based IDSs require hardware-level information as observable by a hypervisor (e.g., CPU register values, main memory content), as well as high-level domain-specific knowledge about the VMs (e.g., file system structure, kernel data structures, and similar). As an example, the hypervisor-based IDS designed and developed by Dehnert [Deh12], which uses VMWare's VProbes technology [vpr] for VMI, requires knowledge on OS-specific kernel data structures (e.g., the Linux kernel structure task_struct) and also on specific memory offsets, which are different for different kernel versions. Hardware-level information and high-level domain-specific knowledge are often needed by a hypervisor-based IDS in order to inspect even a single host or network event.

To illustrate the complexity of the challenges discussed above, we analyze trace recording procedure in the context of Wizard [SSG08], a representative hypervisor-based IDS. For the sake of simplicity, we assume that the goal is to record a mixed workload trace which contains a single attack in a given time interval. Wizard detects anomalies in the execution of VM system calls for the purpose of detecting kernel attacks that alter the kernel behavior (see Section 2.1.1). Note that during execution, VM system calls may invoke VM calls which can be intercepted in the hypervisor (i.e., by the Wizard's hypervisor component). VM calls are requests issued by a VM requesting a specific action or information from the underlying hypervisor. In order to apply intrusion detection logic, Wizard maps sequences of VM calls to the respective invoking system calls for which it requires OS-specific knowledge on the execution of system calls (e.g., used registers), which we refer to as *OS knowledge*. Each time Wizard intercepts a VM call, it reads the value stored in the Control Register (CR) 3 in order to map the VM call to a specific VM process. The CR3 register on Intel x86 platforms stores the page table base address, which is unique for each process enabling the identification of the process that executes a given VM call.

As an example, we look at a specific scenario when a kernel-level keylogger (Section 2.1.1) intercepts and modifies the regular execution of the *read* system call in order to capture and store system input. To record a trace capturing the altered execution of

[5] The EAX register on Intel i386 platforms stores a number identifying the system call that is being executed, which enables the identification of the system call.

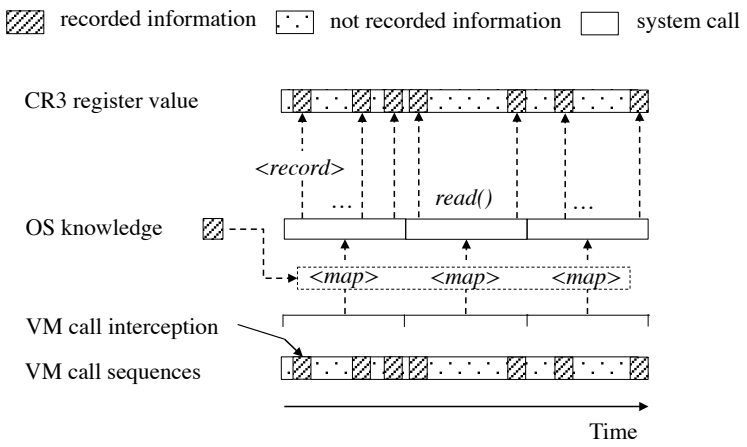


Figure 3.12: Trace recording procedure for Wizard [SSG08].

read, a recording mechanism needs to record the following data: *(i)* the stream of VM call sequences, *(ii)* the required OS knowledge, and *(iii)* the value of the CR3 register when a VM call is intercepted. Note that in order to timely capture the CR3 register value, the recording mechanism should be aware of when exactly a VM call is executed. We depict this trace recording procedure in Figure 3.12. The execution times of the VM calls that we depict in Figure 3.12 are chosen randomly.

Given the complexity of recording procedures, it is expected that replay procedures would also be challenging. Thus, a systematic classification structuring the various types of VMI information used by hypervisor-based IDSs would be an important contribution. The latter can be used as a basis to design configurable recording and replay mechanisms serving as adapters that abstract the underlying architectural details of different hypervisor-based IDSs.

Benign Workload Characterization

A typical hypervisor-based IDS monitors multiple VMs at the same time. Each VM runs a specific OS that uses a particular natively supported filesystem and stores specific OS data in the main memory of the VM (e.g., system call codes, process structures). Many hypervisor-based IDSs monitor both the filesystems and the OS data of VMs. Further, the applications and/or services deployed in the OSes of VMs typically generate workloads that are either executed locally (i.e., host workloads), or use a network interface (i.e., network workloads). Many hypervisor-based IDSs have the ability to monitor both host and network workloads that originate from a given VM. Particularly, when monitoring a host workload, a hypervisor-based IDS usually monitors the execution of system calls. Similar to the filesystem and the OS data, the system calls are also OS-specific. We refer to the previously mentioned monitored

entities (i.e., VMs and respective workloads, filesystems, and OS data) as *monitoring landscape* of the hypervisor-based IDS.

The workloads induced by the VMs are usually of various types (e.g., streaming, data processing, and scientific computations). For instance, the previously mentioned workload types have been identified as among the most representative workload types in cloud environments [SB10]. Each workload type normally has a specific set of characteristics relevant for intrusion detection (e.g., burstiness, intensity of execution of specific system calls, and similar), which we refer to as *workload profile*.

In modern data centers, the number of VMs co-located on a single hypervisor can vary frequently due to the possibility to migrate VMs at run-time; that is, a new VM may arrive or an existing one may be removed from a hypervisor at any time. We refer to these times as arrival time and departure time of a VM, respectively. VM migration might be triggered for different purposes. For instance, a VM user might explictly request VM migration. Further, the virtualization platform might employ dynamic VM placement policies that optimize resource efficiency during operation by automatically migrating VMs between servers in response to changes in their workload profiles. We refer to the dynamic deployment behavior of a VM as its *deployment profile*.

Since VM migration impacts the number of VMs monitored by a hypervisor-based IDS, it ultimately introduces dynamic changes in the overall monitoring landscape of a hypervisor-based IDS over time. This includes changes in the amount, types, and characteristics of the monitored workloads. As an illustration, in Figure 3.13, we depict an overview of the monitoring landscape of a hypervisor-based IDS.

Given the dynamicity of a typical monitoring landscape, we argue that it is challenging to define a benign workload that can be considered as representative for “normal" workload conditions. Among many other uses, as mentioned in Section 2.1.2, the notion of normal workload is important for training anomaly-based IDSs, which is required as initial step when evaluating anomaly-based IDSs.

In traditional (i.e., not virtualized) environments, labeling a given workload as “normal” with respect to a specific usage profile is usually not as a challenging, since the infrastructure that an IDS monitors is relatively static over time. For instance, host-based IDSs monitor a single OS normally used for a single specific purpose (e.g., accounting), and network-based IDSs monitor network traffic of specific network services whose behavior can be characterized as “normal”. Thus, many IDS evaluation datasets (e.g., the DARPA datasets) include training workloads that are labeled as “normal” with respect to a specific usage profile. This is not the case in virtualized environments where a hypervisor-based IDS monitors multiple VMs that may depart at any time, and new VMs, each with specific benign workload characteristics, may arrive. Note that the monitoring mechanisms of a typical hypervisor-based IDS are deployed inside the hypervisor, and thus, it is not clear what should be considered as a “normal” workload from the perspective of a hypervisor-based IDS.

An intuitive approach towards overcoming the above issue would be to train an anomaly-based hypervisor-based IDS with baseline benign activities originating from a fixed set of VMs, so that a subsequent evaluation of the IDS detection accuracy for

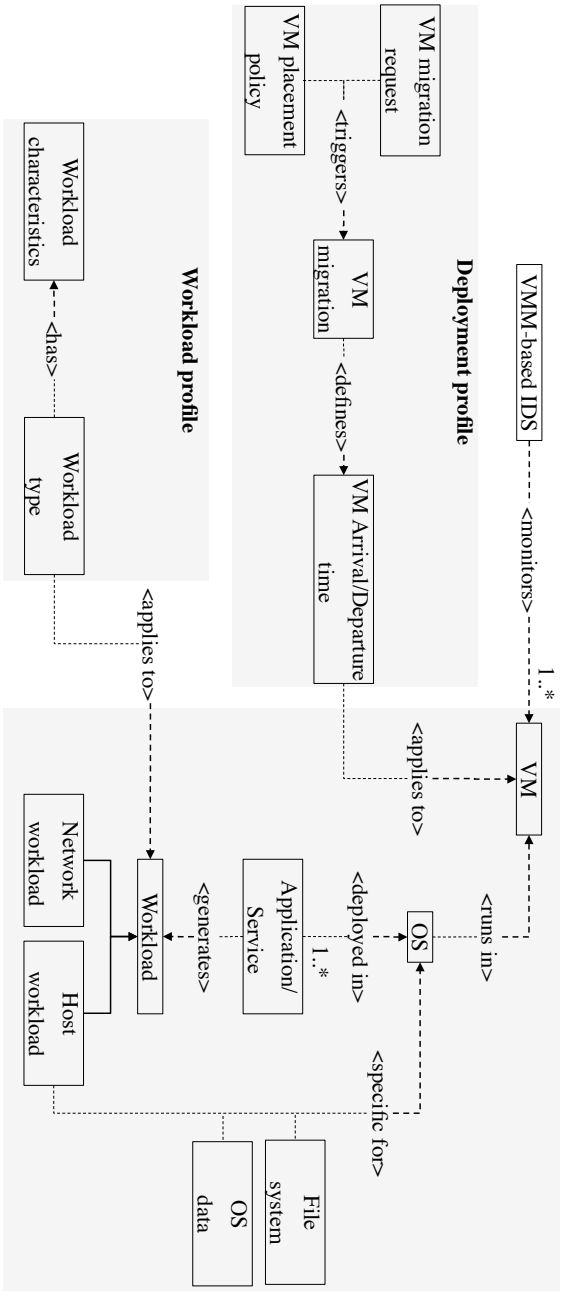


Figure 3.13: Monitoring landscape of a hypervisor-based IDS.

attacks against these VMs may be carried out. However, the obtained results would be of limited representativeness and value, since one cannot assume that in a real-world setting, a hypervisor-based IDS would monitor a fixed set of VMs over long periods of time. To the contrary, one can expect that the behavior of monitored benign workload would change significantly over time due to arriving and departing VMs. This is especially true in modern virtualized data centers where VM migration occurs frequently as part of load balancing or quality-of-service policies.

A sound approach for training an anomaly-based hypervisor-based IDS would be to train it with respect to a given baseline monitoring landscape that defines arrival and departure times for VMs with various workload characteristics. However, the definition of such a monitoring landscape for a given virtualized environment is challenging; that is, one cannot easily determine realistic VM arrival/departure times and respective workload characteristics of the VMs. A typical monitoring landscape depends on many degrees of freedom, such as the configured VM migration policies, the types and usage profiles of the VMs that are deployed in the virtualized environment, and so on. Note that some virtualized environments, such as the Amazon Elastic Compute Cloud (EC2) [AMIA], support the creation of different types of pre-configured VMs that provide pre-packaged software stacks for a specific type of environment. For instance, the Amazon Machine Images (AMIs) enable the creation of VMs with a deployed OS and a specific set of applications in order to accommodate a particular user activity (e.g., software development).

The dynamicity of the monitoring landscape also introduces unique technical challenges. For instance, assuming that one is supposed to create a training dataset for a hypervisor-based IDS that monitors system calls, such as the IDSs proposed by Lombardi et al. [LDP11] and Srivastava et al. [SSG08], multiple datasets consisting of systems calls of all OSes that may be hosted on the respective hypervisor would be required. Lack of such datasets would significantly reduce the usefulness of the IDS. This situation would not occur when it comes to host intrusion detection in traditional non-virtualized environments.

A possible solution to the above mentioned problems would be the development of a modeling approach in order to gain a deeper understanding of the changes that occur in the monitoring landscape over time as well as of the workload characteristics monitored by hypervisor-based IDSs. Such a modeling approach would be highly benefitial, for example, by allowing to analyze the workload characteristics for different scenarios and identify baseline normal workload profiles.

3.5.2 IDS Evaluation Guidelines

We now provide guidelines for planning IDS evaluation studies based on what we observed when analyzing relevant work in the field. We structure our discussion by focusing on three key and inter-related points in the planning of every evaluation study - *goals* of a study, existing *approaches* to realize the set goals (i.e., approaches for generating workloads, and for measuring IDS performance — metrics), and *requirements* that need to be met.

Goals: Under goals of an IDS evaluation study, we understand the IDS properties that one aims to evaluate. The selection of IDS properties for evaluation is normally done by considering the design objectives and the target deployment environment of the IDS under test. In Table 3.12, we present the most commonly considered IDS properties in evaluation studies for various IDS types. We also provide a summarizing

Table 3.12: Overview of common trends, recommendations, and key best practices.

IDS Property	
Attack detection accuracy Attack coverage	These properties are evaluated for IDSs of all types • The dated DARPA and KDD'99 Cup datasets represent at this time standard workloads for comparing novel anomaly-based IDSs with their past counterparts • For the sake of representativeness, evaluate an IDS using not the DARPA or the KDD'99 Cup dataset, but workloads that contain current attacks • Attack detection rates of current IDSs vary greatly, that is, between 8% and 97%, measures which depend on the configurations of the tested IDSs and the applied evaluation methodologies
Attack detection and reporting speed	This property is normally evaluated for distributed IDSs — it is best evaluated by measuring the time needed for the IDS to converge to a state in which all its nodes, or the designated nodes, are notified of an on-going attack • Attack detection delays up to 3 seconds are considered acceptable
Resistance to evasion techniques	This property is often not evaluated since it is considered of limited practical importance • Consider evaluating this property since a single successful IDS evasion attack poses the danger of a high-impact intrusion • Metasploit is deemed the optimal tool for executing IDS evasive attacks, which is required for evaluating this property • Many current IDSs are vulnerable to temporally crafted attacks
Resource consumption-related	These properties are typically evaluated for IDSs deployed in resource constrained environments • Network consumption in particular is often evaluated for distributed IDSs • The resource consumption of a distributed IDS operating in wireless ad-hoc networks is typically evaluated in order to measure the power consumption of its nodes — this is best performed by using a model that estimates power consumption based on resource consumption measurements
Performance overhead	This property is normally evaluated for host-based IDSs • Performance overhead is evaluated by executing tasks twice, once with the tested IDS being inactive, and once with it being active • This property is normally evaluated using workloads in executable form generated by workload drivers — workload drivers enable the straighforward generation of live customized workloads in a repeatable manner • Overheads under 10%, relative to the execution time of tasks measured when the tested IDS is inactive, are generally considered acceptable
Workload processing capacity	This property is normally evaluated for network-based IDSs that monitor high-rate workloads • This property is best evaluated using traces or workload drivers since they allow for the generation of workloads at user-defined speeds • Evaluate the capacity of an IDS together with its resource consumption — this enables to observe how resource consumption scales as workload intensity increases

overview of common trends, recommendations, and key best practices in evaluating IDS properties for different types of IDSs. Finally, we present observed quantitative values (e.g., generally acceptable performance overheads) and relevant observations that may serve as reference points for designing and evaluating future IDSs. For more details, we refer the reader to Section 3.4. For demonstrations on executing IDS tests in order to evaluate the commonly considered IDS properties, we refer the reader to Section 3.4.1, Section 3.4.2, Section 3.4.3, and Section 3.4.4.

Requirements: Before discussing approaches for evaluating IDSs, we identify and systematize the requirements that have to be met for the different approaches:

— *availability of required resources*: *(i) financial resources* (e.g., the costs of building a testbed may be significant, see Section 3.2.6); *(ii) time resources* (e.g., the manual assembly of an exploit database may be time-consuming, see Section 3.2.3); *(iii) manpower* (e.g., the amount of available human resources is important for labeling traces in a time-efficient manner, see Section 3.2.5).

— *access to confidential data*: organizations are often unwilling to share operational traces because of privacy concerns and legal issues (see Section 3.2.5).

— *availability of knowledge* about: *(i) the architecture and inner working mechanisms of the IDS under test* (see, for example, the discussions on evaluating resistance to evasion techniques in Section 3.4.1); *(ii) the characteristics of the employed workloads* (e.g., information about the attacks used as workloads must be known in order to calculate any security-related metric, see Section 3.3); *(iii) the implications of different behavior exhibited by the tested IDS* (e.g., the cost of missing an attack must be known in order to calculate the expected cost metric, see Section 3.3.2).

We observed that the requirements mentioned above often cannot be fully satisfied given the big investment of resources that typically needs to be made. We observed that sacrifices are often made in: *(i) the scale of the employed workloads*: an example is the typical low number of attack scripts used as workloads (see Section 3.2.3); *(ii) the number of considered IDS properties* (i.e., researchers tend to evaluate only a few IDS properties, see Table 3.8).

We suggest trade-offs made between the quality of evaluations and the invested resources to be clearly stated when reporting results from IDS evaluation studies so that the results can be interpreted in a fair and accurate manner.

Approaches: In Table 3.13, we systematize relevant information that facilitates the process of selecting approaches for generating workloads for evaluating IDSs and for measuring IDS performance (i.e., metrics). In-depth information on what is presented in Table 3.13 can be found in Section 3.2 and Section 3.3, whose sub-sections correspond to the approaches listed in Table 3.13. For each approach, where applicable, we provide: *key information* (i.e., advantages, disadvantages, and/or relevant facts); *key requirements* that need to be satisfied, systematized as above; example common *practices* or commonly used *tools*, *datasets*, or *metrics*; and selected *references* to relevant publications where studies that use the approach can be found.

Table 3.13: Guidelines for planning IDS evaluation studies: Workloads and metrics [✓: advantage, −: disadvantage, ○: neutral].

Workloads
Pure benign/mixed/pure malicious →Executable form
Key information:
✓ Generated workloads closely resemble real workloads
○ Multiple evaluation runs are required to ensure statistical significance of IDS behavior
○ Generated malicious workloads require specific victim environments
- Replicating experiments when using malicious workloads is challenging
(Pure benign)→Workload drivers (Section 3.2.1)
Key information:
✓ Generated workloads can be customized in terms of their temporal and intensity characteristics
- Generated workloads do not resemble real-life workloads as closely as those manually generated
Tools: SPEC CPU2000, iozone, Postmark, httpbench, dkftpbench, ApacheBench, UnixBench (see Table 3.1)
References: [ABYSS10], [PKSZ04], [DKC+02], [GR03], [JXZ+11]
(Pure benign)→Manual generation (Section 3.2.2)
Key information:
✓ Generated workloads are similar to those observed by an IDS during regular system operation
○ Suitable for generation of workload traces capturing realistic workloads executed in a recording testbed
- Does not support workload customization
Key requirements: time resources,(a) manpower(a)
Practices: file encoding and tracing, file conversion, copying of large files, kernel compilation (see Table 3.1)
References: [DKC+02], [SSG08], [LDP11], [ABYSS10]
(Pure malicious)→Exploit database →Manual assembly (Section 3.2.3)
Key information:
✓ Generated workloads are realistic and representative
○ Attack scripts need to be collected and adapted, and a victim environment needs to be setup
- Generated workloads are normally of a limited size due to lack of manpower and/or time resources
- Publicly available attack scripts normally do not have IDS evasive characteristics
Key requirements: time resources, manpower, knowledge about the architecture and inner working mechanisms of the IDS under test(b)
Datasets: 1337day, Exploit database, Packetstorm, SecuriTeam, Securityfocus (see Table 3.2)
References: [PCOM97], [LDP11], [DDWL98], [RRL+12]
(Pure malicious)→Exploit database →Readily available exploit database (Section 3.2.3)
Key information:
✓ No time spent on collection and adaptation of attack scripts
- Some databases have critical limitations (e.g., contain only remote exploits)
Key requirements: knowledge about the architecture and inner working mechanisms of the IDS under test(b)

(a) When large scale workloads are generated (e.g., for recording in a testbed).
(b) In case the IDS property "resistance to evasion techniques" is evaluated.

Continued from previous page

Tools: Metasploit, Nikto, w3af, Nessus

References: [GER08]

(Pure malicious)→Vulnerability and attack injection (Section 3.2.4)

Key information:

✓ Useful when collection of attack scipts is unfeasible

✓ Remotely and locally exploitable codes can be injected

- Not an extensively investigated approach

Key requirements: knowledge about the architecture and inner working mechanisms of the IDS under test

Tools: Vulnerability and Attack Injector Tool (VAIT) [FVM14]

References: [FVM14]

Pure benign/mixed/pure malicious → Trace form

Key information:

✓ A single evaluation run is required to ensure statistical significance of IDS behavior

✓ Replicating evaluation experiments is a straightforward task

- Generated workloads closely resemble real workloads only for a short period of time

○ Generated malicious workloads do not require specific victim environments

→ Trace acquisition → Real-world production traces (Section 3.2.5)

Key information:

✓ Highly realistic and possibly large-scale workloads

- Normally very difficult to obtain due to privacy concerns and legal issues

- Normally anonymized with the risk that intrusion detection relevant data is removed

- Normally not labeled, which makes the construction of "ground truth" challenging

Key requirements: time resources, manpower, access to confidential data

→ Trace acquisition → Publicly available traces (Section 3.2.5)

Key information:

✓Can be obtained without any legal constraints

✓Normally labeled

- Many contain errors (e.g., unrealistic distributions of attacks)

- May be outdated due to limited shelf-life of attacks

○ Claims on generalizability of results from IDS tests based on publicly available traces can often be questioned

Key requirements: knowledge about the characteristics of the employed workloads

Datasets: CAIDA, DEFCON, DARPA/KDD'99, ITA, LBNL, MAWILab (see Table 3.4)

References: [AAA+10], [YD11], [RAR12], [DCW+99]

→Trace generation → Testbed environment (Section 3.2.6)

Key information:

✓ Issues related to trace acquisition not present

- The costs of building a testbed that scales to realistic production environments may be high

○ Methods used for trace generation are critical as they may produce faulty or simplistic workloads

Key requirements: financial resources, time resources,$^{(c)}$ manpower$^{(c)}$

$^{(c)}$ In particular when recorded workloads are generated manually.

Continued from previous page

References: [CLF+99], [SSTG12]

→ Trace generation → Honeypots (Section 3.2.6)

Key information:

✓ Enable the generation of traces that contain representative and possibly zero-day attacks

- The outcome of a trace generation campaign is uncertain since it cannot be planned in advance and controlled

○ The attack labeling feasiblity depends on the level of interaction of used honeypot(s)

Key requirements: time resources,$^{(d)}$ manpower$^{(d)}$

Tools: honeyd, nepenthes, mwcollectd, honeytrap, HoneyC, Monkey-Spider, honeybrid, HoneySpider, Sebek, Argos, CaptureHPC, HoneyClient, HoneyMonkey (see Figure 3.4)

References: [DS11]

Metrics

Security-related → Basic (Section 3.3.1)

Key information:

○ Quantify individual attack properties

○ Need to be analyzed together in order to accurately quantify the attack detection accuracy of an IDS

○ "Ground truth" information is a requirement for calculating the basic security-related metrics

Key requirements: knowledge about the characteristics of the employed workloads

Metrics: true positive rate (1-β), false postive rate (α), false negative rate (β), true negative rate (1-α), positive predictive value (PPV), negative predictive value (NPV) (see Table 3.5)

References: [MHL+03], [RAR12], [SSG08], [RRL+12], [FJGS+00]

Security-related → Composite

Key information:

○ Used for analyzing relationships between the basic security-related metrics

○ Useful for identifying (an) optimal IDS operating point(s) for evaluating a single or comparing multiple IDSs

→ ROC curve (Section 3.3.2)

Key information:

○ The first metric of choice for identifying optimal IDS operating points

○ An open issue is how to determine a proper unit and measurement granularity

- ROC curve analysis may be misleading when used for comparing multiple IDSs

Key requirements: knowledge about the architecture and inner working mechanisms of the IDS under test,$^{(e)}$ knowledge about the characteristics of the employed workloads

References: [RAR12], [DCW+99]

→ Cost-based/Information-theoretic (Section 3.3.2)

Key information:

✓ Enable the accurate and straightforward comparison of multiple IDSs

✓ Express the impact of the rate of occurence of intrusion events (i.e., base rate)

○ (Depend/Do not depend) on subjective measures (e.g., cost) and are (unsuitable/suitable) for objective comparisons of IDSs

$^{(d)}$ Depends on the attack labeling feasibility of generated traces.

$^{(e)}$ In particular, knowledge about employed workload processing mechanisms (e.g., units of analysis) in order to avoid misleading results when comparing IDSs.

Continued from previous page

Key requirements: knowledge about the characteristics of the employed workloads, knowledge about the implications of different behavior exhibited by the IDS under test (applies for cost-based metrics only)

Metrics: cost-based — expected cost (see Table 3.5), relative expected cost [Men12]; information-theoretic — intrusion detection capability (see Table 3.5), false alarm reduction capability [YMLFK13]

References: cost-based metrics — [GU01], [Men12]; information-theoretic metrics — [GFD+06], [YMLFK13]

Chapter 4

An Analysis of Hypercall Handler Vulnerabilities

The aim of this chapter is to shed more light on the hypercall attack surface. Currently, to the best of our knowledge, there is no related work analysing this surface. Among other reasons, we analyse the hypercall attack surface for the purpose of constructing hypercall attack models, which are required for injecting hypercall attacks in a representative manner. The latter enables the accurate evaluation of intrusion detection systems (IDSs) designed to detect hypercall attacks, which is one of the goals of this thesis (see Section 1.3).

In this chapter, we present an in-depth analysis of 35 hypercall vulnerabilities, which we found by searching major vulnerability report databases (e.g., cvedetails [CVEj]) based on relevant keywords. We discuss issues, challenges, and gaps that apply specifically to securing hypercall interfaces and how they differ from security concerns related to system calls. Note that a hypercall is conceptually similar to a system call; that is, hypercalls are analogous to system calls in the OS world (see Section 1.3).

Our analysis is based on information obtained by reverse engineering released patches fixing the considered vulnerabilities due to the lack of publicly available scripts that demonstrate hypercall attacks (see Section 1.3). In summary, in this chapter we present:

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

This chapter is organized as follows: in Section 4.1, we present the set of hypercall vulnerabilities that we analyze and the applied analysis method; in Section 4.2, we systematize and discuss the origins of the analyzed vulnerabilities and attackers' activities for executing hypercall attacks; in Section 4.3, we identify open issues and we propose future research directions; in Section 4.4, we summarize this chapter.

In Chapter A, we provide relevant technical information about the hypercall attacks we demonstrate in this chapter for better understanding these attacks.

The work presented in this chapter has been published in [MPA+13] and [MPA+14].

4.1 Sample Set of Hypercall Vulnerabilities

Table 4.1 lists the hypercall vulnerabilities analyzed for this study. We analyzed 35 hypercall vulnerabilities, which we found by searching major Common Vulnerabilities and Exposures (CVE) report databases (e.g., cvedetails [CVEj]) based on relevant keywords, such as names of operations of hypercalls. In order to ensure representativeness of the considered set of vulnerabilities, we analyzed all publicly disclosed hypercall vulnerabilities that we found as previously described, expecting that the number of those that we did not find is negligibly small.

The majority of the vulnerabilities listed in Table 4.1 are from the Xen hypervisor [BDF+03]. This is because Xen has the most extensive hypercall interface, which enables full paravirtualization of virtual machines (VMs), as opposed to other hypervisors, such as Kernel-based Virtual Machine (KVM) [Kiv07], which enables only partial paravirtualization of VMs. The input/output control (ioctl) calls that the KVM hypervisor supports are functionally and conceptionally very similar to hypercalls of the Xen hypervisor. For the purpose of this study, in addition to its standard hypercall interface, we consider the ioctl interface of the KVM hypervisor, and therefore vulnerabilities in handlers of ioctl calls, as hypercall interface and hypercall vulnerabilities, respectively.

At the time of writing, the total number of publicly disclosed hypercall vulnerabilities is small. Based on what we observed when analyzing the vulnerabilities, we argue that this is mainly because the assessment of hypercall handlers for vulnerabilities is a challenging task due to the complexity of the operations that hypercall handlers perform. In addition, the lack of instructive documentation of the program code of hypercall handlers makes the assessment for vulnerabilities by an analyst, who has not been involved in the design and/or development of the handlers, an especially challenging task. As a result, as one can conclude from the range of releases of hypervisors affected by hypercall vulnerabilities (Table 4.1, column 'release' of 'affected hypervisor'), it took a significant amount of time for some hypercall vulnerabilities to be discovered (e.g., CVE-2012-5514, CVE-2013-4494).

Our approach for analyzing a hypercall vulnerability consisted of the following steps:

(i) analysis of the CVE report describing the vulnerability and of other relevant information sources, for example, security advisories;

(ii) reverse-engineering of the released patch fixing the vulnerability; and

(iii) developing proof-of-concept code for triggering the vulnerability in a testbed environment.

Table 4.1: Analyzed hypercall vulnerabilities.

CVE ID	Hypercall	Affected hypervisor	
		Name	Release
CVE-2008-3687	flask_op	Xen	3.3
CVE-2009-2287	KVM_SET_SREGS	KVM	Linux kernel 2.6.x(<2.6.30)
CVE-2009-3290	kvm_emulate_hypercall	KVM	Linux kernel 2.6.25-rc1–2.6.31
CVE-2009-3638	KVM_GET_SUPPORTED_CPUID	KVM	Linux kernel 2.6.25-rc1–2.6.32-rc4
CVE-2009-4004	KVM_X86_SETUP_MCE	KVM	Linux kernel <2.6.32-rc7
CVE-2010-3698	KVM_RUN	KVM	Linux kernel <2.6.36
CVE-2011-4347	KVM_ASSIGN_PCI_DEVICE	KVM	Linux kernel 3.1.7–3.1.9
CVE-2012-1601	KVM_CREATE_IRQCHIP	KVM	Linux kernel <3.3.6
CVE-2012-3494	set_debugreg	Xen	4.0.x–4.1.x(<4.1.4)
CVE-2012-3495	physdev_op	Xen	4.1.x(<4.1.4)
CVE-2012-3496	memory_op	Xen	3.9.x–4.1.x(<4.1.4)
CVE-2012-3497	tmem_op	Xen	>4.0.x
CVE-2012-3516	grant_table_op	Xen	4.2.0
CVE-2012-4461	KVM_SET_SREGS	KVM	Linux kernel <3.6.9
CVE-2012-4538	hvm_op	Xen	4.0.x–4.1.x(<4.1.4), 4.2.x(<4.2.1)
CVE-2012-4539	grant_table_op	Xen	4.0.x–4.1.x(<4.1.4)
CVE-2012-5510	grant_table_op	Xen	4.x(<4.1.4)
CVE-2012-5513	memory_op	Xen	<4.1.4
CVE-2012-5514	memory_op	Xen	3.4.x–4.1.x(<4.1.4)
CVE-2012-5515	memory_op	Xen	3.4.x–4.1.x(<4.1.4)
CVE-2012-5525	mmuext_op	Xen	4.2.0
CVE-2013-0151	hvm_op	Xen	4.2.x(<4.2.2)
CVE-2013-0154	mmuext_op	Xen	4.2.x(<4.2.2)
CVE-2013-1964	grant_table_op	Xen	4.0.x–4.1.x(<4.1.5)
CVE-2013-3898	unknown	Hyper-V	Windows 8 and Server 2012
CVE-2013-4494	grant_table_op	Xen	>3.2.x
CVE-2013-4553	domctl	Xen	>3.4.x
CVE-2013-4554	hvm_do_hypercall	Xen	3.0.x–4.3.2
CVE-2013-5634	KVM_GET_REG_LIST	KVM	Linux kernel 3.9.x
CVE-2014-1891	flask_op	Xen	3.x.x–4.2.x(<4.2.4), 4.3.x(<4.3.2)
CVE-2014-1892	flask_op	Xen	3.3.x–4.1.x
CVE-2014-1893	flask_op	Xen	3.2.x–4.1.x
CVE-2014-1894	flask_op	Xen	3.2.x
CVE-2014-1895	flask_op	Xen	4.2.x(<4.2.4), 4.3.x(<4.3.2)
CVE-2014-3124	hvm_op	Xen	>4.1.x

The execution of proof-of-concept code enabled us to closely observe all stages of an attack life cycle — pre-attack activities, followed by execution of a hypercall, or a

series of hypercalls, triggering a given vulnerability, and finally, the post-attack state of the targeted hypervisor.

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

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

4.2 Analysis of the Hypercall Attack Surface

We analyze the hypercall attack surface from two perspectives: *(i)* a targeted hypervisor; and *(ii)* an attacker triggering a hypercall vulnerability. Assuming the perspective of a targeted hypervisor, we provide in-depth technical information about hypercall vulnerabilities, and systematize and discuss the errors causing the hypercall vulnerabilities analyzed for this study (Section 4.2.1). We also demonstrate attacks triggering hypercall vulnerabilities, performed by executing the proof-of-concept code that we developed (Section 4.2.1), and discuss their effects (Section 4.2.2).

Assuming the perspective of an attacker, we construct attack models based on systematizing activities for executing hypercall attacks (Section 4.2.3).

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

4.2.1 Hypervisor's Perspective: Origins of Hypercall Vulnerabilities

In Table 4.2, for each hypercall vulnerability, we present the type of error that caused it and the effects of a hypercall attack triggering it. We stress that the error categories presented in Table 4.2 are not intended for general use and are defined for the convenience of discussion. We defined error categories as opposed to using existing taxonomies for classifying software errors (e.g., [TCM05]), since none of them fit well to our purpose; that is, we could not classify in the same category errors that share characteristics on which we focus in this chapter.

Table 4.2: Origins of the considered hypercall vulnerabilities and effects of attacks triggering the vulnerabilities [Error types: N (non-implementation); M (Implementation — value validation — missing value validation); I (Implementation — value validation — incorrect value validation); Inv (Implementation — incorrect implementation of inverse procedures)].

CVE ID	Error	Effect
CVE-2008-3687	M	Corrupted state
CVE-2009-2287	N	Hang/crash
CVE-2009-3290	N	Corrupted state
CVE-2009-3638	I	Crash/corrupted state
CVE-2009-4004	I	Crash/corrupted state
CVE-2010-3698	N	Crash
CVE-2011-4347	N	Crash
CVE-2012-1601	N	Corrupted state
CVE-2012-3494	I	Crash
CVE-2012-3495	M	Crash/corrupted state
CVE-2012-3496	N	Crash
CVE-2012-3497	M	Crash/corrupted state
CVE-2012-3516	M	Crash/corrupted state
CVE-2012-4461	N	Crash
CVE-2012-4538	N	Crash
CVE-2012-4539	M	Hang/crash
CVE-2012-5510	Inv	Crash/corrupted state
CVE-2012-5513	M	Crash/corrupted state
CVE-2012-5514	N	Hang
CVE-2012-5515	M	Hang
CVE-2012-5525	M	Crash
CVE-2013-0151	N	Crash
CVE-2013-0154	N	Crash
CVE-2013-1964	N	Crash/corrupted state/information leakage
CVE-2013-3898	M	Crash/corrupted state
CVE-2013-4494	N	Hang
CVE-2013-4553	N	Hang
CVE-2013-4554	N	Corrupted state
CVE-2013-5634	N	Crash
CVE-2014-1891	M	Crash/corrupted state
CVE-2014-1892	N	Crash
CVE-2014-1893	I	Crash/corrupted state
CVE-2014-1894	I	Crash/corrupted state
CVE-2014-1895	I	Crash/information leakage
CVE-2014-3124	N	Crash/corrupted state

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

Implementation Errors

We found several forms of implementation errors: *value validation errors* (i.e., missing value validation and incorrect value validation) and *incorrect implementation of inverse procedures*. It is obvious that the previously mentioned implementation errors are not exclusive to hypercall interfaces. However, in this section, we discuss issues related to these errors that are exclusive to hypercall interfaces.

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

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

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

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

Hypercall attack 2. *PHYSDEVOP_get_free_pirq*, an operation of the *physdev_op* hypercall, is used for allocating a Peripheral Component Interconnect Interrupt Request (PIRQ) to the VM from where it is invoked. In the handler of *PHYSDEVOP_get_free_pirq*, the return value of the function *get_free_pirq*, which corresponds to a free PIRQ if there is one, is used as an index for accessing an element of the array *pirq_irq* in order to mark a free PIRQ as allocated by writing *-1*. However, the return value of *get_free_pirq*

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

is not validated to be a valid PIRQ and not an error code (i.e., *-28*), which *get_free_pirq* returns if there is no free PIRQ. In case *get_free_pirq* returns an error code, the error code is used as an index for accessing *pirq_irq* and as a result, *-1* is written at the memory address **(pirq_irq - 28)*, which is mapped to the hypervisor. In order to trigger CVE-2012-3495, a request for allocating a PIRQ should be made when there are no free PIRQs. It can be concluded that a repetitive execution of *PHYSDEVOP_get_free_pirq* will eventually result in triggering of CVE-2012-3495. Depending on the exact memory layout of the hypervisor, it may crash. In case the hypervisor does not crash, it may be possible to carefully craft an exploit to achieve privilege escalation. We triggered CVE-2012-3495 by executing *PHYSDEVOP_get_free_pirq* 17 times, which caused the hypervisor to crash.[2]

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

Given that hypercalls to hypervisors are complex instructions that take many CPU cycles to execute, hypercalls have only a limited amount of time for execution. For instance, the hypercalls of the Microsoft hypervisor have only 50 microseconds to execute [Hyp]. Although some hypercalls are sufficiently simple so a given time limit is enough for completing their tasks, many hypercalls are complex and cannot complete their tasks in a set time limit. Therefore, hypervisors employ a *hypercall continuation* mechanism. This mechanism saves the execution state of a hypercall and returns control to the VM from where the hypercall had been invoked so that the hypercall can be resumed at a later time. Given that this context switching takes time, it can be concluded that hypercall continuations increase the performance overhead incurred by hypercalls. We observed that performing frequent value validations (e.g., of both input parameters and internal variables) may cause the frequency of hypercall continuations to increase, which incurs performance overhead and reduces the execution speed of hypercalls. Note that partially and fully paravirtualized VMs rely heavily on hypercalls, and therefore the execution speed of hypercalls is crucial for their performance.

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

Hypercall attack 3. CVE-2012-5513 is a vulnerability of *XENMEM_exchange*, an operation of the *memory_op* hypercall of the Xen hypervisor. In the handler of *XEN-*

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

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

Our observations presented above indicate that the trade-off between the two crucial properties of hypercall interfaces (i.e., performance and security) is currently an issue that should be addressed with great care. We discuss more on this issue in Section 4.3.

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

Hypercall attack 4. CVE-2012-5510 is a vulnerability of *GNTTABOP_set_version*, an operation of the *grant_table_op* hypercall of the Xen hypervisor. *GNTTABOP_set_version* is used for downgrading, from version 2 to version 1, and upgrading, from version 1 to version 2, the grant table of a VM. When a grant table is downgraded, the function *gnttab_unpopulate_status_frames*, inverse to *gnttab_populate_status_frames*, is used to deallocate page frames used only by a grant table of version 2. However, *gnttab_unpopulate_status_frames* does not properly perform the standard procedure for deallocating page frames — it does not remove the nodes that are associated with the frames being deallocated from the linked list *xenpage_list*, where the hypervisor stores frame information for memory management purposes. As a result, subsequent attempts to allocate the same frames by upgrading a grant table may lead to adding a node to *xenpage_list* that is a duplicate of the node that has not been removed by *gnttab_unpopulate_status_frames*. This causes corruption of *xenpage_list* and leads to

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

undefined behavior of the hypervisor. We triggered CVE-2012-5510 by downgrading and upgrading the grant table of a VM 58 times in a row, which caused the hypervisor to crash due to memory corruption.[2]

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

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

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

Non-implementation Errors

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

Hypercall attack 6. *GNTTABOP_copy*, an operation of the *grant_table_op* hypercall, is used for copying memory pages from a source VM (SVM) to a destination VM (DVM) with respect to data access permissions set by the SVM and the DVM using grant table entries (i.e., grants). Grant tables of version 2 support transitive grants, which are used for transitive assignment of permissions such that a transitive grant points to another grant. In order for the SVM to copy a page to the DVM, it must first acquire a grant from the hypervisor and, after the page is copied, it requests a release of the grant. However, vulnerable releases of the Xen hypervisor cannot properly handle the scenario where non-transitive grants of a version 2 grant table are used (i.e., a vulnerable hypervisor releases a non-transitive grant of a version 2 grant table as if it is a transitive grant).

The culprit of this error is that the hypervisor has been designed to support the specific scenario where transitive grants that point to a grant of the VM acquiring a grant are used and as a result, the hypervisor wrongly treats non-transitive grants as transitive grants. The triggering of CVE-2013-1964 results in an unrequested release of the first grant of the grant table of the VM that has executed *GNTTABOP_copy*. This may cause the hypervisor to crash, or corrupt its state, which enables further malicious activities. We triggered CVE-2013-1964 by executing *GNTTABOP_copy* such that a release of a version 2 non-transitive grant was requested, which caused the hypervisor to crash.[2]

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

As opposed to the majority of hypercall vulnerabilities due to implementation errors, all hypercall vulnerabilities due to non-implementation errors can be triggered by executing a regular hypercall, or a series of regular hypercalls, in a way such that the hypervisor cannot properly handle. By regular hypercalls, we mean hypercalls that are not specifically crafted for triggering vulnerabilities (i.e., with specifically crafted parameter values), which may be invoked as part of regular system operation. For example, we were able to crash the Xen hypervisor by simply executing *XENMEM_populate_physmap* (an operation of the *memory_op* hypercall) with valid parameter values from an auto-translated paravirtualized VM, only because *XENMEM_populate_physmap* is not intended for use by such a VM (we triggered the vulnerability CVE-2012-3496 [CVEc]).[3]

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

An overlapping objective of the areas of hypervisor security and system reliability is the prevention of system failures, which is critical in mission- and business-critical virtualized environments. The system security and reliability communities advocate the need for advances that enable the computer system reliability community to better prevent failures to be adapted and shared with the former for possible implementation [ASS11], [YL13]. Given the observations from our study, we recognize such a need when it comes to securing hypercall interfaces of hypervisors.

Our observation reveals that for the triggering of many of the current hypercall vulnerabilities (i.e., those due to non-implementation errors, see Table 4.2), the way in

which a hypercall, or a series of hypercalls, is executed plays a key role. This raises several issues related to the efficiency of existing mechanisms for securing hypercall interfaces, for example, intrusion detection and prevention systems. We discuss these issues in detail in Section 4.3.

4.2.2 Hypervisor's Perspective: Effects of Hypercall Attacks

We present in Table 4.2 the effects of the attacks triggering the considered vulnerabilities on the states of the targeted hypervisors (see column 'effect'). We observed that hypercall attacks are very efficient in obstructing the operation of a hypervisor, either by causing it to crash (effect *crash* in Table 4.2, see for example *Hypercall attack 6*, Section 4.2.1) or to hang (effect *hang* in Table 4.2, see for example *Hypercall attack 5*, Section 4.2.1). This is expected given that hypercalls perform system-critical operations. As a result, we argue that hypercall attacks can be very effective hypervisor denial-of-service (DoS) attacks, which are considered as very severe in mission- and business- critical virtualized environments where availability of hypervisors is of high importance.

Some hypercall attacks corrupt the state of the hypervisors they target without causing them to crash (effect *corrupted state* in Table 4.2), part of which only if a given condition is satisfied (effect *crash/corrupted state* in Table 4.2), for example, if the memory of the targeted hypervisor is laid out in a specific way (see for example *Hypercall attack 2*, Section 4.2.1). Our analysis of the post-attack states of hypervisors targeted by attacks corrupting their states revealed that severe intrusions leading to, for example, malicious code execution with hypervisor privilege, are probable (see for example *Hypercall attack 3*, Section 4.2.1). However, it does not seem likely that the execution of a single hypercall attack will lead to such an intrusion. On the contrary, our analysis showed that a hypercall attack is an effective mechanism for intruding hypervisors when executed as part of an elaborate multi-step attack, in which the task of the hypercall attack is to corrupt the state of the hypervisor and pave the way for further malicious activities. We found that this also holds for hypercall attacks resulting in an unauthorized retrieval of information (effect *information leakage* in Table 4.2), read from memory allocated to the hypervisor or a VM collocated with the VM from where a hypercall attack is executed.

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

4.2.3 Attacker's Perspective: Attack Models

Based on analyzing the attacks triggering the vulnerabilities listed in Table 4.1, we identified patterns of activities comprising a successful attack campaign. We then cate-

gorized the identified patterns into attack models. Models of hypercall attacks facilitate the development of approaches for improving the security of hypercall interfaces where mimicking attackers targeting hypercall interfaces is needed, for example, discovery of vulnerabilities by fuzzing. We discuss such approaches in detail in Section 4.3.

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

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

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

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

As one can observe from the hypercall attack models presented above, the majority of the models involve execution only of regular hypercalls, as opposed to the intuitive assumption that most of these models would involve execution of hypercalls specifically crafted for triggering vulnerabilities. This raises a number of issues that we discuss in detail in Section 4.3.

In Figure 4.1(a)–(f), we depict the hypercall attacks described in Section 4.2.1; that is, we provide examples of attacks that conform to the attack models that we define. In Figure 4.1(a)–(f), we depict only the hypercalls executed as part of a hypercall attack and relevant hypercall parameters (i.e., parameters identifying the executed hypercall, and, where applicable, parameters with values specifically crafted for triggering a vulnerability, which are marked in bold).

4.3 Extending the Frontiers

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

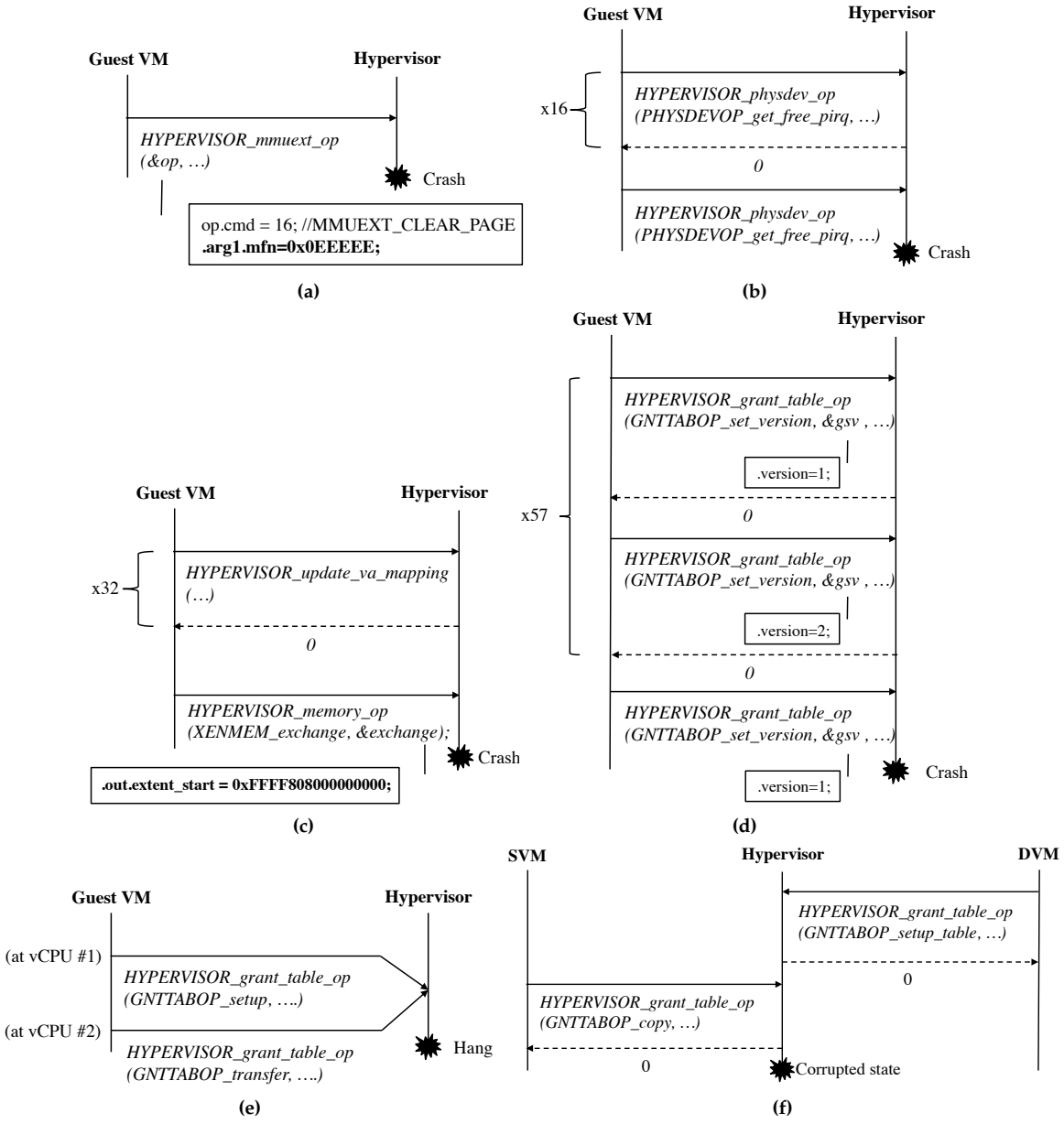


Figure 4.1: (a) *Hypercall attack 1* — execution of a single hypercall with a specifically crafted parameter value, (b) *Hypercall attack 2* — repetitive execution of a single regular hypercall, (c) *Hypercall attack 3* — setup phase and execution of a single hypercall with a specifically crafted parameter value, (d) *Hypercall attack 4* — repetitive execution of multiple regular hypercalls, (e) *Hypercall attack 5* — execution of a series of regular hypercalls, (f) *Hypercall attack 6* — setup phase and execution of a single regular hypercall.

4.3.1 Vulnerability Discovery and Secure Programming Practices

Vulnerability Discovery

We note that publicly available techniques and tools for fuzzing hypercalls are lacking. Such tools can be very effective for discovering hypercall vulnerabilities, especially those due to implementation errors that can be triggered by executing specifically crafted hypercalls. Hypercall fuzzing tools may contribute towards discovering hypercall vulnerabilities in a time-efficient manner by enabling vulnerability analysts, especially those who had not been involved in the design and/or development of the hypercalls that they analyze, to conveniently discover vulnerabilities. The hypercall attack models that we presented in Section 4.2.3 can serve as a basis for designing and developing generation-based fuzzers for hypercalls.

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

Given the severity of hypercall attacks (see Section 4.2.2), we argue that tools for time-efficient discovery of hypercall vulnerabilities, such as fuzzers, should be designed and developed. We observed that the time period between the introduction and the discovery of many of the hypercall vulnerabilities considered in this study is long. For example, the vulnerability CVE-2012-3494 [CVEa] was present in versions of the Xen hypervisor released over the course of 2 years and 8 months.[4] Our analysis of the hypercall vulnerabilities due to implementation errors revealed that the discovery of some of them is trivial in case fuzzing techniques are used (e.g., CVE-2012-3494).

While fuzzers are effective for discovering some vulnerabilities due to implementation errors, they are not that effective for discovering vulnerabilities due to non-implementation errors. Note that the latter are triggered by executing regular hypercalls in a way such that the hypervisor cannot properly handle. We argue that such vulnerabilities can be discovered using formal verification methods that aim at discovering the non-implementation errors causing hypercall vulnerabilities (see Section 4.2.1), which are currently lacking. We refer the reader to [LS09] for an overview of the challenges that apply to formally verifying hypervisors.

The Microsoft Hyper-V verification project [LS09] has made important achievements towards the functional verification of the Hyper-V hypervisor. Many researchers, such

[4] This vulnerability has been introduced in Xen 4.0.0 (released 7 April 2010), a patch has been released on 5 September 2012, and it has been fixed in Xen 4.1.4 (released 18 December 2012).

as Alkassar et al. [AHPP10] and Barthe et al. [BBCL11], develop formal models of various hypervisor components used for verifying functional properties of hypervisors, for example, isolation between VMs. Freitas and McDermott [FM11] formally modelled the hypercall interface of the Xen hypervisor to re-engineer it into interface enforcing information-flow security (i.e., one VM should not flow/leak information into another VM). We argue that the development of models for formally verifying the functional correctness of hypercall interfaces, with a focus on hypervisor security, would be a major achievement towards reducing the number of hypercall vulnerabilities due to non-implementation errors.

Secure Programming Practices

In Section 4.2.1, we mentioned that the trade-off between performance and security of hypercall interfaces is currently an important issue. We observed that missing value validations errors comprise slightly less than 60 percent of all implemenation errors that we reviewed. Missing value validation errors can be addressed by adding program code verifying values of variables. However, when analyzing the hypercall vulnerabilities of the Xen hypervisor, we observed that performing frequent value validations (e.g., of both input parameters and internal variables) may cause an increased frequency of hypercall continuations, therefore reducing the execution speed of hypercalls and increasing the performance overhead incurred by them (see Section 4.2.1).

We argue that secure hypercall programming practices enforcing, for example, value validations of all variables used within a given hypercall handler, should be developed. Given our observations presented in Section 4.2.1, one may conclude that the application of such practices would lead to the development of secure hypercall handlers free of missing value validation errors, however, executing slowly (e.g., hypercalls of the Xen hypervisor may exhibit an increased frequency of overhead incurring hypercall continuations). Therefore, a major challenge is the development and application of programming practices for developing hypercall handlers such that rigorous value validations are performed at a reasonable performance cost.

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

4.3.2 Security Mechanisms

The research and industrial communities have designed and developed security mechanisms for detecting and/or preventing hypercall attacks targeting hypervisors, for

example, intrusion detection and prevention systems (IDPSs). Such security mechanisms are

- *Collabra* — Bharadwaja et al. [BSNS11b] designed a distributed anomaly-based IDPS that labels hypercall invocations as malicious or benign based on hypercall parameter values;
- *Message Authentication Code/Hypercall Access Table (MAC/HAT)* — Le [HL09] designed an IDPS that uses policies, which include hypercall parameter values and hypercall call sites, for labeling a given hypercall invocation as benign or malicious;
- *RandHyp* — Wang et al. [WCMX12] designed an IDPS that randomizes hypercall parameter values in order to detect and block the execution of malicious hypercall invocations that originate from untrusted locations (e.g., attacker's loadable kernel module);
- *XSM-FLASK* — the XSM-FLASK security module of the Xen hypervisor enables mandatory access control based on policies, which include hypercalls and hypercall parameter values [XSM].

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

As we mentioned in Section 4.2.3, all vulnerabilities due to non-implementation errors and some due to implementation errors can be triggered by executing hypercalls with regular parameter values in a specific way (see for example Figure 4.1(b), (d)–(f), Section 4.2.3). Note that the vulnerabilities due to non-implementation errors make approximately 50 percent of all vulnerabilities that we analyzed (see Table 4.2). Therefore, one may infer that current security mechanisms do not cover a significant portion of the hypercall attack space. In Figure 4.2, we depict the coverage of current security mechanisms with respect to the attack models (attack phase only) defined in Section 4.2.3.

Mechanisms for securing hypercall interfaces that consider the way in which hypercalls are executed in addition to hypercall parameter values and call sites, are lacking. Being able to detect and/or prevent not only some attacks triggering vulnerabilities due to implementation errors, we argue that such security mechanisms would be effective against most of the hypercall attacks that can be seen in practice. For example, the attack depicted in Figure 4.1(e) can be detected if the allocation of hypercalls to vCPUs for execution is taken into account as a factor for detecting attacks. The execution of the *GNTTABOP_SETUP* and *GNTTABOP_TRANSFER* hypercalls at the same time, each at different vCPU, may be considered as an abnormal activity by an anomaly-based IDPS.

Besides the issue discussed above, we raise the issue that currently there are no

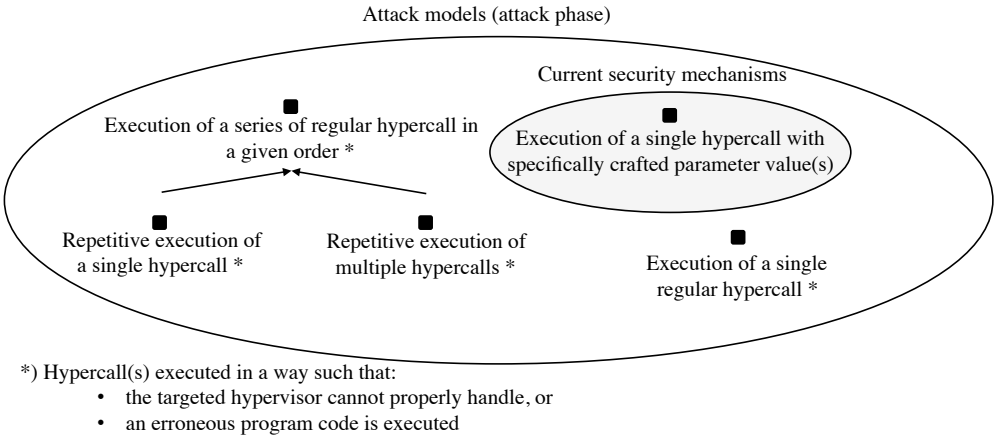


Figure 4.2: Coverage of current security mechanisms with respect to hypercall attack models.

approaches for generating workloads that contain representative hypercall attacks. The latter are crucial for the accurate and rigorous evaluation of IDPSs designed for detecting and preventing hypercall attacks. An inaccurate evaluation of IDPSs may lead to the deployment of misconfigured or ill-performing IDPSs in production environments, increasing the risk of security breaches. Approaches for generating workloads that contain hypercall attacks may be used for purposes beyond evaluation of IDPSs (i.e., for any cyber security experiment where controlled generation of malicious workloads is needed, such as verification of XSM-FLASK policies).

The main reason for the issue mentioned above is the lack of publicly available information on hypercall vulnerabilities and attacks, a problem that this chapter is focussing on. Such information is a basis for the development of approaches for generating artificial hypercall attacks that closely resemble real ones. Bharadwaja et al. [BSNS11b], Wang et al. [WCMX12], and Le [HL09] acknowledge the above issue and explicitly stress that it is a major problem stating, for example, *"We have some difficulties to fully evaluate the efficiency of our hypercall protection measures ... known attack codes on Xen virtualization are not available."* (Le [HL09], Section 5.1.2 — "Attack Experimental Issues", pg. 47). The models that we presented in Section 4.2.3 may serve as a basis for designing tools for generating activities representative of those of an attacker executing hypercall attacks. Preliminary work in this area is described in [MPA+13].

4.4 Summary: Lessons Learned

With the goal of increasing the amount of publicly available information on vulnerabilities of hypervisors' hypercall handlers (i.e., hypercall vulnerabilities) and attacks triggering them (i.e., hypercall attacks), in this chapter, we analyzed a set of 35 hypercall vulnerabilities. Our vulnerability analysis approach consisted of analyzing

publicly available reports describing the considered vulnerabilities (e.g., CVE reports, security advisories), reverse-engineering the patches fixing the vulnerabilities, and developing proof-of-concept code, which allowed us to trigger the vulnerabilities and closely observe all stages of the life cycle of a typical hypercall attack. We made the following observations:

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

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

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

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

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

where the attack models involving execution of regular hypercalls assume that the hypercalls are executed in a way such that:

(a) the targeted hypervisor cannot properly handle (which is typical for triggering vulnerabilities due to non-implementation errors); or

(b) an erroneous program code is executed (which is typical for triggering some vulnerabilities due to implementation errors).

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

Chapter 5

Evaluation of Intrusion Detection Systems Using Attack Injection

As we stated in Section 1.1, the rigorous evaluation of intrusion detection systems (IDSs) in virtualized environments is crucial for preventing breaches in virtualized environments. For instance, one may compare multiple IDSs in terms of their attack detection accuracy in order to identify the optimal IDS.

We focus in this chapter on evaluating IDSs designed to detect hypercall attacks, which is one of the goals of this thesis (see Section 1.3). Workloads that contain virtualization-specific attacks (i.e., attacks targeting hypervisors — hypercall attacks) are a key requirement for evaluating the attack detection accuracy of IDSs designed to detect hypercall attacks (see Section 1.2.1 and Section 1.3). However, the generation of such workloads is challenging since publicly available scripts that demonstrate hypercall attacks are very rare [MPA+14], [HL09]. An approach towards addressing this issue is attack injection, which enables the generation of representative IDS evaluation workloads. Attack injection is controlled execution of attacks during regular operation of the environment where an IDS under test is deployed. The injection of attacks is performed with respect to attack models constructed by analysing realistic attacks. Attack models are systematized activities of attackers targeting a given attack surface.

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

Our approach uses live IDS testing, since existing IDSs designed to detect hypercall attacks perform on-line monitoring. Further, it enables the evaluation of IDSs that do and do not require training (i.e., it involves IDS training, which is needed for evaluating IDSs that require training, see Section 2.1.2). We demonstrate the application

and practical usefulness of the approach by evaluating Xenini [MM11], a representative IDS designed to detect hypercall attacks. We inject realistic attacks triggering publicly disclosed hypercall vulnerabilities and specifically crafted evasive attacks. We extensively evaluate Xenini considering multiple configurations of the IDS. Such an extensive evaluation would not have been possible before due to the previously mentioned issues.

The work presented in this chapter has been published in [MPA+15].

This chapter is organized as follows: in Section 5.1, we provide the essential background and discuss related work; in Section 5.2, we present an approach for evaluating IDSs; in Section 5.3, we introduce the hInjector tool; in Section 5.4, we demonstrate the application of the proposed approach; in Section 5.5, we conclude this chapter.

5.1 Background and Related Work

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

With recent advances in hardware design, paravirtualizing privileged instructions and pagetables often does not provide performance benefits over full (native) virtualization. However, paravirtualizing the other VM components mentioned above is beneficial. As a result, different modes of virtualization have emerged, many of which involve paravirtualizing VM components of fully virtualized VMs. In Figure 5.1, we depict the spectrum of virtualization modes as conceptualized by Dunlap [Dun].

	Disk and network devices	Interrupts and timers	Platform components	Instructions and pagetables
Full virtualization	F	F	F	F
	P	F	F	F
	P	P	F	F
	P	P	P	F
Full paravirtualization	P	P	P	P

Figure 5.1: Spectrum of virtualization modes [P: paravirtualized, F: fully virtualized].

Hypercalls are operations that VMs use for working with their paravirtualized components. They are software traps from a kernel of a VM to the underlying hypervisor (see Section 1.3). At this time hypercalls are widely used in practice since the majority of the current virtualization modes involve paravirtualization (see Figure 5.1). For

instance, Amazon offers partially and fully paravirtualized VMs as part of their Elastic Computing Cloud (EC2) service [ama].

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

In Chapter 4 of this thesis (see also [MPA+14]), we analyzed 35 publicly disclosed hypercall vulnerabilities and identified patterns of activities for triggering the considered vulnerabilities. We categorized the identified patterns into the following attack models:

- *setup phase* (optional) — execution of one or multiple regular hypercalls (i.e., hypercalls with regular parameter value(s) that may be executed during regular system operation) setting up the virtualized environment as necessary for triggering a given hypercall vulnerability; and
- *attack phase* — execution of a single regular hypercall, or a hypercall with specifically crafted parameter value(s); or, execution of a series of regular hypercalls in a given order.

In this chapter, we use these models for injecting hypercall attacks.

Intrusion detection Given the high severity of hypercall attacks, the research and industrial communities have developed IDSs that can detect such attacks. Examples are Collabra [BSNS11b], Xenini [MM11], Covert Channel (C^2) Detector [WDW+14], Wizard [SSG08], Mandatory Access Control/Hypercall Access Table (MAC/HAT) [HL09], RandHyp [WCMX12], and Open Source Security (OSSEC) [oss]. Most of these IDSs have the following characteristics in common:

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

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

IDS evaluation and attack injection The accurate and rigorous evaluation of IDSs is crucial for preventing security breaches. IDS evaluation workloads that contain realistic

attacks are a key requirement for such an evaluation. In Section 1.3, we stated that IDSs designed to detect hypercall attacks currently cannot be evaluated in a rigorous manner due to the lack of publicly available attack scripts that demonstrate hypercall attacks. Attack injection is a method addressing this issue, which is in the focus of this chapter.

To the best of our knowledge, we are the first to focus on evaluating IDSs designed to operate in virtualized environments, such as IDSs designed to detect hypercall attacks. Further, we are the first to consider the injection of hypercall attacks and of attacks targeting hypervisors in general. Pham et al. [PCKI11] and Le et al. [LGT08] focus on injecting generic software faults directly into hypervisors. This is not suitable for evaluating IDSs — IDSs do not monitor states of hypervisors since they are not relevant for detecting attacks in a proactive manner.

Fonseca et al. [FVM14] present an approach for evaluating network-based IDSs, which involves injection of attacks. They built Vulnerability Injector, a mechanism that injects vulnerabilities in the source code of web applications, and an Attack Injector, a mechanism that generates attacks triggering injected vulnerabilities. There are fundamental differences between our work and the work of Fonseca et al. [FVM14], which is focussing on attack injection at application level. This includes the characteristics of the IDSs in focus, the required attack models, and the criteria for designing procedures and tools for injecting attacks.

5.2 Approach

Figure 5.2a shows our approach, which has two phases: planning and testing. The planning phase consists of: *(i)* specification of an IDS monitoring landscape (i.e., specifying a virtualized environment where the IDS under test is to be deployed), *(ii)* characterization of benign hypercall activities (i.e., making relevant observations about the benign hypercall activities), and *(iii)* specification of attack injection scenarios (Section 5.2.1). The testing phase consists of: *(i)* IDS training, *(ii)* attack injection, and *(iii)* calculation of metric values (Section 5.2.2). The activities of the testing phase are performed based on observations made in the planning phase. IDS training needs to be performed only when evaluating an IDS that requires training (i.e., an anomaly-based IDS).

5.2.1 Planning

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

- *virtualization mode* influences which hypercalls can be executed,
- *workloads* influence which system calls can be executed, many of which map to hypercalls, and

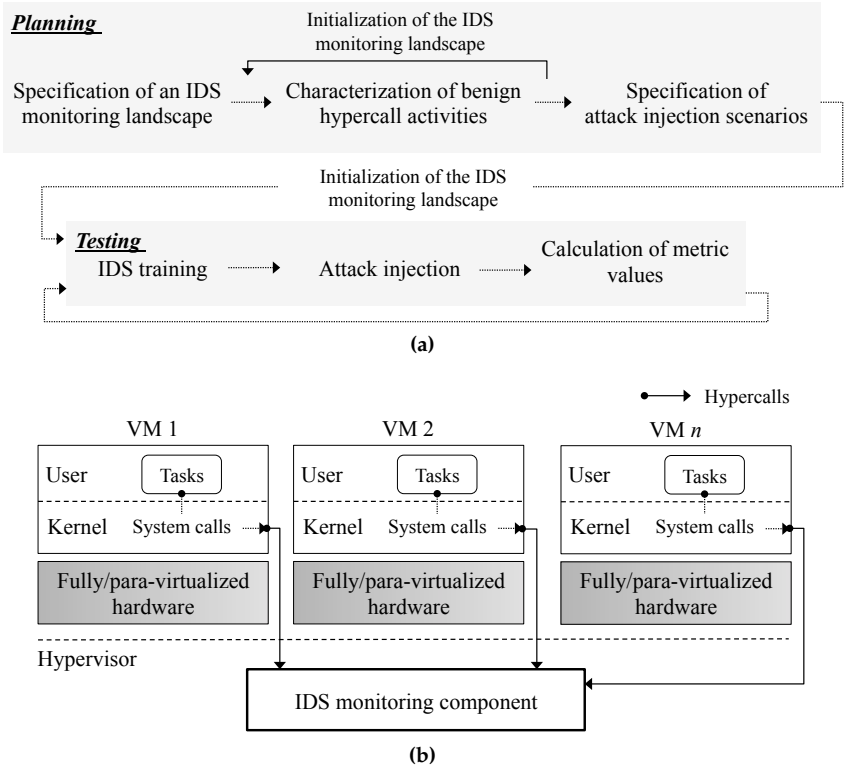


Figure 5.2: (a) Approach for evaluating IDSs, (b) IDS monitoring landscape.

- *system architecture and hardware* influence the VM's interface, and the type and frequency of hypercalls needed (e.g., page table update operations, which take place when page swapping occurs due to insufficient memory).

The aggregate of these characteristics across all VMs on a hypervisor is the *monitoring landscape* of an IDS designed to detect hypercall attacks. Figure 5.2b depicts an IDS monitoring landscape. The first activity of the planning phase of our approach is to specify an IDS monitoring landscape by defining the characteristics above for the test system. By defining workloads, we mean specifying drivers generating workloads in an automated and repeatable manner. By defining hardware, we mean allocating an amount of hardware resources to VMs that is fixed over time (i.e., disabling CPU or memory ballooning). We discuss more on the importance of specifying an IDS monitoring landscape in Section 5.2.2.

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

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

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

In order to estimate steady-state, an IDS evaluator should first *initialize the IDS monitoring landscape*; that is, bring the VMs in the landscape to the state after their creation and start workloads in the VMs. Then the steady-state of the benign hypercall activities of a VM may be estimated by setting a target for the slope of a growth curve depicting S_t until a given time t_{max}. The slope of such a curve, when observed over a given period, indicates the rate of first-time occurring variations of the detection-relevant property in the period. Letting σ be a target for the slope of a growth curve over a period $t_s = t_{s2} - t_{s1}$, we have $0 <= \frac{S_{t_{s2}} - S_{t_{s1}}}{t_s} <= \sigma$. This process may be repeated multiple times for different values of t_{max} to experimentally determine σ for each VM.[1] Attacks should be injected from a VM until time t_{max}, but only after the VM's hypercall activity has reached steady-state.

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

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

Specification of attack injection scenarios Two characteristics distinguish each attack injection scenario: *attack content* and *attack injection time*.

Attack content is the detection-relevant property of a hypercall attack in the context of a given IDS evaluation study (e.g., a specific sequence of hypercalls). Specification of attack content enables the injection of attacks that conform to representative attack models (see Section 5.1). In addition, it enables the injection of evasive attacks, for example, attacks that closely resemble common regular activities; that is, these attacks

[1] This raises the question whether hypercall activities are repeatable. We discuss this topic in Section 5.2.2.

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

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

- *Base rate*: Base rate is the prior probability of an intrusion (attack). The error occurring when the attack detection accuracy of an IDS is assessed without taking the base rate into account is known as the base rate fallacy [Axe00] (see Section 3.3.2). The specification of attack injection times provides a close estimation of the actual base rate in the testing phase. As we demonstrate in Section 5.4, base rate can be estimated by considering the number of injected attacks and the number of variations of the detection-relevant property that have occurred during attack injection. The latter is estimated based on the statistic 'average rate of occurrence of the detection-relevant property'.
- *IDS evasive properties*: Specification of the attack injection time enables the injection of "smoke screen" evasive attacks. In the context of this chapter, the "smoke screen" technique consists of delaying the invocation of the hypercalls comprising an attack such that a given amount of benign hypercall activity occurs between each hypercall invocation. This is an important test since some IDSs have been shown to be vulnerable to such attacks (e.g., Xenini; see [WS02]).

5.2.2 Testing

IDS training IDS training is the first activity of the testing phase. We require reinitialization of the IDS monitoring landscape between the planning and testing phases (see Figure 5.2a). The rationale behind this is practical: many parameters of the existing IDSs designed to detect hypercall attacks (e.g., length of IDS training period, attack detection threshold) require a priori configuration. These parameters are tuned based on observations made in the planning phase (see Section 5.2.1). This raises concerns related to the non-determinism of hypercall activities, a topic that we discuss in paragraph 'on repeatability concerns'.

Attack injection For this critical step, we developed a new tool called *hInjector*. Section 5.3 introduces this tool and describes how it is used.

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

On repeatability concerns Observations and decisions made in the planning phase might be irrelevant if hypercall activities are highly non-deterministic and therefore not repeatable. For example, the benign hypercall activities occurring in the testing phase may not reach steady-state at a point in time close to the estimated one in the planning phase.

In addition, metric values reported as end-results of an evaluation study, where workloads that are not fully deterministic are used, have to be statistically accurate. This is crucial for credible evaluation. Principles of statistical theory impose metric values to be repeatedly calculated and their means to be reported as end-results. Therefore, we require repeated execution of the testing phase (see Figure 5.2a). However, this may be time-consuming if the number of needed repetitions is high due to high non-determinism of hypercall activities.

Specifying an IDS monitoring landscape as we define it (see Section 5.2.1) alleviates the above concerns; that is, it helps to reduce the non-determinism of hypercall activities by removing major sources of non-determinism, such as non-repeatable workloads. This is in line with Burtsev [Bur13], who observes that, given repeatability of execution of VMs' user tasks is preserved, VMs always invoke the same hypercalls. We acknowledge that achieving complete repeatability of hypercall activities by specifying VM characteristics is infeasible. This is mainly due to the complexity of the architectures and operating principles of kernels.

In Section 5.4, we empirically show that, provided an IDS monitoring landscape is specified, a VM's hypercall activities exhibit repeatability to an extent sufficient to conclude that: *(i)* the decisions and observations made in the planning phase are of practical relevance when it comes to IDS testing, and *(ii)* the number of measurement repetitions needed to calculate statistically accurate metric values is small. This is in favor of the practical feasibility of our approach, which involves repeated initialization of an IDS monitoring landscape.

5.3 hInjector

hInjector is a tool for injecting hypercall attacks. It realizes the attack injection scenarios specified in the planning phase (see Section 5.2.1). The current implementation of hInjector is for the Xen hypervisor, but the techniques are not Xen-specific and can be ported to other hypervisors.

hInjector supports the injection of attacks crafted with respect to the attack models that we presented in Chapter 4 of this thesis. We extend these attack models with a model involving different hypercall call sites. Hypercall call sites are one of the detection-relevant properties that existing IDSs designed to detect hypercall attacks analyze. We consider that hypercalls can be executed from *regular* or *irregular* call sites. The latter is typically a hacker's loadable kernel module (LKM) used to mount hypercall attacks.

Our design criteria for hInjector are *injection of realistic attacks*, *injection during regular*

system operation, and *non-disruptive attack injection*. These criteria are crucial for the representative, rigorous, and practically feasible IDS evaluation. We discuss more in Section 5.3.2.

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

5.3.1 hInjector Architecture

Figure 5.3 depicts the architecture of hInjector. It shows the primary components: *Injector*, *LKM*, *Filter*, *Configuration*, and *Logs*. We refer to the VM from where hypercall attacks are injected as the malicious VM (MVM). We also depict a typical IDS designed to detect hypercall attacks, with components in the hypervisor and a designated VM (DesVM), co-located with MVM (see Section 5.1). The IDS monitors the MVM's hypercall activity by monitoring virtual CPU registers and the virtual memory of MVM using its hypervisor component.

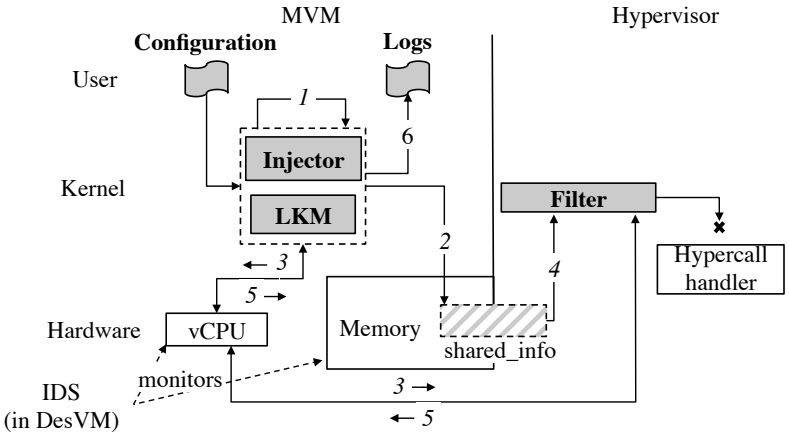


Figure 5.3: The architecture of hInjector.

The *Injector* component, deployed in the MVM's kernel, intercepts at a given rate hypercalls invoked by the kernel and modifies hypercall parameter values on-the-fly *(i)* making them specifically crafted for triggering a vulnerability, or *(ii)* replacing them with random, irregular values that an IDS may label as anomalous. The Injector injects hypercalls invoked from a regular call site (i.e., from the kernel address space). We discuss more on Injector in Section 5.3.3.

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

The *Filter* component, deployed in the hypervisor's hypercall handlers, identifies hypercalls injected by the Injector or the LKM, blocks the execution of the respective

hypercall handlers, and returns valid error codes. The Filter identifies injected hypercalls based on information stored by the Injector/LKM in the *shared_info* structure, a memory region shared between a VM and the hypervisor. To this end, we extended *shared_info* with a string field named *hypercall identification (hid)*, which contains identification information on injected hypercalls. We discuss more about the Filter when we discuss the design criterion 'non-disruptive attack injection' in Section 5.3.2.

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

The *Logs* are user files containing records about invoked hypercalls that are part of attacks; that is, hypercall IDs and parameter values, as well as timestamps. The logged data serves as reference data (i.e., as "ground truth") used for distinguishing false positives from injected attacks and calculating IDS attack detection accuracy metrics, such as true and false positive rate. We discussed the importance of "ground truth" information in Section 3.2.5, Chapter 3 of this thesis.

We now present an example of the implemented hypercall attack injection procedure. Figure 5.3 depicts the steps to inject a hypercall attack by the LKM:

(1) the LKM crafts a parameter value of a given hypercall as specified in the configuration

(2) the LKM stores the ID of the hypercall, the number of the crafted parameter, and the parameter value in *hid*;

(3) the LKM passes the hypercall to MVM's virtual CPU (vCPU), which then passes control to hypervisor;

(4) the Filter, using the data stored in *hid*, identifies the injected hypercall when the respective hypercall handler is executed;

(5) the Filter updates *hid* indicating that it has intercepted the injected hypercall, then returns a valid error code to block execution of the handler;

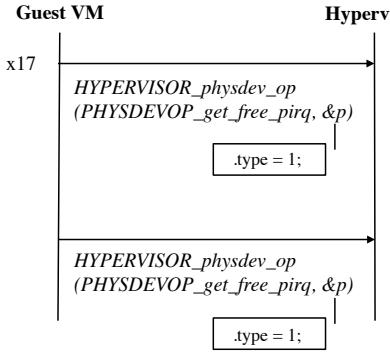
(6) after the error code arrives at MVM's kernel, the LKM first verifies whether *hid* has been updated by the Filter and then logs the ID and parameter values of the injected hypercall.

5.3.2 hInjector Design Criteria

Injection of realistic attacks The injection of realistic attacks is crucial for the representative IDS evaluation. In order to inject realistic hypercall attacks, hInjector requires representative hypercall attack models. hInjector supports the injection of

attacks crafted with respect to arbitrary attack models, for example, the models that we presented in Chapter 4 of this thesis.

We developed proof-of-concept code for triggering the hypercall vulnerabilities that we analyzed (see Chapter 4 and [MPA+14]). We developed this code based on reverse-engineering the released patches fixing the considered vulnerabilities. The proof-of-concept code enables granularization of the attack models. For example, we can specify specific parameter values or the order of a series of hypercalls that trigger a hypercall vulnerability. This enables the injection of realistic hypercall attacks, crafted to trigger publicly disclosed hypercall vulnerabilities. In Figure 5.4a, we show how we triggered the vulnerability CVE-2012-3495 [CVEb] of the Xen hypervisor in a testbed environment. In Figure 5.4b, we present the configuration of hInjector for injecting an attack triggering CVE-2012-3495. Configuration files for injecting attacks that trigger publicly disclosed hypercall vulnerabilities are distributed with hInjector.




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

(b)

Figure 5.4: (a) Triggering CVE-2012-3495 [the hypercall *physdev_op* is executed 18 times: the value of its first parameter is 23 (*PHYSDEVOP_get_free_pirq*); the value of the field *type* of its second parameter (struct *physdev_get_free_pirq*) is 1], (b) Configuration of hInjector for injecting an attack triggering CVE-2012-3495.

Injection during regular system operation Benign activities, mixed with attacks, are needed to subject an IDS under test to realistic attack scenarios (see Section 3.2). hInjector is designed to inject hypercall attacks *during* regular operation of guest VMs. Thus, provided that during an IDS evaluation experiment representative user tasks run in the VMs in the IDS monitoring landscape, the presence of representative benign hypercall activities is guaranteed.

Non-disruptive attack injection The state of the hypervisor or the VM(s) from where attacks are injected may be altered by the attacks injected by hInjector. This may cause crashes obstructing the execution of the IDS evaluation process. Filter prevents crashes by blocking the execution of the hypervisor's handlers that handle the injected

hypercalls. This preserves the states of the hypervisor and of the VM(s) from where attacks are injected, and, in addition, it ensures that injected attacks do not impact the operation of the IDS under test, which normally has components in the hypervisor and in a VM (see Section 5.1). After blocking the execution of hypervisor's handlers, Filter returns valid error codes. This allows the control flow of the kernel of the VM from where hypercall attacks are injected to properly handle failed hypercalls that have been executed by it and have been modified by the Injector on-the-fly.

5.3.3 Injector: Performance Overhead

The rate at which the kernel invokes hypercalls is high (i.e., in some cases more than 30000 hypercalls per second, see Section 5.4). Therefore, Injector, which manipulates hypercalls on-the-fly, can easily incur intolerable system performance overhead. We made the following observation when developing Injector: manipulating orders of series of hypercalls is very performance-expensive; therefore, Injector can manipulate only hypercall parameter values. Further, we measured the overhead incurred by Injector on the execution rate of hypercalls, relative to this rate when Injector is inactive, when replacing regular hypercall parameter values with random, irregular values. In Figure 5.5, we depict this overhead, which we measured as follows. We deployed Injector in the kernel of a Debian 8.0 operating system running on top of Xen 4.4.5. We invoked the *mmuext_op* hypercall 40000 times using a loadable kernel module. We measured the time, in microseconds (μs), needed for the invoked hypercalls to complete their operation ('Execution time' in Figure 5.5) in scenarios where:

(i) Injector is inactive ('Base' in Figure 5.5), and

(ii) Injector manipulates the value of the second parameter of *mmuext_op* at the rate of 1:50 (i.e., Injector manipulates parameter value once in 50 invocations of *mmuext_op*), 1:100, 1:500, 1:1000, and 1:10000.

We repeated the measurements 30 times and averaged the results.

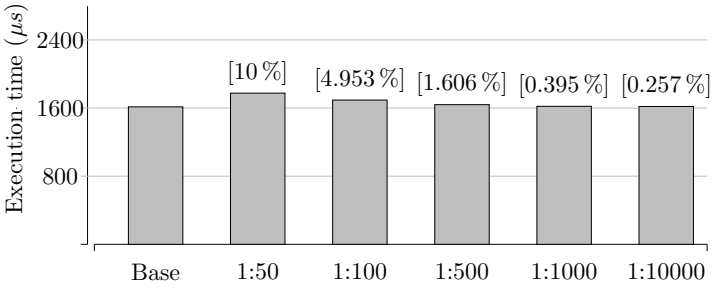


Figure 5.5: Overhead incurred by Injector [measurements of the incurred overhead are depicted in square brackets].

Based on the results from the above experiment, we conclude that a user should constrain the rate at which Injector manipulates hypercall parameter values to a value such that the incurred overhead is not higher than 2%. This is important since we observed that overheads higher than 2% often cause noticeable system slowdowns or crashes. We showed that Injector normally incurs overheads higher than 2% when it manipulates hypercall parameter values approximately once in less than 500 hypercall invocations (see Figure 5.5). Note that overheads incurred by Injector for hypercalls other than *mmuext_op* do not significantly differ from those depicted in Figure 5.5 since the implementation of Injector is the same for all hypercalls.

5.4 Case Study

We now demonstrate the application of our approach by evaluating Xenini [MM11] following the steps presented in Section 5.2. Xenini is a representative anomaly-based IDS. It uses the popular Stide [FHSL96] method. Xenini slides a window of size k over a sequence of n hypercalls and identifies mismatches (anomalies) by comparing each k-length sequence with regular patterns learned during IDS training. Xenini records the number of mismatches as a percentage of the total possible number of pairwise mismatches for a sequence of n hypercalls (i.e., $(k-1)(n-k/2)$). We call this percentage *anomaly score*. When the anomaly score exceeds a given threshold $th \in [0;1]$, Xenini fires an alert. For the purpose of this study, we configured Xenini such that its detection-relevant property is sequences of hypercall IDs of length 4 (i.e., $k = 4$; $n = 10$).

It is important to emphasize that we focus on demonstrating the feasibility of attack injection in virtualized environments for IDS testing purposes and not on discussing the behavior of Xenini in detail or comparing it with other IDSs. We specify arbitrary attack injection scenarios and evaluate Xenini with the sole purpose of demonstrating all steps and functionalities of the proposed approach. We refer the reader to Section 7.2 for an overview of further application scenarios.

5.4.1 Case Study: Planning

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

In Figure 5.6, we depict the deployment of SPECvirt_sc2013 as an IDS monitoring landscape. The workload drivers generate workloads that map to hypercalls. We used Xen 4.4.1 as hypervisor and we virtualized the VMs using full paravirtualization. We did not use any other virtualization mode because of a technical limitation; that is, the xentrace tool [xena], which we use to capture benign hypercall activities in files for processing off-line, currently supports only full paravirtualization. However, support for other virtualization modes is currently being implemented.

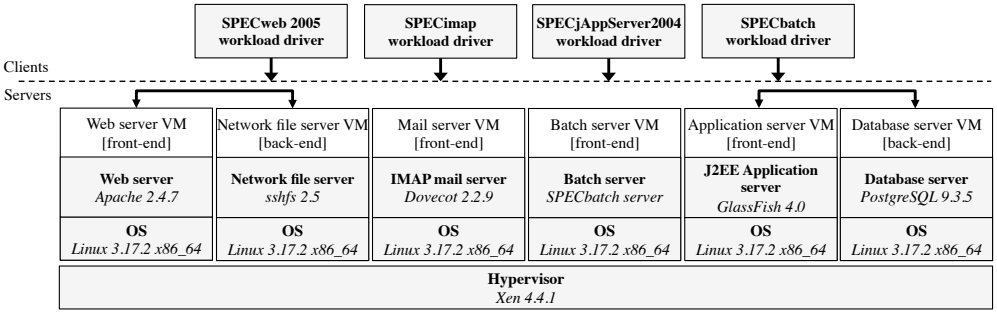


Figure 5.6: SPECvirt_sc2013 as an IDS monitoring landscape [Internet Message Access Protocol (IMAP); Java 2 Enterprise Edition (J2EE)].

To each server VM, we allocated 8 virtual CPUs pinned to separate physical CPU cores of 2 GHz, 3 GB of main memory, and 100 GB of hard disk memory. In Figure 5.6, we depict the operating systems and architectures of the server VMs, and the server software we deployed in the VMs.[2]

Characterization of benign hypercall activities We now estimate steady-states of the benign hypercall activities of the server VMs and calculate the relevant statistics (see Section 5.2.1). We initialized the IDS monitoring landscape and deployed Xenini before the characterization. We used xentrace [xena], the tracing facility of the Xen hypervisor, to capture hypercall activities in trace files.

Figure 5.7a–5.7f show growth curves depicting S_t until time t_{max} = 5500 seconds for each server VM (see the curves entitled 'Run 1'). We set the target σ to 15 over a time period of 100 seconds for the slope of each growth curve. In Table 5.1, column 'Run 1', we present t_s (in seconds – sec.), which is the time at which the VMs' hypercall activities reach steady-state. We also present r (in number of occurrences per second – occ./sec.), which is the average rate of occurrence of the detection-relevant property. We also calculated the statistic 'number of occurrences of each variation of the detection-relevant property' (not presented in Table 5.1), which we use to craft "mimicry" attacks (see Section 5.4.2).

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

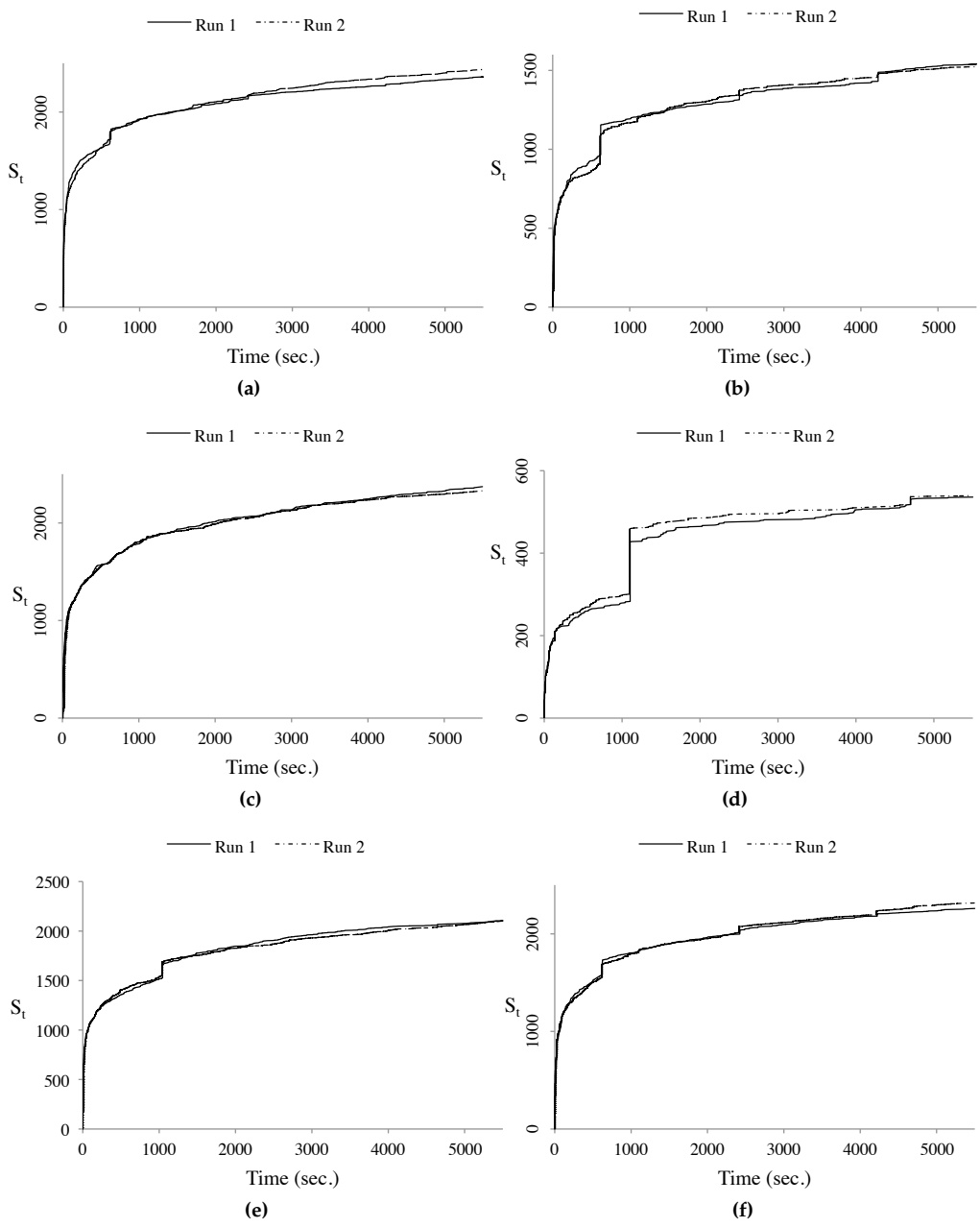


Figure 5.7: Growth curves: (a) web, (b) network file, (c) mail, (d) batch, (e) application, (f) database server VM.

Table 5.1: Benign workload characterization.

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

We now empirically show that, provided an IDS monitoring landscape is specified, VMs' hypercall activities exhibit repeatability in terms of the characteristics of interest to an extent sufficient for accurate IDS testing (see Section 5.2.2). We performed the above characterization campaign twice and compared the results. In Figure 5.7a–5.7f, we depict the obtained growth curves (see the curves entitled 'Run 1' and 'Run 2'). These curves are very similar, which indicates that the characteristics of the VMs' hypercall activities of interest are also similar. In Table 5.1, we present t_s and r for each server VM (see column 'Run 1' and 'Run 2'). We observe a maximum difference of only 17 sec. for t_s and 26.4 occ./sec. for r. We repeated this process over 30 times and calculated maximum standard deviation of only 8.036 for t_s and 15.95 for r. These small deviations indicate that benign hypercall activities exhibit non-repeatability to such a small extent that it has no significant impact on metric values, which we repeatedly calculate for statistical accuracy (see Section 5.2.2).

Specification of attack injection scenarios We now specify attack injection scenarios that we will realize in separate testing phases. We focus on injecting attacks triggering publicly disclosed hypercall vulnerabilities. However, the injection of any malicious hypercall activity using hInjector is possible (e.g., covert channel operations as described in [WDW+14]), in which case an IDS evaluation study would be performed following the same process we demonstrate here.

Scenario #1

We will first evaluate the attack coverage of Xenini when configured such that $th = 0.3$. We will evaluate Xenini's ability to detect attacks triggering the vulnerabilities CVE-2012-5525, CVE-2012-3495, CVE-2012-5513, CVE-2012-5510, CVE-2013-4494, and CVE-2013-1964. We thus demonstrate injecting realistic attacks that conform to the attack models that we presented in Chapter 4 of this thesis. We will inject attacks from the web and mail server VM using the LKM component of hInjector.

Attack contents: In Figure 5.8(a)–(e), we depict the contents of the considered at-

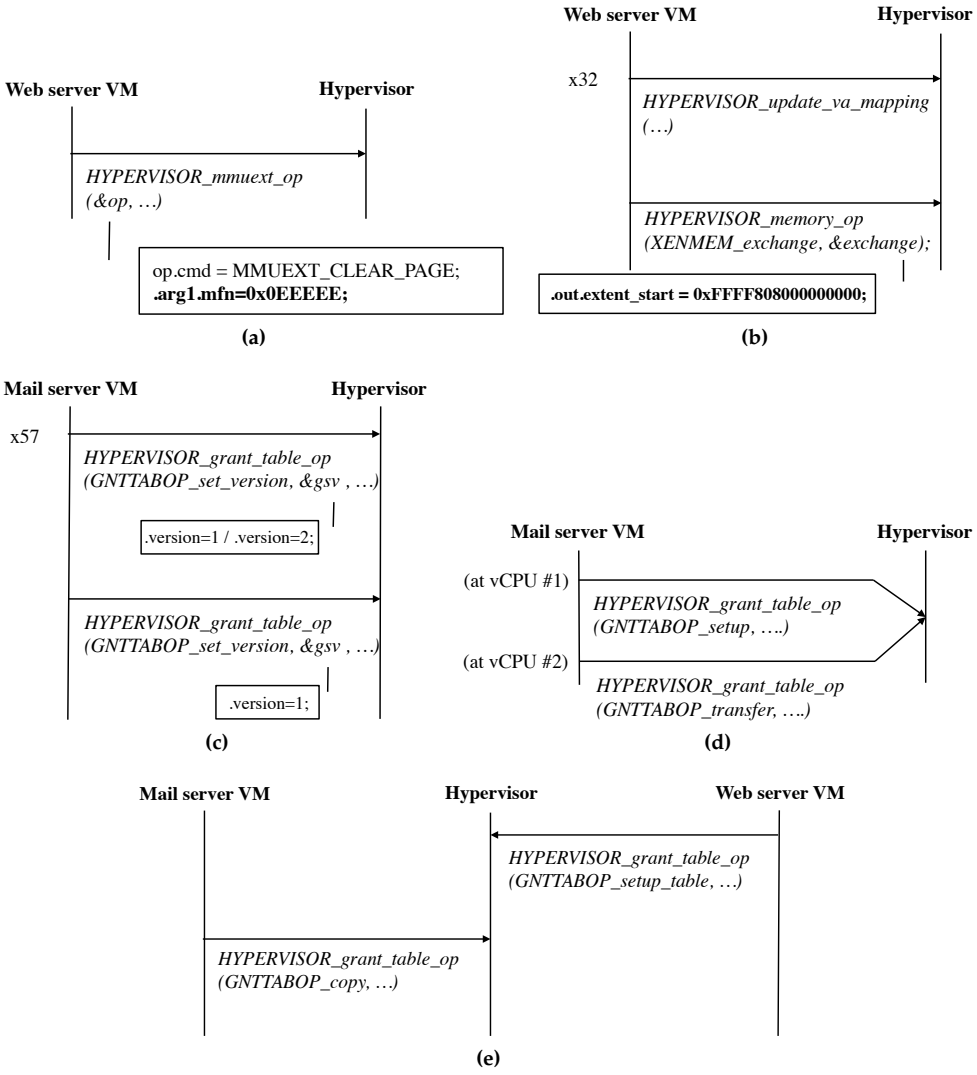


Figure 5.8: Injecting attacks that trigger: (a) CVE-2012-5525, (b) CVE-2012-5513, (c) CVE-2012-5510, (d) CVE-2013-4494 [invoking hypercalls from two vCPUs], (e) CVE-2013-1964 [this vulnerability can also be triggered by invoking hypercalls from one VM].

tacks (the content of the attack triggering CVE-2012-3495 is depicted in Figure 5.4a; we will inject this attack from the web server VM). The semantics of these figures is the same as that of Figure 5.4a — we depict the hypercalls executed as part of an attack and relevant hypercall parameters; that is, integer parameters defining the semantics of the executed hypercalls (e.g., *XENMEM_exchange*), and, where applicable, parameters

with values specifically crafted for triggering a vulnerability, which are marked in bold.

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

Scenario #2

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

Attack contents: The contents of the unmodified attacks and the "smoke-screen" attacks we will inject are depicted in Figure 5.4a and Figure 5.8(b)–(e). To craft "mimicry" attacks, we place each individual hypercall that is part of an attack in the middle of a sequence of 20 injected hypercalls (i.e., at position 10). We built this sequence by starting with the most common detection-relevant property we observed in the planning phase — *iret*, *iret*, *iret*, *iret*. We then added 16 hypercalls such that sliding a window of size 4 over the sequence provides common detection-relevant properties seen during IDS training (i.e., while the hypercall activity of the database server VM has been progressing towards a steady state); we were able to perform this because we calculated the statistic 'number of occurrences of each variation of the detection-relevant property' (see Section 5.2.1). Therefore, we obscure attack patterns making them similar to regular patterns. For example, in Figure 5.9a, we depict the content of the "mimicry" attack triggering CVE-2013-1964.

Attack injection times: We craft "smoke screen" attacks by specifying attack injection times (see Section 5.2.1). We will inject a "smoke screen" attack by delaying for 0.5 seconds the invocation of the hypercalls comprising the attack. Since the average rate of occurrence of the detection-relevant property for the database server VM is 27294.9 occ./sec. (see Table 5.1, column 'Run 1'), we obscure attack patterns by making Xenini analyze approximately 13647 benign occurences of the detection-relevant property before encountering a hypercall that is part of an attack. For example, in Figure 5.9b, we depict the "smoke screen" attack triggering CVE-2013-1964.

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

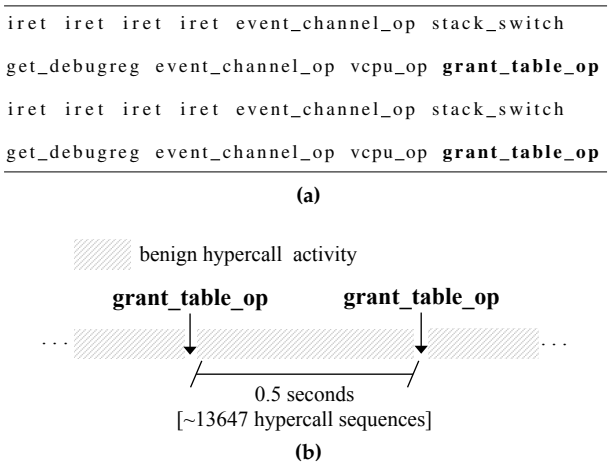


Figure 5.9: Injecting IDS evasive attacks triggering CVE-2013-1964: (a) "mimicry" attack, (b) "smoke screen" attack [the hypercalls triggering CVE-2013-1964 are marked in bold].

5.4.2 Case Study: Testing

We now test Xenini with respect to the scenarios presented in Section 5.4.1.

Scenario #1

IDS Training We deployed and configured Xenini and hInjector. We initalized the IDS monitoring landscape and we trained Xenini until time t_s=5391 seconds. This is the time period needed for the hypercall activities of both the web and mail server VM to reach steady-state (see Table 5.1, column 'Run 1').

Attack injection and calculation of metric values We injected the considered attacks over a period of $t_{max} - ts$ = 109 seconds and then calculated metric values, that is, true and false positive rate. These are calculated as ratios between the number of true, or of false, alerts issued by Xenini, and the total number of injected attacks, or of benign variations of the detection-relevant property occuring during attack injection, respectively. We estimate the latter based on the statistic 'average rate of occurrence of the detection-relevant property'. We repeated the testing phase only 3 times in order to calculate statistically accurate metric values with a relative precision of 2% and 95% confidence level. In addition, we repeated the testing phase over 30 times observing that the obtained metric values negligibly differ from those we present here. This is primarily because of the high repeatability of hypercall activities and it indicates that only a small number of repetitions is needed to calculate statistically accurate metric values.

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

In Table 5.2, we present Xenini's attack detection score. It can be concluded that Xenini exhibited a true positive rate of 0.5 when configured such that $th = 0.3$. We now consider multiple IDS operating points (i.e., IDS configurations which yield given values of the false and true positive rate). In Figure 5.10, we depict a Receiver Operating Characteristic (ROC) curve, which plots operating points for different values of th. We executed separate testing phases to quantify the false and true positive rate exhibited by Xenini for each value of th. We quantified these rates by comparing the output of Xenini with the "ground truth" information recorded by hInjector. We considered the total number of true and false alerts issued by Xenini (i.e, 6 and 6), injected attacks, and occurences of the detection-relevant property during attack injection, originating from both the web and mail server VM. The results depicted in Figure 5.10 match the expected behavior of Xenini (i.e., the lesser the value of th, the more sensitive the IDS, which results in higher true and false positive rates; see [MM11]). This shows the practical usefulness of our approach.

Table 5.2: Detection score of Xenini [✓: detected / x: not detected; $th = 0.3$].

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

We now calculate values of the 'expected cost' metric (C_{exp}) developed by Gaffney and Ulvila [GU01], which expresses the impact of the base rate (see Section 3.3.2). This metric combines ROC curve analysis with cost estimation by associating an estimated cost with each IDS operating point. The measure of cost is relevant in scenarios where a response that may be costly is taken when an IDS issues an alert. Gaffney and Ulvila introduce a cost ratio $C = C_\beta / C_\alpha$, where C_α is the cost of an alert when an intrusion has not occured, and C_β is the cost of not detecting an intrusion when it has occurred. To calculate values of C_{exp}, we set C to 10 (i.e., the cost of not responding to an attack is 10 times higher than the cost of responding to a false alert; see [GU01]).

We estimate the base rate as follows. We have injected 6 attacks consisting of 115 hypercalls over 109 seconds. Further, the average rate of occurence of the detection relevant property originating from the web and mail server VM during attack injection

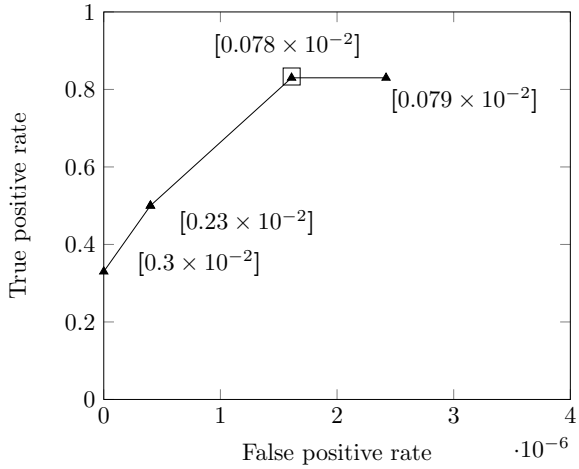


Figure 5.10: Attack detection accuracy of Xenini [th=0.1: $(2.42\times10^{-6}; 0.83)$ • th=0.2: $(1.61\times10^{-6}; 0.83)$ • th=0.3/th=0.4: $(0.4\times10^{-6}; 0.5)$ • th=0.5: $(0; 0.33)$ • □ marks the optimal operating point].

is estimated at $19644.5 + 3141.5 = 22786$ occ./sec. (see Table 5.1, column 'Run 1'). Therefore, the base rate is $\frac{115}{(22786\times109+3)}$=$0.5 \times 10^{-4}$.

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

In Figure 5.10, we depict in square brackets values of C_{exp} associated with each IDS operating point. The 'expected cost' metric enables the identification of an optimal IDS operating point. An IDS operating point is considered optimal if it has the lowest C_{exp} associated with it compared to the other operating points. We mark in Figure 5.10 the optimal operating point of Xenini.

Scenario #2

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

Attack injection and calculation of metric values We injected the unmodified, the "mimicry", and the "smoke screen" attacks as part of three separate testing phases. In Table 5.3, we present the anomaly scores reported by Xenini for the injected attacks. We thus quantify the success of the "mimicry" and "smoke screen" attacks at evading Xenini. Their evasive capabilities are especially evident in the case of the attacks triggering CVE-2012-3495 and CVE-2012-5510. That is, these attacks, when unmodified, can be very easily detected by Xenini (see the high anomaly scores of 1.0 in Table 5.3). However, when transformed into "mimicry" attacks, the detection of these attacks is significantly challenging (see the low anomaly scores of 0.17 and 0.14 in Table 5.3).

Table 5.3: Anomaly scores for the injected non-evasive and evasive attacks.

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

The results presented in Table 5.3 match the expected behavior of Xenini when subjected to evasive attacks (i.e., Xenini reports lower anomaly scores for the evasive attacks than for the unmodified attacks; see [WS02]). This shows the practical usefulness of our approach and the relevance of the observations made in the planning phase, which we used to craft evasive attacks.

5.5 Summary

In this chapter, we presented an approach for the live evaluation of IDSs in virtualized environments using attack injection. We presented hInjector, a tool for generating IDS evaluation workloads that contain virtualization-specific attacks; that is, attacks leveraging or targeting the hypervisor via its hypercall interface — hypercall attacks, which are in the focus of this thesis (see Section 1.1 and Section 1.3). Such workloads are currently not available, which significantly hinders IDS evaluation efforts (see Section 1.2.1). We designed hInjector with respect to three main criteria: injection of realistic attacks, injection during regular system operation, and non-disruptive attack injection. These criteria are crucial for the representative, rigorous, and practically feasible evaluation of IDSs. We demonstrated the application of our approach and showed its practical usefulness by evaluating a representative IDS designed to detect hypercall attacks. We used hInjector to inject attacks that trigger real vulnerabilities as well as IDS evasive attacks.

Chapter 6

Quantifying Attack Detection Accuracy

Any intrusion detection system (IDS) evaluation experiment requires careful planning, which includes selection of metrics and measurement methodologies for quantifying IDS properties (e.g., attack detection accuracy, see Chapter 3). As we mentioned in Section 1.2.2, a common aspect of all existing, conventional metrics for quantifying IDS attack detection accuracy (which we refer to as IDS evaluation metrics) is that they are defined with respect to a *fixed* set of hardware resources available to the IDS under test [MHL+03]. However, a virtualized environment may have *elastic* properties. Under elasticity, we understand on-demand provisioning (i.e., allocation or deallocation) of virtualized hardware resources to virtual machines (VMs). This is also known as vertical VM scaling [SHK+15]. For example, the hypervisor may be configured to hotplug virtualized resources on a VM where the intrusion detection engine of an IDS in virtualized environment, that is, a hypervisor-based IDS or an IDS deployed as virtual network function (VNF), typically operates (see Section 1.1). This may have a significant impact on many properties of the IDS, including its attack detection accuracy. Thus, we argue that using existing IDS evaluation metrics for evaluating IDSs in virtualized environments may lead to inaccurate measurements. We argue that novel metrics and measurement methodologies, which take into account the behavior of an IDS as its operational environment changes, are needed.

This chapter makes the following contributions:

- it reviews conventional IDS evaluation metrics, and demonstrates how elasticity of virtualized environments may impact IDS attack detection accuracy;
- it empirically demonstrates that using conventional IDS evaluation metrics may lead to practically challenging and inaccurate measurements when it comes to evaluating IDSs in virtualized environments featuring elasticity;
- it proposes, and demonstrates the practical use of, a novel metric and measurement methodology that allow for quantifying the impact of elasticity on IDS attack detection accuracy. We designed the metric with respect to a set of criteria for the accurate and practically feasible IDS evaluation. Our metric is meant to complement conventional metrics — it is specifically designed for evaluating IDSs that perform run-time monitoring and are deployed in virtualized environments featuring elasticity. Our metric allows for the accurate assessment of the attack detection accuracy of such IDSs. This enables the identification and deployment of optimally performing IDSs, which significantly reduces the risk

of security breaches in virtualized environments.

The work discussed in this chapter is presented in [MJA+16] and [MJK17].

This chapter is organized as follows: In Section 6.1, we provide an overview of conventional IDS evaluation metrics; in Section 6.2, we discuss and demonstrate the impact of elasticity on IDS attack detection accuracy; in Section 6.3, we present a novel metric and measurement methodology, which take elasticity into account; in Section 6.4, we demonstrate the practical use of the proposed metric and measurement methodology; in Section 6.5, we conclude this chapter.

6.1 Related Work

In this section, for the sake of completeness, we discuss and systematize in a compact manner metrics for quantifying the attack detection accuracy of a given IDS under test (i.e., IDS evaluation metrics). In Chapter 3, Section 3.3, we provided in-depth analysis of the metrics we discuss in this section.

We distinguish between *basic* and *composite* IDS evaluation metrics:

Basic metrics The basic metrics quantify various individual attack detection properties. For instance, the false negative rate $\beta = P(\neg A|I)$ quantifies the probability that an IDS does not generate an alert when an intrusion occurs;[1] therefore, the true positive rate $1 - \beta = 1 - P(\neg A|I) = P(A|I)$ quantifies the probability that an alert generated by an IDS is really an intrusion. The false positive rate $\alpha = P(A|\neg I)$ quantifies the probability that an alert generated by an IDS is not an intrusion, but a regular benign activity; therefore, the true negative rate $1 - \alpha = 1 - P(A|\neg I) = P(\neg A|\neg I)$ quantifies the probability that an IDS does not generate an alert when an intrusion does not occur.

Composite metrics IDS evaluators often combine the basic metrics in order to analyze relationships between them, for example, to identify an optimal IDS operating point — an IDS configuration which yields optimal values of both the true and false positive detection rate — or to compare multiple IDSs.

It is a common practice to use a Receiver Operating Characteristic (ROC) to investigate the relationship between the true and false positive detection rate exhibited an IDS. A ROC curve plots true positive rate against the corresponding false positive rate; that is, a ROC curve depicts multiple IDS operating points of an IDS under test and, as such, it is useful for identifying an optimal operating point or for comparing IDSs. However, ROC curves do not express the impact of the rate of occurence of intrusion events ($B = P(I)$; prior probability that an intrusion event occurs), known as *base rate*, on α and $1 - \beta$. The attack detection performance of an IDS should be assessed with respect to a base rate measure in order for such an assessment to be accurate [Axe00].

[1] P denotes a probability; A denotes an alert event (i.e., an IDS generates an attack alert); I denotes an intrusion event (i.e., an attack is performed).

The error occurring when α and $1 - \beta$ are assessed without taking the base rate into account is known as the base rate fallacy.

Security researchers have proposed metrics that are more expressive than ROC curves. One of the most prominent metrics that belong to this category are the expected cost metric (C_{exp}) proposed by Gaffney and Ulvila [GU01] and the intrusion detection capability metric (C_{ID}) proposed by Gu et al. [GFD+06].

Gaffney and Ulvila [GU01] propose the measure of cost as an IDS evaluation parameter. They combine ROC curve analysis with cost estimation by associating an estimated cost with each IDS operating point. The measure of cost is relevant in scenarios where a response that may be costly is taken (e.g., stopping a network service) when an IDS generates an attack alert. Gaffney and Ulvila introduce a cost ratio $C = C_\beta / C_\alpha$, where C_α is the cost of an IDS alert when an intrusion has not occured, and C_β is the cost of not detecting an intrusion when it has occurred. To calculate the cost ratio, one would need a cost-analysis model that can estimate C_α and C_β.

C_{exp} for a given IDS operating point can be calculated as $C_{exp} = Min(C\beta B, (1 - \alpha)(1 - \beta)) + Min(C(1 - \beta)B, \alpha(1 - B))$. This formula can be obtained by analyzing (i.e., "rolling back") a decision tree whose leaves are costs that may be incurred by an IDS (i.e., C_α and C_β). For more details on the analytical formula of the expected cost metric, we refer the reader to [GU01] and Section 3.3 of this thesis.

Using C_{exp}, one can identify an optimal IDS operating point in a straightforward manner — a given operating point of an IDS is considered optimal if it has the lowest C_{exp} associated with it compared to the other operating points. Further, one can compare IDSs by comparing the estimated costs when each IDS operates at its optimal operating point. The IDS that has lower C_{exp} associated with its optimal operating point is considered as better.

Gu et al. [GFD+06] propose a metric called intrusion detection capability ($C_{ID} = I(X, Y)/H(X)$). They model the input to an IDS as a stream of a random variable X ($X = 1$ denotes an intrusion, $X = 0$ denotes benign activity), and the IDS output respectively as a stream of a random variable Y ($Y = 1$ denotes IDS alert, $Y = 0$ denotes no alert). It is assumed that both the input stream and the output stream have a certain degree of uncertainty reflected by the entropies $H(X)$ and $H(Y)$, respectively. Thus, Gu et al. [GFD+06] model the number of correct guesses of an IDS (i.e., $I(X; Y)$) as mutual shared information between the random variables X and Y ($I(X; Y) = H(X) - H(X|Y)$). The intrusion detection capability metric C_{ID} is obtained by normalizing the shared information $I(X; Y)$ with the entropy of the input variable $H(X)$.

C_{ID} incorporates the uncertainty of the input stream $H(X)$ (i.e., the distribution of intrusions in the IDS input) and the accuracy of an IDS under test $I(X; Y)$. Thus, one may conclude that C_{ID} incorporates the base rate B and many basic metrics, such as the true positive rate $(1 - \beta)$, the false positive rate (α), and similar. For the definition of the relationship between C_{ID}, on the one hand, and B, $1 - \beta$, and α, on the other hand, we refer the reader to [GFD+06]. Given this relationship, a value of C_{ID} may be assigned to any operating point of an IDS on the ROC curve. This allows for a straightforward identification of the optimal operating point of an IDS — the point

that marks the highest C_{ID}. One can compare IDSs by analyzing the maximum C_{ID} of each IDS and considering as better performing the IDS whose optimal operating point has higher CID associated with it.

Summary A common aspect of all conventional IDS evaluation metrics is that they are defined with respect to a fixed set of hardware resources available to the IDS under test. Mell et al. [MHL+03] and Hall et al. [HW02] confirm that the values of existing IDS evaluation metrics express the attack detection accuracy of an IDS only for a specific hardware environment in which the IDS is expected to reside during its lifetime. However, many virtualized infrastructures have elastic properties; that is, resources can be provisioned and used by the IDS on-demand during operation, which may have a significant impact on the attack detection accuracy exhibited by the tested IDS. In Section 6.2, we demonstrate through a case study this impact and we show that existing IDS evaluation metrics express the attack detection accuracy of an IDS only for the specific hardware environment in which the IDS resides.

Based on the above, we argue that the use of conventional IDS evaluation metrics may lead to inaccurate measurements when it comes to evaluating IDSs in virtualized environments. This, in turn, may result in the deployment of misconfigured or ill-performing IDSs, increasing the risk of security breaches. We argue that novel metrics and measurement methodologies are needed. Such metrics and methodologies should take into account the behavior of an IDS under test, deployed in a virtualized environment, as its operational environment changes. As a result, they would allow to quantify the ability of the IDS to scale its attack detection efficiency as resources are allocated to it, or deallocated from it, during operation.

6.2 Elasticity and Accuracy

The elastic behavior of a virtualized environment, that is, the hypervisor applying a resource provisioning policy, may have significant impact on the attack detection accuracy exhibited by the IDS deployed in the environment and performing real-time monitoring. This impact is caused by the hypervisor impacting relevant *transient behaviors* of the IDS. Under relevant transient IDS behaviors, we understand IDS behaviors that are influenced by the amount of resources available to an IDS over time and may impact the IDS's attack detection accuracy. For example, the attack detection accuracy exhibited by a network-based IDS may be correlated to the number of packets dropped by the IDS in the time intervals when attacks are performed. Large amounts of dropped packets in such intervals due to lack of resources may manifest themselves as low IDS attack detection accuracy.

In this section, we demonstrate the impact of the hypervisor, and therefore, of transient IDS behaviors influenced by it (i.e., amount of dropped packets over time), on IDS attack detection accuracy. We demonstrate through this case study the impact of CPU hotplugging considering the case of a network-based IDS deployed as a VNF. We deployed Suricata 2.0.9 in a paravirtualized VM running on top of the Xen 4.4.1 hyper-

visor, and allocated 12 GB of main memory and a network interface card (NIC) with a maximal data transfer rate of 1 Gbit/second to this VM. We replayed over 240 seconds, at the speed of 150 Mbps, a trace file from the Defense Advanced Research Projects Agency (DARPA) datasets.[2] All configuration options of Suricata were set to their default values. For each considered experimental scenario, we repeated measurements 30 times and we averaged the results.

We first considered three separate experimental scenarios, where we hotplug two, three, and four virtual CPUs of 2.6 GHz on the VM where Suricata is deployed. In Table 6.1, we present the true positive rate ($1 - \beta$) measured for each scenario (2/3/4 CPUs in Table 6.1), in relation to the total amount of dropped packets.

We then allocated two virtual CPUs of 2.6 GHz to the VM where Suricata was deployed so that the VM is under CPU pressure when workloads are run. This enabled us to observe the impact of CPU hotplugging on IDS attack detection accuracy in scenarios where such a hotplugging is normally performed. We considered two separate experimental scenarios, where we hotplug one and two additional virtual CPUs on the VM where Suricata is deployed, at the 120th second of each experiment.

In Figure 6.1, we depict the number of packets dropped by Suricata over 240 seconds for each considered CPU hotplugging scenario (2→3/4 CPUs in Figure 6.1). In Table 6.1, section 'CPU hotplugging', we present the true positive rate ($1 - \beta$) measured for each scenario, in relation to the total amount of dropped packets. As expected, $1 - \beta$ increases as more CPUs are hotplugged on the VM where Suricata is deployed. This is due to the decrease of the number of packets dropped by Suricata after CPUs have been hotplugged (see the trend lines in Figure 6.1).

Table 6.1: Attack detection accuracy of Suricata.

Scenario	$1 - \beta$	**No. of true alerts**	**Dropped packets (%)**
2 CPUs	0,393	80184	49,32%
3 CPUs	0,507	103444	33,60%
4 CPUs	0,562	114659	22,77%
		CPU hotplugging	
2 → 3 CPUs	0,465	94906	37,57%
2 → 4 CPUs	0,484	98771	34,46%

In Table 6.1, column 'no. of true alerts', we present the actual number of true alerts issued by Suricata. One can observe that Suricata issued 14722 (i.e., 18.36%) true alerts more when one additional CPU (2→3 CPUs in Table 6.1) was provisioned in comparison to when only two CPUs (2 CPUs in Table 6.1) were made available to the IDS over 240 seconds. Further, Suricata issued 3865 true alerts more when two (2→4

[2]The trace file we replayed is available at `https://www.ll.mit.edu/ideval/data/1998/testing/week1/monday/tcpdump.gz`. The base rate B is 0.025.

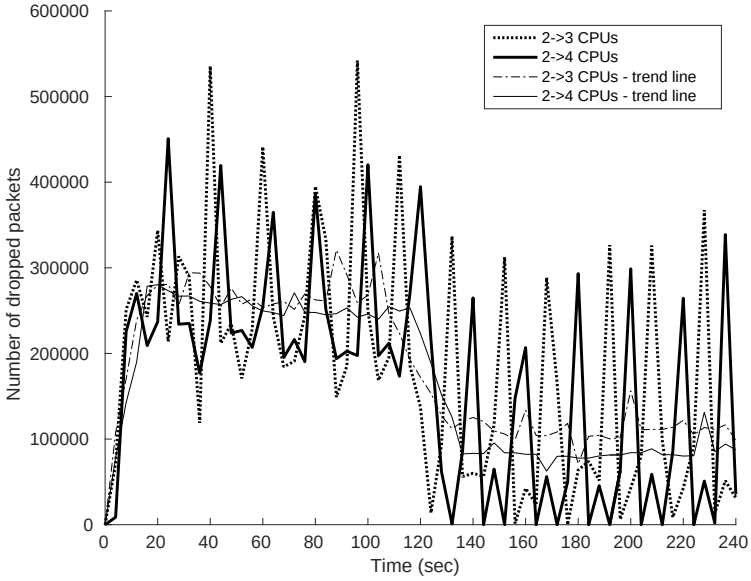


Figure 6.1: Number of packets dropped over time.

CPUs in Table 6.1), instead of one (2→3 CPUs in Table 6.1), additional CPUs were provisioned. This effectively demonstrates the impact of the hypervisor on Suricata's attack detection accuracy.

In a scenario such as the above, where an IDS evaluator aims to understand the relation between a given transient behavior of an IDS and the attack detection accuracy the IDS exhibits, the use of conventional IDS evaluation metrics (see Section 6.1) introduces the following inter-related issues:

◦ Challenging metric value correlation: The IDS evaluator would have to correlate values of metrics belonging to two categories: *(i)* metrics that quantify attack detection accuracy (e.g., true positive rate), and *(ii)* metrics that quantify the considered transient IDS behavior (e.g., amount of dropped packets over time). However, given the lack of metrics and measurement methodologies designed for this purpose, such a correlation would be approximative, which may lead to inaccurate assessments;

◦ Inaccurate comparisons of IDSs: The approximative nature of the correlation mentioned above rules out accurate comparisons of the attack detection accuracy of multiple IDSs. Note that comparing IDSs is a common goal of IDS evaluation studies [MVK+15], where precise measurements of considered metric values are crucial so that the comparisons are accurate and fair.

The metric and measurement methodology we propose in this chapter aim to address the above issues.

6.3 Metric and Measurement Methodology

In this section, we propose a novel metric and measurement methodology, which enable the accurate measurement of the attack detection accuracy of an IDS that performs run-time monitoring and is deployed in a virtualized environment featuring elasticity; that is, they enable the evaluation of the attack detection accuracy exhibited by such an IDS with respect to the impact that on-demand resource provisioning performed by the underlying hypervisor has on the accuracy.

We name the metric we propose *hypervisor factor (HF)*, since it quantifies the impact of the hypervisor as a factor impacting IDS attack detection accuracy. Quantifying this impact calls for a novel definition of the boundaries of a system under test (SUT) in the area of IDS evaluation. The precise definition of the boundaries of an SUT is critical for the accurate measurement of system performance and interpretation of evaluation results. In contrast to the conventional understanding in the area of IDS evaluation about what comprises an SUT (i.e., the IDS under test), we advocate a novel SUT with *extended* boundaries including the hypervisor as well. In Figure 6.2, we depict the boundaries of the conventional SUT in the area of IDS evaluation and of the novel SUT we propose considering a network-based IDS deployed as a VNF.

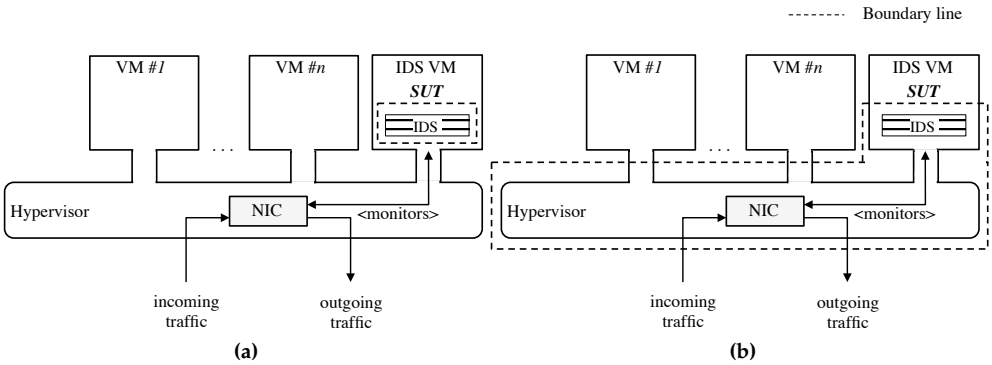


Figure 6.2: Boundaries of: (a) the conventional SUT, and (b) novel SUT in the area of IDS evaluation.

6.3.1 Metric Design

We distinguish three states in which a given IDS, part of an SUT as we define it, may be over the duration of an IDS evaluation experiment: baseline, underprovisioned, and overprovisioned state. By baseline IDS state, we mean a state of the IDS in which it is provisioned by the hypervisor with the minimum amount of resources such that provisioning more resources does not have an impact on the attack detection accuracy of the IDS (e.g., it does not improve the positive rate exhibited by the IDS, see Section 6.2). Therefore, by overprovisioned, or underprovisioned, IDS state, we mean a state of the

IDS in which it is provisioned by the hypervisor with more, or less, resources than the amount needed for the IDS to be considered in baseline state. Given these definitions of IDS states, we design the HF metric with respect to the following criteria, which are crucial for the accurate and practically useful IDS evaluation:

Criterion C_1: If configured accordingly, the HF metric penalizes resource overprovisioning with respect to the

(a) time the IDS has spent in overprovisioned state over the duration of an IDS evaluation experiment, and

(b) the false positive and false negative rate exhibited by the IDS under test when in overprovisioned state, since provisioning excess amount of resources has not contributed towards improving the accuracy of the IDS. We design the HF metric to penalize equally various extents of overprovisioning since we consider any extent of overprovisioning an equally negative phenomenon;

Criterion C_2: If configured accordingly, the HF metric penalizes resource underprovisioning with respect to the

(a) time the IDS has spent in underprovisioned state over the duration of an IDS evaluation experiment, and

(b) the extent of the impact that the underprovisioning has had on the true positive rate exhibited by the IDS. We consider this impact a negative phenomenon since it causes the reduction of the number of true alerts issued by the IDS (see Section 6.2). The HF metric does not penalize resource underprovisioning that has had no impact on the true positive rate since we consider resource saving, which does not cause reduction of this rate, a positive phenomenon.

Criterion C_3: If configured accordingly, the HF metric rewards resource underprovisioning with respect to the

(a) time the IDS has spent in underprovisioned state over the duration of an IDS evaluation experiment, and

(b) the extent of the impact that the underprovisioning has had on the false positive rate exhibited by the IDS. We consider this impact a positive phenomenon since it brings practical benefits —- reduced number of issued false alerts and increased amount of saved resources. Underprovisioning may cause the reduction of the false positive rate exhibited by an IDS if a given amount of workload units (e.g., packets), which would have been falsely labeled as malicious by the IDS if processed by it, are not processed by the IDS due to lack of resources.

In summary, the HF metric favors the most an SUT configured in a way such that the

hypervisor saves the most resources while impacting the true positive rate exhibited by the IDS to the least extent and the false positive rate exhibited by the IDS to the biggest extent.

Criterion C_4: The HF metric expresses the base rate. The attack detection performance of an IDS should be assessed with respect to a base rate measure in order for such an assessment to be accurate (see Section 6.1). Therefore, it is important that the HF metric expresses this rate.

Criterion C_5: The HF metric enables the straightforward identification of optimal operating points. In the context of IDS evaluation, an optimal operating point is an IDS configuration which yields values of both the true and false positive rates considered optimal with respect to a given measure (e.g., cost, see Section 6.1). In the context of this chapter, under optimal operating point, we understand a configuration of both the IDS under test *and* the underlying hypervisor, which yield values of metrics quantifying the performance of the hypervisor at provisioning resources and of metrics quantifying IDS attack detection accuracy (e.g., true and false positive rate) considered optimal with respect to the impact of the former on the latter (see criterion C_1, C_2, and C_3). This is because we consider a novel SUT with boundaries that include an IDS and a hypervisor provisioning the IDS with resources (see Figure 6.2b).

We design the HF metric to enable a straightforward identification of optimal operating points; that is, for a given set of operating points, the optimal operating point yields an extreme value of HF. In Section 6.3.3, we discuss more on operating points and on identifying optimal operating points.

Criterion C_6: The HF metric enables the accurate comparison of multiple SUTs. This is feasible only if criterion C_5 is fulfilled, a topic that we discuss more in Section 6.3.3.

6.3.2 Metric Construction

We present here the main principles of construction for the HF metric. Similar to Gaffney et al. [GU01], we construct the HF metric using a construct from decision theory — a decision tree — as a basis. In Figure 6.3, we depict the decision tree that we use for constructing the HF metric. The tree shows the sequence of uncertain events (circles) that describe:

◦ the *workload*, say $W[B]$, to which the IDS is subjected over the duration of a given IDS evaluation experiment, say T_{max}. We characterize W by the base rate (i.e., probability of an intrusion $B = P(I)$, see Section 6.1);

◦ the *operation* of the IDS processing workload $W[B]$. The operation of the IDS is characterized by the probabilities of the IDS issuing or not issuing an alert when an intrusion has or has not occurred (i.e., the probabilities $P(A|I)$, $P(\neg A|I)$, and so on, see Section 6.1);

◦ the *state* of the IDS (i.e., baseline, overprovisioned, or underprovisioned IDS state, see Section 6.3.1) when it issues or does not issue an alert. The IDS being in one of the considered states during operation primarily depends on the resource provisioning policy applied by the underlying hypervisor, say $H[T_o, T_b, T_u]$; that is, on its precision at meeting the demand for resources by the IDS over time T_{max}. We characterize H by the amount of time the IDS has spent in overprovisioned (T_o), baseline (T_b), and underprovisioned (T_u) state over time T_{max} (i.e., $T_{o/b/u} \in [0; T_{max}], T_o + Tb + Tu = T_{max}$).

Associated with each uncertain event is the probability of occurrence. There are six probabilities specified in the tree: $p_1 = P(I) = B$: the probability that an intrusion occurs; $p_2 = P(A|I) = 1 - \beta$: the probability that the IDS issues an alert when an intrusion occurs (i.e., the true positive rate); $p_3 = P(A|\neg I) = \alpha$: the probability that the IDS issues an alert when an intrusion does not occur (i.e., the false positive rate); and $p_{4/5/6} = \frac{T_{o/b/u}}{T_{max}}$: the probability that the IDS under test is in overprovisioned/baseline/underprovisioned state when it issues or does not issue an alert (i.e., at any moment in the time interval $[0; T_{max}]$).

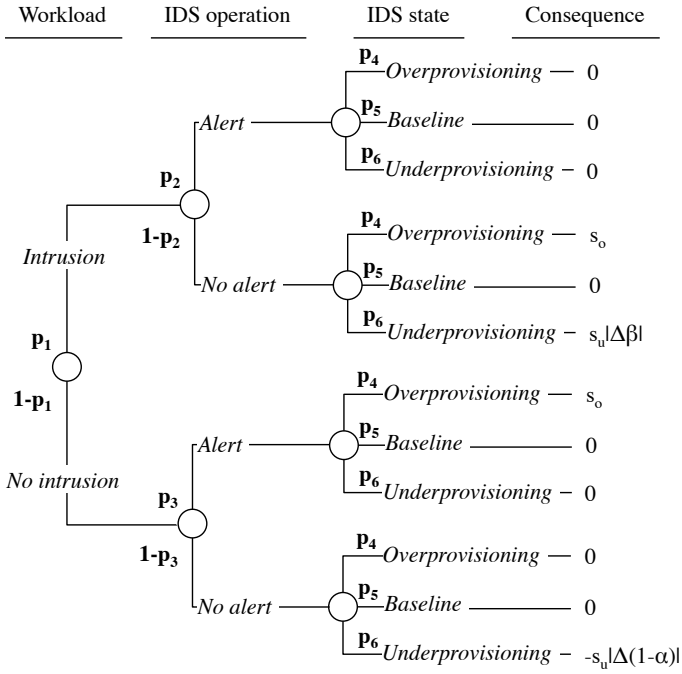


Figure 6.3: The decision tree used for constructing the HF metric.

The attractiveness of each combination of events represented in the tree depicted in Figure 6.3 is characterized by the *consequence* (i.e., the penalty or the reward score) associated with it. With respect to the metric design criteria C_1, C_2, and C_3 (see Sec-

tion 6.3.1), the HF metric:

◦ penalizes the SUT for the IDS issuing false positive or false negative alerts when the IDS is in overprovisioned state. A user of the HF metric may disable or enable this penalization by setting the value of $s_o, s_o \in \{0, 1\}$ to 0 or 1, respectively;

◦ penalizes the SUT for the IDS (in underprovisioned state) not issuing an alert when an intrusion has occurred with the score $s_u|\Delta\beta|$, where $|\Delta\beta| = |\beta - \beta_b|$. A user of the HF metric may disable or enable this penalization by setting the value of $s_u, s_u \in \{0, 1\}$ to 0 or 1, respectively. β_b is the false negative rate exhibited by the IDS in a scenario where it has operated in baseline state over time T_{max} and subjected to workload $W[B]$. Therefore, the HF metric quantifies the impact of underprovisioning on the true positive rate $(1 - \beta)$ exhibited by the IDS – it penalizes the SUT for the IDS not issuing a true alert because of discarded workloads due to lack of resources;

◦ rewards the SUT for the IDS (in underprovisioned state) not issuing an alert when an intrusion has not occurred with the score $s_u|\Delta(1-\alpha)|$, $|\Delta(1-\alpha)| = |(1-\alpha)-(1-\alpha)_b|$. A user of the HF metric may disable or enable this rewarding by setting the value of $s_u, s_u \in \{0, 1\}$ to 0 or 1, respectively. $(1 - \alpha)_b$ is the true negative rate exhibited by the IDS in a scenario where it has operated in baseline state over time T_{max} and subjected to workload $W[B]$. Therefore, the HF metric quantifies the impact of underprovisioning on the false positive rate (α) exhibited by the IDS – it rewards the SUT for the IDS not issuing a false alert.

The formula of the HF metric can be obtained by "rolling back" the decision tree depicted in Figure 6.3; that is, from right to left, the penalty, or the reward, score at an event node is the sum of products of probabilities and scores for each branch:

$$\begin{aligned} HF &= B[\beta(\frac{T_o}{T_{max}}s_o + \frac{T_u}{T_{max}}s_u|\Delta\beta|)] \\ &\quad + (1-B)[\alpha\frac{T_o}{T_{max}}s_o - (1-\alpha)\frac{T_u}{T_{max}}s_u|\Delta(1-\alpha)|)] \\ &= \frac{T_o}{T_{max}}s_o[B\beta + (1-B)\alpha] + \frac{T_u}{T_{max}}s_u[B\beta|\Delta\beta| - (1-B)(1-\alpha)|\Delta(1-\alpha)|] \end{aligned} \tag{6.1}$$

If the values of s_o and s_u are set to 1, Equation 6.1 can be alternatively represented as the sum of the two components of the HF metric, that is, HF_o and HF_u, where $HF_o = \frac{T_o}{T_{max}}[B\beta + (1-B)\alpha]$ is the penalty associated with overprovisioning and $HF_u = \frac{T_u}{T_{max}}[B\beta|\Delta\beta| - (1-B)(1-\alpha)|\Delta(1-\alpha)|]$ is the penalty, or reward, associated with underprovisioning. Distinguishing these components of the HF metric allows for separately observing the quantified consequences of the hypervisor over- and/or underprovisioning the IDS in relation to the IDS's attack detection accuracy.

On Baseline IDS State

Calculating values of the HF metric requires calculating T_o, T_u, and T_b, and, in addition, β_b and $(1-\alpha)_b$ (see Equation 6.1). This, in turn, may require extensive experimentation in order to: *(i)* identify the baseline state of the IDS that is part of the SUT; that is, to determine the minimum amount of resources, say R_b, such that provisioning more resources does not have an impact on the attack detection accuracy exhibited by the IDS (see Section 6.3.1); and *(ii)* compare this amount with the amount of resources provisioned by the hypervisor applying a given resource provisioning policy $H[T_o, T_b, T_u]$, say R_p.

The above activities may be practically challenging because they require the use of measurement approaches considering various resource unit and measurement granularities, and determining how R_b changes over time T_{max} with respect to the intensity of the workload to which the IDS is subjected. Therefore, we assume the following simplifications:

◦ R_b is constant over time T_{max} — we consider R_b the minimum amount of resources allocated to the VM where the IDS operates, such that the IDS does not discard workload when the workload is most intensive. This reflects a realistic scenario where resources are provisioned to an IDS considering the peak intensity of the workload that the IDS may process during operation;

◦ R_b and R_p differ with regard to a single measurement unit (e.g., MB of memory) — that is, we assume that the hypervisor allocates and/or deallocates a single type of resource over the duration of an IDS evaluation experiment. This allows for determining the difference between R_b and R_p over time T_{max} in a straightforward and accurate manner.

We plan to address the above simplifications as part of our future work.

6.3.3 Properties of the HF Metric

In this section, we show how the HF metric satisfies each of the design criteria we presented in Section 6.3.1:

Criterion C_1: For a given T_{max}, the value of the HF_o component of the HF metric (see Section 6.3.2) is positively correlated with T_o (i.e., the time the IDS has spent in overprovisioned state over time T_{max}), the false positive rate (α), and the false negative rate (β);

Criterion C_2 **and** C_3: For a given T_{max}, the value of the HF_u component of the HF metric (see Section 6.3.2):

◦ is positively correlated with T_u (i.e., the time the IDS has spent in underprovisioned

state over time T_{max}) and $|\Delta\beta|$, which quantifies the extent of the impact that underprovisioning has had on the false negative rate, and therefore on the complementary true positive rate;

$\circ$ is negatively correlated with T_u and $|\Delta(1-\alpha)|$, which quantifies the extent of the impact that underprovisioning has had on the true negative rate, and therefore on the complementary false positive rate;

Criterion C_4: The HF metric expresses the base rate B (see Equation 6.1);

Criterion C_5: In Definition 6.1, we define an operating point of an SUT (see Figure 6.2b):

Definition 6.1. An operating point of an SUT consisting of an IDS and a hypervisor, say $O(I \rightarrow (\alpha_b, 1-\beta_b); H[T_o, T_b, T_u]) \rightarrow (\alpha, 1-\beta)$, is a configuration I of the IDS, which yields distinct values of α_b and $(1-\beta)_b$, and a configuration of the hypervisor, that is, a configured resource provisioning policy $H[T_o, T_b, T_u]$. These configurations yield values of $1-\beta$ and α (i.e., the true and false positive rate exhibited by the IDS with configuration I in a scenario where the hypervisor applies resource provisioning policy $H[T_o, T_b, T_u]$).

A single value of the HF metric may be associated with a specific configuration of the IDS and of the hypervisor comprising a given SUT (i.e., with each operating point of the SUT considered in a given evaluation study, see Equation 6.1 and Definition 6.1). Given that the HF metric may penalize an SUT, a given operating point of the SUT is considered optimal if it has the lowest value of the HF metric associated with it. Theoretically, there may be more than one operating point having the same lowest value of the HF metric associated with them. In such a scenario, a given operating point may be considered optimal based on subjective criteria. For example, an IDS evaluator may consider optimal the operating point with the highest value of T_b (i.e., the operating point such that the IDS spends at most time in baseline state).

Measuring values of the HF metric and identifying the optimal operating point of an SUT, out of multiple operating points, is performed in practice by executing multiple experiments using a given workload, and varying the configuration of the IDS and/or of the hypervisor between experiments.

Criterion C_6: Multiple SUTs can be compared by comparing their optimal operating points—the SUT with the lowest value of the HF metric associated with its optimal operating point is considered best.

6.4 Case Studies

We demonstrate here the practical use of the HF metric and the measurement methodology we propose. We conducted multiple experiments considering various SUTs and operating points, that is, configurations of the hypervisor (Section 6.4.1) and of the

IDS (Section 6.4.2) comprising an SUT. For each considered experimental scenario, we repeated measurements 30 times and we averaged the results. The transient IDS behavior of interest is amount of packets dropped over time (see Section 6.2).

It is important to emphasize that we focus on demonstrating the practical use of the HF metric and of the proposed measurement methodology, and not on discussing in depth the behavior of the considered SUTs. We specify arbitrary evaluation scenarios with the sole purpose of demonstrating how the proposed metric and measurement methodology may be used in practice for any evaluation and SUT deployment scenario.

6.4.1 Hypervisor Configurations

We structure this section with respect to the different hypervisor configurations we consider; we investigate the impact of two relevant characteristics of CPU and memory on-demand provisioning (i.e., hotplugging): *(i)* hotplugging *intensity* (i.e., amount of hotplugged resources), and *(ii)* hotplugging *speed* (i.e., the hypervisor's speed at provisioning resources with respect to resource demands).

CPU hotplugging intensity and speed We first quantify the impact of CPU hotplugging intensity on IDS attack detecting accuracy. We deployed Suricata 2.0.9 in a paravirtualized VM running on top of the Xen 4.4.1 hypervisor. We allocated 12 GB of main memory and a NIC with a maximal data transfer rate of 1 Gbit/second to the VM where Suricata was deployed. We also allocated two virtual CPUs of 2.6 GHz to this VM so that the VM is under CPU pressure when workloads are run. This enabled us to observe the impact of CPU hotplugging on IDS attack detection accuracy in scenarios where such a hotplugging is normally performed. We replayed over 240 seconds, at the speed of 150 Mbps, a trace file from the DARPA datasets.[2] All configuration options of Suricata were set to their default values.

We considered two separate experimental scenarios, where we hotplug one and two additional virtual CPUs on the VM where Suricata is deployed, at the 120th second of the experiment; that is, we consider two operating points of the SUT, O_1 and O_2, respectively.

We first identified the baseline state of Suricata at 3 CPUs allocated to the VM where the IDS was deployed. This results in $T_o = 0, T_b = 120, T_u = 120$ for operating point O_1, and $T_o = 120, T_b = 0, T_u = 120$ for operating point O_2. We measured α_b of 0.182×10^{-4} and $(1 - \beta)_b$ of 1.0 (see Section 6.3.2). We then calculated values of the HF metric such that we enabled penalization of overprovisioning and penalization, or rewarding, of underprovisioning (i.e., $s_o, s_u = 1$). In Table 6.2, section 'CPU hotplugging intensity', we present the operating points O_1 and O_2, the impact that underprovisioning has had on Suricata's attack detection accuracy, and values of the HF metric associated with each operating point, including values of its HF_o and HF_u components.

Since the value of the HF metric for O_1 (0.195×10^{-3}) is lower than that for O_2 (0.184×10^{-2}), one can conclude that the SUT performs better when configured at the O_1 operating point (see Section 6.3.3). This is because when the SUT was configured

Table 6.2: Hypervisor configurations: Operating points and associated values of the HF metric [$B = 0.025$; operating points are presented in the form $O(I \rightarrow (\alpha_b, 1-\beta_b); H[T_o, T_b, T_u]) \rightarrow (\alpha, 1-\beta)$; α_b and $1-\beta_b$ are the false and true positive rate exhibited by the IDS when it operates in baseline state given a configuration of the IDS I; T_o, T_b, T_u are the times the IDS has spent in overprovisioned, baseline, and underprovisioned state given a configured resource provisioning policy H; α and $1-\beta$ are the false and true positive rate exhibited by the IDS for a configuration of the IDS I and configured resource provisioning policy H (see Definition 6.1)].

Operating point	**Impact**		**Metric values**		
	$\lvert\Delta(1-\alpha)\rvert$	$\lvert\Delta\beta\rvert$	HF_o	HF_u	HF
CPU hotplugging intensity					
$O_1(I \rightarrow (0.182 \times 10^{-4}, 1.0); H[0, 120, 120]) \rightarrow (0.669 \times 10^{-6}, 0.873)$	0.175×10^{-4}	0.126	0	0.195×10^{-3}	0.195×10^{-3}
$O_2(I \rightarrow (0.182 \times 10^{-4}, 1.0); H[120, 0, 120]) \rightarrow (0.669 \times 10^{-6}, 0.873)$	0.175×10^{-4}	0.126	0.17×10^{-2}	0.195×10^{-3}	0.184×10^{-2}
CPU hotplugging speed					
$O_3(I \rightarrow (0.182 \times 10^{-4}, 1.0); H[0, 160, 80]) \rightarrow (0.707 \times 10^{-5}, 0.934)$	0.111×10^{-4}	0.065	0	0.328×10^{-4}	0.328×10^{-4}
$O_4(I \rightarrow (0.182 \times 10^{-4}, 1.0); H[0, 120, 120]) \rightarrow (0.669 \times 10^{-6}, 0.873)$	0.175×10^{-4}	0.126	0	0.195×10^{-3}	0.195×10^{-3}
$O_5(I \rightarrow (0.182 \times 10^{-4}, 1.0); H[0, 80, 160]) \rightarrow (0.573 \times 10^{-6}, 0.841)$	0.176×10^{-4}	0.158	0	0.416×10^{-3}	0.416×10^{-3}
Memory hotplugging intensity					
$O_6(I \rightarrow (0.189 \times 10^{-4}, 1.0); H[0, 120, 120]) \rightarrow (0.119 \times 10^{-4}, 0.973)$	0.697×10^{-5}	0.026	0	0.574×10^{-5}	0.574×10^{-5}
$O_7(I \rightarrow (0.189 \times 10^{-4}, 1.0); H[120, 0, 120]) \rightarrow (0.119 \times 10^{-4}, 0.973)$	0.697×10^{-5}	0.026	0.346×10^{-3}	0.574×10^{-5}	0.369×10^{-3}
Memory hotplugging speed					
$O_8(I \rightarrow (0.189 \times 10^{-4}, 1.0); H[120, 0, 120]) \rightarrow (0.119 \times 10^{-4}, 0.973)$	0.697×10^{-5}	0.026	0.346×10^{-3}	0.574×10^{-5}	0.369×10^{-3}
$O_9(I \rightarrow (0.189 \times 10^{-4}, 1.0); H[80, 0, 160]) \rightarrow (0, 0.785)$	0.189×10^{-4}	0.214	0.18×10^{-2}	0.768×10^{-3}	0.26×10^{-2}

at the O_1 operating point, Suricata spent 120 seconds in baseline state; to the contrary, at the O_2 operating point, Suricata spent 120 seconds in overprovisioned state issuing false positive and false negative alerts and no time in baseline state. This results in lower value of HF_o for O_1 (0) than that for O_2 (0.17×10^{-2}). The values of the HF metric match the expected behavior of the metric (see criterion C_1, Section 6.3.1). This shows the practical usefulness of the metric and measurement methodology we propose.

We now quantify the impact of CPU hotplugging speed on IDS attack detection accuracy. We allocated two virtual CPUs of 2.6 GHz to the VM where Suricata was deployed. We replayed over 240 seconds, at the speed of 150 Mbps, a trace file from the DARPA datasets.[2] We considered three separate experimental scenarios, where we hotplug one additional virtual CPU at the 80th, 120th, and 160th second of the experiment; that is, we consider three operating points of the SUT, O_3, O_4, and O_5, respectively.

The baseline state of Suricata is at 3 CPUs allocated to the VM where the IDS was deployed ($\alpha_b = 0.182 \times 10^{-4}$, $(1-\beta)_b = 1.0$). This results in $T_o = 0, T_b = 160, T_u = 80$ for operating point O_1, $T_o = 0, T_b = 120, T_u = 120$ for operating point O_2, and $T_o = 0, T_b = 80, T_u = 160$ for operating point O_3. We calculated values of the HF metric such that we enabled penalization of overprovisioning and penalization, or rewarding, of underprovisioning (i.e., $s_o, s_u = 1$).

In Table 6.2, section 'CPU hotplugging speed', where we present the operating points O_3, O_4, and O_5, one can observe that the value of the HF metric (i.e., of its HF_u component) increases as the time at which a CPU was provisioned to the VM where Suricata was deployed increases. The later a CPU was provisioned, the more time the IDS has spent in underprovisioned state, and the bigger is the impact of underprovisioning on the true positive rate exhibited by Suricata (see column '$|\Delta\beta|$'). Therefore, the HF metric penalizes the SUT the most when it is configured at the O_5 operating point. This matches the expected behavior of the metric (see criterion C_2, Section 6.3.1).

Memory hotplugging intensity and speed We first quantify the impact of memory hotplugging intensity on IDS attack detecting accuracy. We deployed Snort 2.9.8.0 in a paravirtualized VM running on top of the Xen 4.4.1 hypervisor. We allocated four virtual CPUs of 2.6 GHz and a NIC with a maximal data transfer rate of 1 Gbit/second to the VM where Snort was deployed. We also allocated 1240 MB of main memory to this VM so that the VM is under memory pressure when workloads are run. This enabled us to observe the impact of memory hotplugging on IDS attack detection accuracy in scenarios where such a hotplugging is normally performed. We replayed over 240 seconds, at the speed of 150 Mbps, a trace file from the DARPA datasets.[2] All configuration options of Snort were set to their default values.

We considered two separate experimental scenarios, where we hotplug additional 260 and 560 MB of main memory on the VM where Snort was deployed, at the 120th second of each experiment; that is, we consider two operating points of the SUT, O_6 and O_7, respectively.

We first identified the baseline state of Snort at 1500 MB of memory allocated to the VM where the IDS was deployed. This results in $T_o = 0, T_b = 120, T_u = 120$ for operating point O_6, and $T_o = 120, T_b = 0, T_u = 120$ for operating point O_7. We measured α_b of 0.189×10^{-4} and $(1-\beta)_b$ of 1.0. We then calculated values of the HF metric such that we enabled penalization of overprovisioning and penalization, or rewarding, of underprovisioning (i.e., $s_o, s_u = 1$).

In Table 6.2, section 'Memory hotplugging intensity', we present the operating points O_6 and O_7, the impact that underprovisioning has had on the attack detection accuracy exhibited by Snort, and values of the HF metric associated with each operating point, including values of its HF_o and HF_u components. As expected, the SUT performs better when configured at O_6 since at this point, in contrast to when the SUT was configured at the O_7 operating point, the IDS did not spent time in overprovisioned state issuing false positive and negative alerts (i.e., $HF_o = 0$, see criterion C_1, Section 6.3.1).

We now quantify the impact of memory hotplugging speed on IDS attack detection accuracy. We allocated 1240 MB of memory to the VM where Snort was deployed. We replayed over 240 seconds, at the speed of 150 Mbps, a trace file from the DARPA datasets.[2] We considered two separate experimental scenarios, where we hotplug additional 560 MB of memory on the VM where Snort was deployed, at the 120th and 160th second of the experiment; that is, we consider two operating points of the SUT, O_8 and O_9, respectively.

The baseline state of Snort is at 1500 MB of memory allocated to the VM where the IDS was deployed ($\alpha_b = 0.189 \times 10^{-4}$, $(1-\beta)_b = 1.0$). This results in $T_o = 120, T_b = 0, T_u = 120$ for operating point O_8, and $T_o = 80, T_b = 0, T_u = 160$ for operating point O_9. We calculated values of the HF metric such that we enabled penalization of overprovisioning and penalization, or rewarding, of underprovisioning (i.e., $s_o, s_u = 1$).

In Table 6.2, section 'Memory hotplugging speed', where we present the operating points O_8 and O_9, one can observe that the SUT performs better when configured at the O_8 operating point — the value of the HF metric is lower than that for the O_9 operating point. This is primarily because, at the O_9 operating point, underprovisioning of memory has caused a much more significant reduction of the true positive rate exhibited by Snort ($|\Delta\beta| = 0.214$, $HF_u = 0.768 \times 10^{-3}$) in contrast to when the SUT was configured at the O_8 operating point ($|\Delta\beta| = 0.026$, $HF_u = 0.574 \times 10^{-5}$, see criterion C_2, Section 6.3.1). Note that at the O_9 operating point, underprovisioning has also caused a reduction of issued false positives to 0 (see column '$|\Delta(1-\alpha)|$', $|\Delta(1-\alpha)| = \alpha_b$), which, although considered a positive phenomenon rewarded by the HF metric (see criterion C_3, Section 6.3.1), does not outweigh the previously mentioned penalization of the reduction of the true positive rate.

6.4.2 IDS Configurations

We now consider various IDS configurations for a given hypervisor configuration. Varying configurations of an IDS under test in order to identify the optimal operating

point of the IDS is a common practice (see Section 6.1). We demonstrate here the use of the HF metric for identifying an optimal operating point of an IDS that is part of an SUT as we define it (see Figure 6.2b). In addition, we show how the HF metric complements conventional IDS evaluation metrics, that is, an ROC curve, which is the most commonly used conventional metric.

We deployed Snort 2.9.8.0 in a paravirtualized VM running on top of the Xen 4.4.1 hypervisor. We allocated two virtual CPUs of 2.6 GHz, 12 GB of main memory, and a NIC with a maximal data transfer rate of 1 Gbit/second to the VM where Snort was deployed. We replayed over 240 seconds, at the speed of 150 Mbps, a trace file from the DARPA datasets.[2]

We considered six separate experimental scenarios, where we vary the configuration of Snort; that is, we consider six operating points $O_{1,2,..6}$. We observed that Snort's rule with ID 1417 led to mislabeling many benign Simple Network Management Protocol (SNMP) packets as malicious (see Section 3.4.1). Therefore, we examined the influence of the configuration parameter *threshold* on the attack detection accuracy of Snort. The parameter *threshold* is used for reducing the number of false alerts by suppressing rules that often mislabel benign activities as malicious. A rule may be suppressed such that it is configured to not generate an alert for a specific number of times (specified with the keyword *count*) during a given time interval (specified with the keyword *seconds*).

We considered five configurations of Snort where the rule with ID 1417 was suppressed by setting the value of *count* to 2, 3, 4, 5, and 6, whereas *seconds* was set to 120. We also considered the default configuration of Snort, according to which the signature with ID 1417 is not suppressed. We configured a resource provisioning policy such that the hypervisor provisions 2 additional CPUs at the 120th second of each experiment.

We first identified the baseline states of Snort and calculated values of α_b and $(1-\beta)_b$ for each considered configuration of the IDS. The baseline state of Snort for all configurations was at 3 CPUs allocated to the VM where the IDS was deployed. This results in $T_o = 120, T_b = 0, T_u = 120$ for all operating points. We then calculated values of the HF metric such that we disabled penalization of overprovisioning and enabled penalization, or rewarding, of underprovisioning (i.e., $s_o = 0, s_u = 1$, $HF_o = 0$); that is, the goal of this study is to identify the optimal configuration of Snort for which underprovisioning is most beneficial.

In Table 6.3, we present the operating points $O_{1,2,..6}$, the impact that underprovisioning has had on the attack detection accuracy exhibited by Snort, and values of the HF metric associated with each operating point. One can observe that underprovisioning has been beneficial in all considered scenarios since it has caused significant reductions of the number of false positives issued by Snort (see column '$|\Delta(1-\alpha)|$' and the negative values of the HF metric). Negative values of the HF metric indicate that an SUT is rewarded for underprovisioning that has caused significant reduction of the false positive rate exhibited by the IDS, which outweighs other behaviors of the SUT that the HF metric penalizes (see Section 6.3.1).

Underprovisioning has been most beneficial when the SUT was configured at the O_5

Table 6.3: IDS configurations: Operating points and associated values of the HF metric [see caption of Table 6.2 for a description of the form in which operating points are presented].

IDS configuration [*seconds* = 120]	**Operating point**	**Impact**		**Metric values**
		$\lvert\Delta(1-\alpha)\rvert$	$\lvert\Delta\beta\rvert$	HF
count = 6	$O_1(I \rightarrow (0.08 \times 10^{-2}, 0.333); H[120,0,120]) \rightarrow (0.072 \times 10^{-2}, 0.332)$	0.753×10^{-4}	0.268×10^{-4}	-0.364×10^{-4}
count = 5	$O_2(I \rightarrow (0.11 \times 10^{-2}, 0.416); H[120,0,120]) \rightarrow (0.102 \times 10^{-2}, 0.415)$	0.788×10^{-4}	0.572×10^{-4}	-0.379×10^{-4}
count = 4	$O_3(I \rightarrow (0.13 \times 10^{-2}, 0.5); H[120,0,120]) \rightarrow (0.122 \times 10^{-2}, 0.499)$	0.733×10^{-4}	0.999×10^{-4}	-0.35×10^{-4}
count = 3	$O_4(I \rightarrow (0.17 \times 10^{-2}, 0.624); H[120,0,120]) \rightarrow (0.168 \times 10^{-2}, 0.623)$	0.1×10^{-4}	0.879×10^{-3}	-0.738×10^{-6}
count = 2	$O_5(I \rightarrow (0.24 \times 10^{-2}, 0.833); H[120,0,120]) \rightarrow (0.23 \times 10^{-2}, 0.832)$	0.952×10^{-4}	0.795×10^{-3}	-0.446×10^{-4}
Default configuration	$O_6(I \rightarrow (0.26 \times 10^{-2}, 0.958); H[120,0,120]) \rightarrow (0.251 \times 10^{-2}, 0.957)$	0.805×10^{-4}	0.297×10^{-3}	-0.39×10^{-4}

operating point ($HF = -0.446 \times 10^{-4}$, *count*=2; -0.446×10^{-4} is the smallest value of all values of the HF metric presented in Table 6.3). Therefore, one may conclude that the optimal operating point of Snort, such that underprovisioning is most beneficial, is when the IDS was configured such that *count* is set to 2.

The HF metric and ROC curves In the area of IDS evaluation, it is a common practice to use ROC curves for analyzing the relationship between the true positive ($1 - \beta$) and false positive rate (α) exhibited by an IDS under test (see Section 6.1). The HF metric expresses $1 - \beta$ and α (see Equation 6.1). Therefore, values of the HF metric can be associated with each IDS operating point depicted using an ROC curve. Thus, evaluation of an IDS with respect to values of the HF metric can be combined with ROC curve analysis. This allows for a visual and comprehensive overview of the attack detection accuracy of the IDS and the impact that the underlying hypervisor has had on the accuracy exhibited by the IDS.

In Figure 6.4, we depict an ROC curve expressing the relationship between α and $1-\beta$ exhibited by Snort for the considered configurations of the IDS (see Table 6.3). The ROC curve is annotated with the values of the HF metric (in square brackets) associated with the IDS configuration points. This allows for a visual, straightforward identification of the IDS operating point such that the SUT, which includes the underlying hypervisor, performs optimally (i.e., the IDS operating point with the lowest value of the HF metric associated with it, marked using □ in Figure 6.4). We note that an IDS evaluator, based on subjective criteria, may consider another IDS operating point optimal (e.g., the one at which Snort exhibits the highest true positive rate). However, in such a case, the evaluator would still have insight into the impact of the hypervisor on the attack detection accuracy of the IDS when configured at this operating point because of the value of the HF metric associated with it.

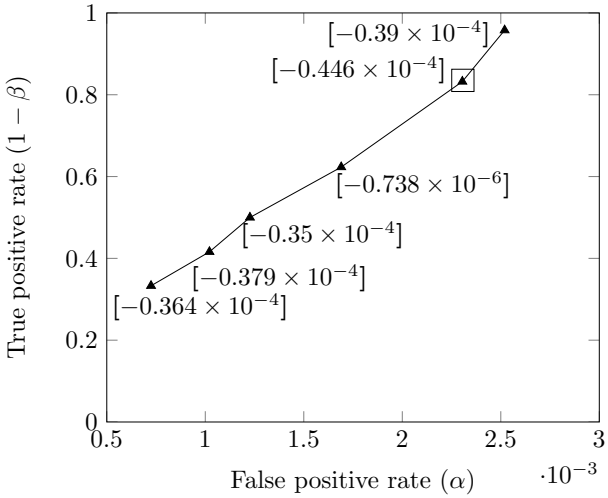


Figure 6.4: An ROC curve and values of the HF metric associated with each IDS operating point.

Multiple ROC curves, each depicting α and $1 - \beta$ exhibited by a single IDS, may be used for comparing multiple IDSs. For example, an IDS may be considered better than the other IDSs if it features higher $1 - \beta$ at all operating points along the ROC curve [MVK+15]. However, such an analysis may be misleading if the ROC curves intersect. In such a case, if the compared IDSs are in virtualized environments featuring elasticity, the IDSs may be compared in a straightforward manner based on values of the HF metric; that is, the IDS that performs optimally considering the impact of the hypervisor on its attack detection accuracy, is the IDS with the lowest value of the HF metric associated with its optimal operating point (see criterion C_6, Section 6.3.3).

6.5 Summary

A virtualized environment may be elastic, that is, virtualized resources may be allocated to, or deallocated from, VMs during operation by the hypervisor applying a given resource provisioning policy. Elasticity might have significant impact on the attack detection accuracy exhibited by an IDS in a virtualized environment featuring elasticity (e.g., an IDS deployed as a VNF). Conventional metrics for quantifying IDS attack detection accuracy (i.e., IDS evaluation metrics) do not express this impact, which might lead to inaccurate measurements.

In this chapter, we first provided an overview of conventional IDS evaluation metrics. We demonstrated the impact that elasticity may have on IDS attack detection accuracy and we showed how the use of conventional IDS evaluation metrics may lead to practically challenging and inaccurate measurements. We then proposed a novel metric (i.e., the HF metric) and measurement methodology, which take elasticity into account. We designed the HF metric with respect to a set of criteria for the accurate and practically useful evaluation. For example, the HF metric penalizes resource underprovisioning causing the reduction of the true positive rate exhibited by a given IDS. The metric and measurement methodology we proposed are meant to complement conventional IDS evaluation metrics and are useful when it comes to assessing the attack detection accuracy of an IDS that is deployed in a virtualized environment featuring elasticity and performs real-time monitoring. We demonstrated the practical use of the HF metric through a set of case studies.

Chapter 7

Conclusions and Outlook

In this chapter, we conclude this thesis with a summary of the contributions presented in the previous chapters. In addition, we discuss future work on improving the security of virtualized environments and on evaluating intrusion detection systems (IDSs) in such environments.

7.1 Summary

Intrusion detection is the process of monitoring the events occurring in a computer or networked system and analyzing those events for signs of possible incidents [SM07]. IDSs — software systems automating the intrusion detection process — are therefore crucial security mechanisms. With the increasing variety and complexity of IDSs, the development of evaluation methodologies, techniques, and tools has become a key research topic that has received a considerable amount of attention [MVK+15]. The benefits of IDS evaluation are manyfold. For instance, one may compare multiple IDSs in terms of their attack detection accuracy in order to deploy an IDS that operates optimally in a given environment, thus reducing the risks of a security breach. Further, one may tune an already deployed IDS by varying its configuration parameters and investigating their influence through evaluation tests. This enables comparison of the evaluation results with respect to the configuration space of the IDS, which can help to identify an optimal configuration.

Given the significant amount of existing practical and theoretical work related to IDS evaluation, a structured classification and a scientifically rigorous analysis is needed to improve the general understanding of the topic and to provide an overview of the current state of the field. Such an overview would be benefitial for identifying and contrasting advantages and disadvantages of different IDS evaluation methods and practices. It would also help identifying requirements and best practices for evaluating current as well as future IDSs. To this end, in Chapter 2, we first provided the background knowledge essential for understanding the topic of intrusion detection and IDS evaluation. We discussed different types of attacks and put intrusion detection into a common context with other security mechanisms. We also defined different types of IDSs. We systematized the latter according to the monitored platform, the attack detection method, the monitoring method, and the deployment architecture. In addition, we demonstrated the wide applicability of IDS evaluation by discussing

its relevance to: *(i)* researchers, who typically perform small-scale evaluation studies to compare novel IDSs with other IDSs in terms of selected IDS properties that are subject of research; *(ii)* industrial software architects, who typically evaluate IDSs by carrying out internationally standardized tests of a large scale; and *(iii)* IT security officers, who evaluate IDSs in order to select an IDS that is optimal for protecting a given environment, or to optimize the configuration of an already deployed IDS. Finally, we provided a historical overview of major developments in the area of IDS evaluation ordering them chronologically.

After providing the essential background knowledge, in Chapter 3, we systematized existing knowledge on IDS evaluation by defining an IDS evaluation design space that puts existing work into a common context. The IDS evaluation design space that we presented is structured into three parts: workload, metrics, and measurement methodology. For each part of the design space, we compared multiple approaches and methods that IDS evaluation practitioners can employ; that is, we provided guidelines for the selection of particular workloads, metrics, and measurement methodologies to use in a given scenario based on the established goals and ability to meet specific requirements. Throughout our discussions on workloads, we identified, and provided links to, commonly used tools, including, for example, workload drivers and trace capturing & replay tools, exploit repositories, and trace repositories. Throughout our discussions on measurement methodologies, we demonstrated how different IDS evaluation approaches are applied in practice. We covered approaches for evaluating the IDS properties attack coverage, resistance to evasion techniques, attack detection accuracy, resource consumption, performance overhead, and workload processing capacity (cf. Chapter 3).

Virtualization allows the creation of logical instances of physical devices. In a virtualized system, governed by a hypervisor, resources are shared among virtual machines (VMs) running on top of the hypervisor. Virtualized environments are becoming increasingly ubiquitous with the growing proliferation of virtualized data centers and cloud environments.

While virtualization provides many benefits, such as reduction of costs through server consolidation, it also introduces some new challenges. More specifically, the introduction of a hypervisor is a critical aspect introducing new threats and vulnerabilities, such as attacks targeting the hypervisor. For example, hypercalls (i.e., software traps from a kernel of a fully or partially paravirtualized VM to the hypervisor) can enable intrusion into vulnerable hypervisors, initiated from a malicious VM kernel, in a procedural manner through hypervisors' hypercall interfaces. The exploitation of a vulnerability of a hypercall handler may lead to altering the hypervisor's memory enabling, for example, the execution of malicious code with hypervisor privileges.

Despite the importance of hypercall vulnerabilities, there is not much publicly available information on them. Publicly disclosed vulnerability reports describing hypercall vulnerabilities (e.g., CVE-2013-4494 [CVEi]) are typically the sole source of information and provide only high-level descriptions. With the goal of increasing the amount of publicly available information on vulnerabilities of hypervisors' hypercall handlers (i.e., hypercall vulnerabilities) and attacks triggering them (i.e., hypercall attacks), in

Chapter 4, we analyzed a set of 35 hypercall vulnerabilities. Our vulnerability analysis approach consisted of analyzing publicly available reports describing the considered vulnerabilities, for example, Common Vulnerabilities and Exposures (CVE) reports and security advisories, reverse-engineering the patches fixing the vulnerabilities, and developing proof-of-concept code, which allowed us to trigger the vulnerabilities and closely observe all stages of the life cycle of a typical hypercall attack. It also allowed us to systematize attackers' activities into attack models. Among other things, models of hypercall attacks facilitate the development of approaches for improving the security of hypercall interfaces where mimicking attackers targeting these interfaces is needed, for example, discovery of vulnerabilities by fuzzing (cf. Chapter 4).

The wide adoption of virtualization has led to the emergence of novel IDSs specifically designed to operate in virtualized environments (i.e., hypervisor-based IDSs, such as Xenini [MM11]). These IDSs have components both inside the hypervisor and in a designated VM. The increased adoption of virtualization has also led to the practice of deploying conventional IDSs as virtual network functions (VNFs). Some of these IDSs have the functionality to detect hypercall attacks, such as Xenini [MM11] and the de-facto standard host-based IDS Open Source Security (OSSEC) [oss]. In the context of this thesis, under hypercall attack, we understand not only attacks triggering hypercall vulnerabilities, but also other malicious hypercall activities, for example, covert channel operations [WDW+14] (see Section 1.3).

Workloads that contain hypercall attacks are crucial for evaluating the attack detection accuracy of IDSs designed to detect hypercall attacks. However, the generation of such workloads is challenging since publicly available scripts that demonstrate hypercall attacks are rare [MPA+14], [HL09] (cf. Chapter 1). An approach towards addressing this issue is attack injection (see Section 1.2.1). In Chapter 5, we presented an approach for the evaluation of IDSs deployed in virtualized environments using attack injection. We presented hInjector, a tool for generating IDS evaluation workloads that contain virtualization-specific attacks (i.e., attacks leveraging or targeting the hypervisor via its hypercall interface — hypercall attacks). Such workloads were previously not available, which significantly hindered IDS evaluation efforts. We designed hInjector with respect to three main criteria: *(i)* injection of realistic attacks with respect to representative attack models, which we presented in Chapter 4; *(ii)* injection of attacks during regular system operation, and *(iii)* injection of attacks in a non-disruptive manner. These criteria are crucial for the representative, rigorous, and practically feasible evaluation of IDSs. We demonstrated the application of our approach and showed its practical usefulness by evaluating a representative IDS designed to detect hypercall attacks. We used hInjector to inject attacks that trigger real vulnerabilities as well as IDS evasive attacks.

A virtualized environment may be elastic, that is, virtualized resources may be allocated to, or deallocated from, VMs during operation by the hypervisor applying a given resource provisioning policy. Elasticity might have significant impact on the attack detection accuracy exhibited by an IDS in a virtualized environment featuring elasticity (i.e., a hypervisor-based IDS or an IDS deployed as a VNF). Conventional metrics for quantifying IDS attack detection accuracy (i.e., IDS evaluation metrics) do

not express this impact, which might lead to inaccurate measurements. To this end, in Chapter 6, we first provided a compact overview of conventional IDS evaluation metrics. Note that we provided an extensive analysis of conventional IDS evaluation metrics in Section 3.3 of Chapter 3. We then demonstrated the impact that elasticity may have on IDS attack detection accuracy and we showed how the use of conventional IDS evaluation metrics may lead to practically challenging and inaccurate measurements. Furthermore, we proposed a novel metric, that is, the hypervisor factor (HF) metric, and measurement methodology that take elasticity into account. We designed the metric with respect to a set of criteria for the accurate and practically feasible evaluation of IDSs. For example, the HF metric penalizes resource underprovisioning causing the reduction of the true positive rate exhibited by a given IDS. We demonstrated the practical use of the HF metric through a set of case studies (cf. Chapter 6).

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

7.2 Outlook

The results presented in this thesis provide the basis for several opportunities for future work, summarized in the sections below. We structure the discussions in this section with respect to the previous chapters of this thesis, where we presented our research contributions (i.e., Chapter 3, Chapter 4, Chapter 5, and Chapter 6). In Section 7.2.5, we discuss system evaluation scenarios that can be studied using the methods presented in Chapter 5 and Chapter 6, in some of which these methods can be applied in combination.

7.2.1 Future Topics in IDS Evaluation

We now discuss relevant future topics in the area of IDS evaluation, which we surveyed in Chapter 3. We focus on open issues and challenges that apply to evaluating novel IDSs, more specifically high-speed IDSs, hypervisor-based IDSs, and IDSs for detecting advanced persistent threats (APTs)/zero-day attacks (see Section 2.1.2).

High-speed IDSs Due to the ever-increasing amount of network traffic, much attention has been given to designing high-speed network-based IDSs (see, for example, [LP13]). These IDSs use workload processing components specifically designed for efficiently processing high-rate workloads (e.g., pattern matchers, see Section 3.4.2). When it comes to designing high-speed network-based IDSs and evaluating their workload processing capacity, current work is biased in favor of designing novel pattern matchers and evaluating their impact on IDS capacity [LL13]. Lin et al. [LL13] profiled the source code of the IDSs Snort [Roe99] and Bro [TBNSM] in order to identify

performance bottlenecks. They discovered that the workload processing components used for detecting current IDS evasive attacks (e.g., packet reassembly mechanisms) are often bigger performance bottlenecks than pattern matchers. This indicates that novel white-box IDS evaluation methods and tests targeting the previously mentioned components should be designed to better understand how these components affect the workload processing capacity of IDSs. This is challenging since it requires in-depth knowledge on the designs of tested IDSs. Further, in contrast to current practice in IDS capacity testing (see Section 3.4.3), the needed tests would involve the generation of workloads that contain modern IDS evasive attacks in order to exercise the targeted workload processing components.

Hypervisor-based IDSs IDSs specifically designed for deployment and operation in virtualized environments (i.e., hypervisor-based IDSs) are becoming common with the growing proliferation of virtualized data centers. Such IDSs are deployed in the virtualization layer, usually with components inside the hypervisor and in a designated VM (see [JXZ+11] and Section 1.1). This enables them to monitor the network and/or host activities of all co-located VMs at the same time. Next, we list key issues related to evaluating hypervisor-based IDSs:

(i) Challenging generation of workload traces: A typical hypervisor-based IDS combines hardware-level information about VMs (e.g., CPU register values) with high-level domain-specific knowledge (e.g., kernel data structures, see [SSG08]) to detect attacks. As a result, we argue that it is a significant challenge to capture workload trace files that contain the exact information required by a hypervisor-based IDS. Given the complexity of recording procedures, it is expected that replay procedures would also be challenging. Thus, a systematic classification structuring the various types of information used by hypervisor-based IDSs would be an important contribution. This can be used as a basis to design configurable recording and replay mechanisms. We discussed this issue in detail in Section 3.5.1.

(ii) Challenging definition of a baseline workload profile: A hypervisor-based IDS monitors the activities of multiple VMs at the same time. In modern data centers, the number of VMs co-located on a hypervisor can vary due to VM migration — a new VM may arrive or an existing one may be removed from a hypervisor due to, for example, load balancing. Thus, one can expect that in a real-world setting, the VMs' activities monitored by a hypervisor-based IDS would change drastically over time. Given this diversity, we argue that it is challenging to define a baseline workload profile that is representative of a "normal" workload for a given virtualized environment, which is needed for IDS training (see Section 2.1.2). A solution would be the development of a model allowing to estimate the characteristics of the VMs' activities monitored by a hypervisor-based IDS for different VM deployment scenarios, and identify baseline workload profiles. We discussed this issue in detail in Section 3.5.1.

IDSs for detecting APTs/zero-day attacks The detection of APTs is becoming an increasingly important topic due to the escalating number of incidents and severity of

APTs. An APT is a carefully executed attack with the objective of intruding a given domain and remaining undetected for an extended period of time. The execution of an APT consists of multiple steps, such as executing a zero-day attack and deploying malware to establish a command & control channel. Therefore, the detection of APTs is performed by composite IDSs consisting of mechanisms for detecting a variety of malicious activities in a coordinated manner, for example, host-, anomaly-based IDSs for detecting zero-day attacks, and network-, misuse-based IDSs for discovering command & control channels. An example of such an IDS is Deep Discovery by TrendMicro [DD].

The rigorous and realistic evaluation of IDSs designed to detect APTs/zero-day attacks is an issue that has not been thoroughly addressed so far. Since mechanisms used for detecting APTs are anomaly- or misuse-based IDSs, their attack detection accuracies can be quantified using the security-related metrics we discussed in Section 3.3 of this thesis. However, the generation of IDS evaluation workloads that contain APTs/zero-day attacks is an open issue. One of the first methodologies for evaluating IDSs designed to detect APTs/zero-day attacks has been published only recently by NSS Labs (for more information on NSS Labs see Section 2.2.2) [NSSb]. This methodology involves the use of high-interaction honeypots for capturing zero-day attacks that can be executed as part of IDS evaluation workloads.

Given the increasing demand for approaches to evaluate IDSs designed to detect APTs/zero-day attacks, the focus of the IDS evaluation community may shift in the near future towards addressing related issues. For instance, of great importance is the design and evaluation of novel honeypots that capture zero-day attacks such that the attacks can be used as IDS evaluation workloads with minimal investment of time. An example is a high-interaction honeypot that specializes in constructing "ground truth" (e.g., a honeypot that can distinguish different attack sessions in order to assign a unique identifier to each session). Current honeypots do not support the automatic construction of "ground truth", which at this time is performed manually by a human expert and therefore is very time-consuming and error-prone (see Section 3.2.6).

The development of approaches for the generation of representative command & control traffic is also important. With the proliferation of cloud environments, attackers have started using cloud services (e.g., Google Apps) as command & control channels in order to evade IDSs — communication with such services is normally part of regular production traffic, and is therefore often considered trusted and not a priority for analysis [TMI]. The development of activity models involving the use of popular cloud services that can be used as a basis for the generation of command & control traffic to record traces (i.e., IDS evaluation workloads in trace form, see Section 3.2.6), would enable the rigorous evaluation of IDSs designed to detect APTs.

7.2.2 Security of Hypervisors' Hypercall Interfaces

We now present an action plan for improving the security of hypervisors' hypercall interfaces, which we analyzed in Chapter 4.

- Tools for fuzzing hypercalls, which can be used for the convenient and time-

efficient discovery of hypercall vulnerabilities, are lacking and should be developed. Such tools would significantly speed up the process of discovering hypercall vulnerabilities, especially those due to implementation errors. Hypercall fuzzing is challenging since, unlike, for example, system calls, many hypercalls perform operations that alter the state not only of the system executing them (i.e., a VM), but also of the underlying hypervisor (see Section 4.3.1).

- Our study revealed that non-implementation errors causing hypercall vulnerabilities are common. As a result, we argue that methods for formally verifying the functional correctness of hypercalls aimed at discovering non-implementation errors causing hypercall vulnerabilities (see Section 4.2.1) should be developed (see Section 4.3.1).
- Secure hypercall programming practices enforcing, for example, value validations of all variables used within a given hypercall handler (i.e., input parameters and internal variables), would help to eliminate missing value validation errors. The application of such practices would lead to the development of secure hypercalls, which, however, may execute more slowly. This poses the challenge of developing hypercall programming practices such that, for example, rigorous and frequent value validations can be performed at a reasonable performance cost.
- Existing security mechanisms, for example, intrusion detection and prevention systems (IDPSs), which take into account hypercall parameter values to detect and/or prevent hypercall attacks, are not effective against hypercall attacks carried out by executing regular hypercalls in a specific way (see Section 4.2.3). Given that many of the current hypercall vulnerabilities can be triggered by executing regular hypercalls in a specific way, we argue that security mechanisms that not only consider hypercall parameter values, but also the way in which hypercalls are executed, should be developed.
- Approaches for generating artificial workloads that contain representative hypercall attacks should be developed since they are crucial for the accurate and rigorous evaluation of IDPSs designed for detecting and preventing hypercall attacks. The development of approaches for generating workloads that contain hypercall attacks is challenged by the lack of publicly available information on hypercall vulnerabilities and attacks (see Section 1.3).

7.2.3 Evaluation of Intrusion Detection Systems Using Attack Injection

In this section, we discuss further scenarios, where the IDS evaluation approach we presented in Chapter 5, can be applied. We also discuss opportunities for future work in the area of evaluation of IDSs designed to detect hypercall attacks.

Besides evaluating typical anomaly-based IDSs, such as Xenini (see Section 5.4), our approach, or hInjector in particular, can be used for:

- evaluating hypercall access control (AC) systems — an example of such a system is Xen Security Modules - Flux Advanced Security Kernel (XSM-FLASK). By evaluating AC systems, we mean verifying AC policies for correctness. This is performed by first executing hypercalls whose execution in hypervisor context should be prohibited and then verifying whether their execution has indeed been prohibited. hInjector can greatly simplify this process since it allows for executing arbitrary hypercall activities and recording relevant information (e.g., information on whether invoked hypercalls have been executed in hypervisor context, see Section 5.3.1);
- evaluating whitelisting IDSs — by whitelisting IDS, we mean IDS that fires an alarm when it observes an activity that has not been whitelisted, either by a user or by the IDS itself while being trained. For example, OSSEC [oss] can be configured to whitelist the hypercall activities it observes during training — our approach involves both rigorous IDS training and execution of arbitrary hypercall activities (see Section 5.2); RandHyp [WCMX12] and Message Authentication Code/Hypercall Access Table (MAC/HAT) [HL09] detect and block the execution of hypercall invocations that originate from untrusted locations (e.g., a loadable kernel module) — hInjector supports the injection of hypercall attacks both from the kernel as well as from a kernel module (see Section 5.3.1).

Our work on evaluating IDSs designed to detect hypercall attacks can be continued in several directions. The integration of VM replay mechanisms in our approach, such as XenTT [Bur13], should be investigated. This may help to further alleviate concerns related to the repeatability of VMs' hypercall activities (see Section 5.2.2). Further, the establishment of a continuous effort on analyzing publicly disclosed hypercall vulnerabilities in order to regularly update hInjector's attack library (see Section 5.3.2) is an important contribution. This is because the lack of up-to-date workloads is a major issue in the field of IDS evaluation. In addition, a wide variety of security mechanisms (see above) that we have not yet evaluated, can be extensively evaluated using our approach. Finally, the application of our approach for injecting attacks involving operations that are functionally similar to hypercalls, such as the input/output control (ioctl) calls of the Kernel-based Virtual Machine (KVM) hypervisor, could be explored. This would allow for evaluating in an accurate and representative manner a broader set of security mechanisms than the one our approach is currently intended for.

7.2.4 Quantifying Attack Detection Accuracy

We now discuss possibilities for future work continuing the work presented in Chapter 6, that is, the HF metric and the measurement methodology we developed.

The measurement methodology we proposed can be extended such that resource provisioning scenarios where the hypervisor allocates or deallocates different types of resources (e.g., both processing and memory resources) are considered when determining the baseline state of an IDS (see Section 6.3.2). This would enable the application of our methodology in any resource provisioning scenario. Further, the HF metric can be extended such that penalization of overprovisioning and rewarding, or penalization,

of underprovisioning can be scaled up or down by a factor configured by the user of the metric. This would allow for further customization of the HF metric and its output with respect to the user requirements. Finally, a set of IDS evaluation experiments using the HF metric and the measurement methodology we proposed, and involving a variety of IDSs and hypervisors applying various resource provisioning policies, may be conducted. Among other scenarios, of particular interest are evaluation scenarios involving IDSs that have adaptive characteristics leading to frequently changing resource requirements, such as Workload-aware Intrusion Detection (WIND) [SJP06].

7.2.5 Future Evaluation Scenarios

In this section, we discuss evaluation scenarios involving existing systems whose attack detection accuracy can be evaluated using the contributions presented in Chapter 5 and Chapter 6 (i.e., the proposed IDS evaluation approach as well as the HF metric and the proposed measurement methodology). We present some of these systems in Table 7.1, which include Bro [TBNSM], Covert Channel (C^2) Detector [WDW+14], OSSEC [oss], and sHype Access Control Mechanism (ACM) [SPB+].

Table 7.1: Systems that can be evaluated using the contributions of this thesis.

	Name	Hypercall	Real-time	IDS type
Intrusion detection systems	Bro [TBNSM]		✓	conventional
	Collabra [BSNS11a]	✓	✓	hypervisor-based
	C^2 Detector [WDW+14]	✓	✓	hypervisor-based
	OSSEC [oss]	✓	✓	conventional
	Snort [Roe99]		✓	conventional
	Suricata [sur]		✓	conventional
	Wizard [SSG08]	✓	✓	hypervisor-based
	Xenini [MM11]	✓	✓	hypervisor-based
Hypercall protection mechanisms	**MAC/HAT** [HL09]	✓	✓	n/a
	RandHyp [WCMX12]	✓	✓	n/a
	sHype ACM [SPB+]	✓	✓	n/a
	XSM-FLASK [XSM]	✓	✓	n/a

The IDS evaluation approach based on hypercall attack injection and the hInjector tool we presented in Chapter 5 are useful for evaluating the attack detection accuracy of IDSs that have the functionality to detect hypercall attacks (column 'hypercall' in Table 7.1). To this category belong primarily hypervisor-based IDSs ('hypervisor-based' in column 'IDS type' in Table 7.1) and some conventional IDSs deployed as VNFs ('conventional' in column 'IDS type' in Table 7.1).

The HF metric and the measurement methodology we presented in Chapter 6 are more general and can be used to test any IDS that performs real-time monitoring (col-

umn 'real-time' in Table 7.1). To this category belong both hypervisor-based IDSs and conventional IDSs deployed as VNFs, such as the de-facto standard IDSs OSSEC [oss] and Snort [Roe99].

For IDSs that have the functionality to detect hypercall attacks and perform real-time monitoring, our approach for injecting hypercall attacks can be applied in combination with the HF metric and the proposed measurement methodology. Such IDSs are marked in bold in Table 7.1.

As we mentioned in Section 7.2.3, besides IDSs, the contributions of this thesis are useful for the evaluation of hypercall protection mechanisms ('hypercall protection mechanisms' in Table 7.1), for example, AC mechanisms. These include the de-facto standard hypercall protection mechanism XSM-FLASK [XSM], which enables mandatory access control of hypercalls based on policies. For instance, our hypercall attack injection approach is useful for verifying XSM-FLASK policies, especially policies of large sizes. For hypercall protection mechanisms that perform real-time monitoring, our approach for injecting hypercall attacks can be applied in combination with our metrics and measurement methodologies. Such mechanisms are marked in bold in Table 7.1.

The wide spectrum of security mechanisms in virtualized environments whose evaluation would benefit from the contributions of this thesis reflects the practical implication of the thesis.

Appendices

Appendix A

Technical Information on Vulnerabilities of Hypercall Handlers

The goal of this chapter is to provide detailed technical information on hypercall vulnerabilities needed for the improvement of the security of hypercall interfaces (see Chapter 4). We focus on the vulnerabilities described in the vulnerability reports CVE-2012-3494 [CVEa], CVE-2012-3495 [CVEb], CVE-2012-3496 [CVEc], CVE-2012-4539 [CVEd], CVE-2012-5510 [CVEe], CVE-2012-5513 [CVEf], CVE-2012-5525 [CVEg], and CVE-2013-1964 [CVEh], which we discussed in Chapter 4 and Chapter 5 of this thesis. These vulnerabilities are representative of the vulnerabilities that we analyzed in terms of the errors causing them and the ways in which they can be triggered.

The vulnerabilities we consider in this chapter are from the Xen hypervisor [BDF+03], which has the most extensive hypercall interface as opposed to other hypervisors, such as the Kernel-based Virtual Machine (KVM) hypervisor [Kiv07]. The considered vulnerabilities are in the handlers of the hypercalls *memory_op*, *gnttab_op*, *set_debugreg*, *physdev_op*, and *mmuext_op*. For each considered vulnerability, we provide background information essential for understanding the vulnerability, and information on the vulnerable hypercall handler (i.e., information about the workflow presented in pseudocode, and input and output data of the handler), and the error causing the vulnerability. We also show how the vulnerability can be triggered and discuss the state of the targeted hypervisor after the vulnerability has been triggered.

We stress that we provide information on a vulnerable hypercall handler to the extent that is relevant for understanding a given vulnerability, for example, we discuss only some input parameters of the handler. We also stress that we do not provide proof-of-concept code for triggering the considered vulnerabilities ready for use. We present only the hypercalls executed as part of an attack triggering a given hypercall vulnerability, and the values of relevant hypercall parameters (i.e., parameters identifying the executed hypercalls and, where applicable, parameters with values specifically crafted for triggering the vulnerability). Finally, we stress that we do not demonstrate vulnerability exploitation where it is possible (e.g., malicious code execution). We focus instead on the errors causing the considered vulnerabilities, the activities for triggering them, and the effects of triggering the vulnerabilities on the state of the vulnerable hypervisors. We argue that the information that we provide is relevant for better understanding the security threats that hypercall interfaces pose, which will

help to focus approaches for improving the security of hypervisors.

The work presented in this chapter has been published in [MVP+14].

A.1 Hypercall memory_op

The *memory_op* hypercall is used for managing the memory of a guest virtual machine (VM), for example, altering the layout of a given memory region. We refer the reader to [xenb] for more information on the functionalities of the *memory_op* hypercall. In the handler of *memory_op*, the different types of memory addresses that the Xen hypervisor supports for abstracting physical memory available to guest VMs are used for accessing locations in memory:

- virtual address - an address of a location in the virtual memory of a guest VM;
- Guest Pseudo-Physical Frame Number (GPFN) - an address of a page frame that is a physical memory address from the perspective of a guest VM;
- Guest Machine Frame Number (GMFN) - an address of a page frame that is a machine address from the perspective of a guest VM;
- Machine Frame Number (MFN) - an address of a page frame that is a real machine address.

For accessing contiguous memory blocks, the different types of addresses mentioned above are used for accessing *extents* of a given *order* such that an extent consists of 2^{order} memory pages.

Mappings between the different types of memory addresses are stored in tables for that purpose. Mappings between virtual addresses and GPFNs are stored in a page table, between GPFNs and GMFNs in a physical-to-machine table, and between GMFNs and GPFNs in a machine-to-physical table.

We refer the reader to [xenc] and [Chi07] for further information on how the Xen hypervisor manages memory.

A.1.1 Vulnerability CVE-2012-3496

"XENMEM_populate_physmap in Xen 4.0, 4.1, and 4.2, and Citrix XenServer 6.0.2 and earlier, when translating paging mode is not used, allows local paravirtualized (PV) OS guest kernels to cause a denial of service (BUG triggered and host crash) via invalid flags such as MEMF_populate_on_demand." [CVEc]

XENMEM_populate_physmap is an operation of the *memory_op* hypercall, which is used for requesting extents from the hypervisor. *XENMEM_populate_physmap* is also used for marking extents as "populate-on-demand". Extents marked as "populate-on-demand" can be assigned to the physical memory of a given guest VM, or removed from it, on demand at run time.

Input:[1] *XENMEM_populate_physmap* takes as input a structure of type *xen_memory_reservation*, which is defined as:

```
struct xen_memory_reservation {
  GUEST_HANDLE(xen_pfn_t) extent_start;
  unsigned int  extent_order;
  unsigned int  address_bits;
  . . .
}
```

extent_start stores the virtual address of the head of an array that contains memory addresses (GPFNs) at which the extents obtained from the hypervisor are to be mapped, or addresses (GPFNs) of the beginnings of the extents that are to be marked as "populate-on-demand"; *extent_order* stores the order of a single extent; *address_bits* stores the flags of the *XENMEM_populate_physmap* hypercall operation, one of which is *MEMF_populate_on_demand*. *MEMF_populate_on_demand* is enabled when *XENMEM_populate_physmap* is used for marking extents as "populate-on-demand".

Output:[1] On success, *XENMEM_populate_physmap* returns the number of the obtained extents or of the extents marked as "populate-on-demand". In case *XENMEM_populate_physmap* has been used for obtaining extents, the array that starts at the virtual address stored in *extent_start* is populated with the memory addresses (MFNs) of the beginnings of the obtained extents. On failure, *XENMEM_populate_physmap* returns an error code (typically a negative integer value).

Workflow of the vulnerable hypercall handler:[1]

```
do_memory_op (XENMEM_populate_physmap, (struct xen_memory_reservation) res)
  . . .
  call populate_physmap(...)
    . . .
      for each GPFN in res.extent_start
        if MEMF_populate_on_demand
          call guest_physmap_mark_populate_on_demand(...)
            call BUG_ON(...)
            . . .
          return
          . . .
        else:
          . . .
    . . .
  return
return
```

[1] As in Xen of version 4.1.0.

Description of the vulnerability: In *guest_physmap_mark_populate_on_demand*, a function invoked in the handler of *XENMEM_populate_physmap*, the BUG_ON macro is used for checking whether the guest VM from where the *memory_op* hypercall has been invoked has the "translated paging" mode disabled. BUG_ON is a macro that crashes the system where it is executed if the condition that it evaluates is true. If *guest_physmap_mark_populate_on_demand* is invoked from a paravirtualized guest VM (note that paravirtualized guest VMs have the "translated paging" mode disabled by default), the condition that the BUG_ON macro evaluates is true and the hypervisor crashes. Thus, CVE-2012-3496 can be triggered by invoking the *XENMEM_populate_physmap* hypercall operation, with the *MEMF_populate_on_demand* flag enabled, from a paravirtualized guest VM.

Vulnerability fix: A patch fixing the vulnerability CVE-2012-3496 was released on 5 September 2012 and is available at [patc]. The patch replaces the *BUG_ON* macro with an *if* clause.

Triggering CVE-2012-3496: We triggered CVE-2012-3496 in the following environment:

- guest VM: OS - Debian Squeeze (64 bit), kernel - 2.6.32-5-amd64;
- host VM: OS - Debian Squeeze (64 bit), kernel - 2.6.32-5-amd64;
- hypervisor: Xen 4.1.0.

The attack that we executed is depicted in Figure A.1.

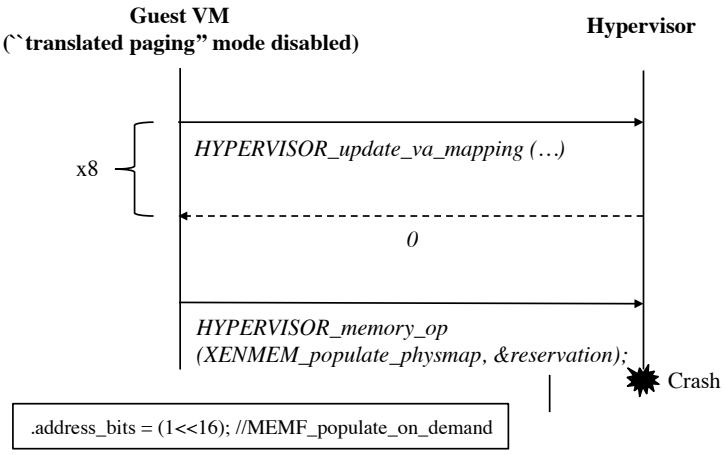


Figure A.1: An attack triggering CVE-2012-3496.

Post-attack state of the hypervisor: The hypervisor crashes when the BUG_ON macro is executed.

A.1.2 Vulnerability CVE-2012-5513

"The XENMEM_exchange handler in Xen 4.2 and earlier does not properly check the memory address, which allows local PV guest OS administrators to cause a denial of service (crash) or possibly gain privileges via unspecified vectors that overwrite memory in the hypervisor reserved range." [CVEf]

XENMEM_exchange is an operation of the *memory_op* hypercall, which is used for modifying the layout of a memory region of a guest VM by "exchanging" extents between the guest VM and the hypervisor. The latter is performed by remapping a set of memory addresses (GPFNs) of beginnings of extents of the guest VM to memory addresses (GMFNs) of beginnings of extents, requested by the guest VM and allocated by the hypervisor for the "exchange" operation. For instance, *XENMEM_exchange* can be used for defragmenting memory such that, for example, 2 extents consisting of 2 pages are exchanged for a single extent consisting of 4 pages.

Input:[2] *XENMEM_exchange* takes as input a structure of type *xen_memory_exchange* defined as:

```
struct xen_memory_exchange {
  struct xen_memory_reservation in;
  struct xen_memory_reservation out;
  xen_ulong_t nr_exchanged;
}
```

, where *xen_memory_reservation* is defined as:

```
struct xen_memory_reservation {
  GUEST_HANDLE(xen_pfn_t) extent_start;
  unsigned int extent_order;
  xen_ulong_t nr_extents;
  ...
}
```

The fields of the *(struct xen_memory_exchange) in* structure store information about the extents that are to be "exchanged". *in.nr_extents* stores the number of extents to be "exchanged"; *in.extent_start* stores the virtual address of the head of an array that contains the memory addresses (GMFNs) of the beginnings of the extents to be "exchanged"; *in.extent_order* stores the order of a single extent.

The fields of the *(struct xen_memory_exchange) out* structure store information about the extents requested from the hypervisor. *out.nr_extents* stores the number of requested extents; *out.extent_order* stores the order of a single requested extent; *out.extent_start* stores the virtual address of the head of an array that consists of GPFNs at which the

[2] As in Xen of version 4.1.0.

requested extents are to be mapped in guest VM's memory.

Output:[2] On success, *XENMEM_exchange* returns 0. The array that starts at the address stored in *(struct xen_memory_exchange) out.extent_start* is populated with the memory addresses (GMFNs) of the beginnings of the extents allocated by the hypervisor for the "exchange" operation. On failure, *XENMEM_exchange* returns an error code (typically a negative integer value).

Workflow of the vulnerable hypercall handler:[2]

---

do_memory_op (XENMEM_exchange, (struct xen_memory_exchange) exch)
  call memory_exchange (XENMEM_exchange, (struct xen_memory_exchange) exch)
    . . .
    allocate extent(s) of $2^{exch.in.order}$ pages
    store the addresses (GMFNs) of the beginnings of the allocated extents in array *mfn*
    . . .
    call __copy_to_guest_offset(...)
        populate memory beginning at *exch.out.extent_start* with the GMFNs in *mfn*
    return
    . . .
  return
return

Description of the vulnerability: The function *__copy_to_guest_offset(to, offset, from, size)*, which is invoked in the handler of the *XENMEM_exchange* hypercall operation, copies data from a virtual address in hypervisor context (*from*) to a virtual address in guest VM context (*to*). For the sake of performance, *__copy_to_guest_offset(to, offset, from, size)* did not perform value validation of the *from* and *to* parameters. As a result, a malicious VM user can invoke *__copy_to_guest_offset(to, offset, from, size)* such that *to* is an address reserved for use by the hypervisor, which leads to overwriting hypervisor's memory. CVE-2012-5513 can be triggered by invoking the *XENMEM_exchange* hypercall operation with an address reserved for use by the hypervisor stored in the *(struct xen_memory_exchange) out.extent_start* parameter.

Vulnerability fix: A patch fixing the vulnerability CVE-2012-5513 was released on 3 December 2012 and is available at [patf]. The patch inserts an invocation of the function *guest_handle_okay* in the handler of the *XENMEM_exchange* hypercall operation, which validates the values of the *from* and *to* parameters of *__copy_to_guest_offset*. For instance, a valid virtual address is an address that is not reserved for use by the hypervisor.

Triggering CVE-2012-5513: We triggered CVE-2012-5513 in the following environment:

- guest VM - OS: Debian Squeeze (64 bit), kernel 2.6.32-5-amd64;
- host VM - OS: Debian Squeeze (64 bit), kernel 2.6.32-5-amd64;
- hypervisor - Xen 4.1.0.

The attack that we executed is depicted in Figure A.2.

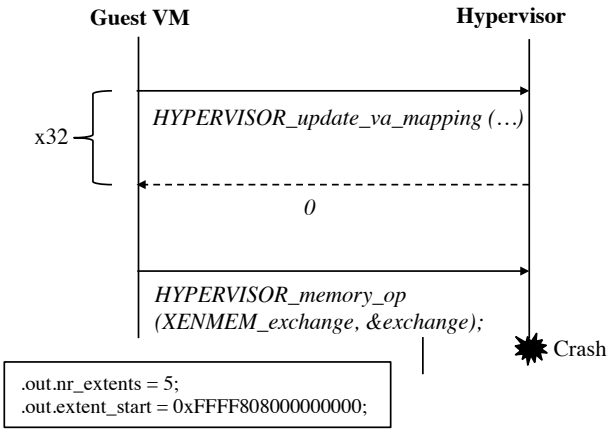


Figure A.2: An attack triggering CVE-2012-5513.

Post-attack state of the hypervisor: When CVE-2012-5513 is triggered, the memory region of the hypervisor beginning at the address stored in *(struct xen_memory_exchange) out.extent_start* is overwritten with the memory addresses (GMFNs) of the beginnings of the extents allocated by the hypervisor for the "exchange" operation. Thus, an attacker cannot control the values with which the hypervisor's memory is overwritten. The amount of data written to the hypervisor's memory is *(struct xen_memory_exchange) out.nr_extents* bytes.

Triggering CVE-2012-5513 may result in a crash of the hypervisor or corrupting its state. Whether the hypervisor crashes depends on which region of the hypervisor's memory is overwritten. An attacker can specify a memory region for overwriting by storing values in the parameters *(struct xen_memory_exchange) out.extent_start* and *(struct xen_memory_exchange) out.nr_extents*. For instance, when we triggered CVE-2012-5513 in our testbed environment, for the values of 0xFFFF808000000000 and 32, and 0xFFFF808000000000 and 16, of *(struct xen_memory_exchange) out.extent_start* and *(struct xen_memory_exchange) out.nr_extents*, respectively, the hypervisor crashed. For the values of 0xFFFF808000000000 and 8 of *(struct xen_memory_exchange) out.extent_start* and *(struct xen_memory_exchange) out.nr_extents*, the hypervisor continued operating with its memory overwritten.

A.2 Hypercall gnttab_op

The *gnttab_op* hypercall is used for managing grant tables. *Grant tables* provide a mechanism for sharing memory between guest VMs (*domains* in Xen terminology) running on top of a Xen hypervisor; that is, it enables the sharing of page frames by granting page frame access permissions to domains or transferring ownerships of

pages between domains. Each domain maintains a grant table, which is shared with the hypervisor. A grant table consists of *grant table entries* (i.e., *grants*) indexed by grant references (i.e., *grefs*). In order to access a page frame for which it needs an access permission, a domain first has to *acquire* the grant that grants the access permission from the domain that has issued the grant. When an acquired grant is not needed anymore, it is *released*.

There are version 1 and version 2 grant tables. The format of a grant table entry of a version 1 grant table is *[gref][domid][frame][flags]*, where *gref* is a grant reference, *domid* is the identification number of domain to which permissions are granted, *frame* is the MFN of the page frame for which permissions are granted, and *flags* are the permissions granted (e.g., read, write, or read and write permissions), which are also referred to as *status* of a grant table entry.

Grant tables of version 2, in addition to grants of the format mentioned above, support transitive grants. *Transitive grants* are used for granting transitive permissions such that a domain issues a grant that refers to a grant issued by another domain.

For the sake of performance, the status of grant table entries of a grant table of version 2 are stored in *status frames*, which are separate from the frames where the rest of the grant table entries are stored.

There are shared and active grants. *Shared grants* are grants issued by a domain. *Active grants* are grants that are in use (i.e., that are acquired) at a given time. A transitive active grant has the fields *trans_domain* and *trans_gref*, where *trans_domain* is the domain that has issued the grant to which the transitive grant refers, and *trans_gref* is the reference of the grant to which the transitive grant refers.

For in-depth information on the grant table mechanism of the Xen hypervisor, we refer the reader to [xenc] and [Chi07].

A.2.1 Vulnerability CVE-2012-4539

"Xen 4.0 through 4.2, when running 32-bit x86 PV guests on 64-bit hypervisors, allows local guest OS administrators to cause a denial of service (infinite loop and hang or crash) via invalid arguments to GNTTABOP_get_status_frames, aka Grant table hypercall infinite loop DoS vulnerability." [CVEd]

GNTTABOP_get_status_frames is an operation of the *grant_table_op* hypercall, which is used for retrieving MFNs of status frames (i.e., status frame MFNs) of a domain.

Input:[3] *GNTTABOP_get_status_frames* takes as input a structure of type *gnttab_get_status_frames* defined as:

```
struct gnttab_get_status_frames {
  uint32_t nr_frames;
```

[3] As in Xen of version 4.1.2.

```
  domid_t dom;
  int16_t status;
  XEN_GUEST_HANDLE(uint64_t) frame_list;
}
```

nr_frames stores the number of requested status frame MFNs; *dom* stores the identification number of the domain whose status frame MFNs are requested; *frame_list* stores the virtual address of the head of an array where status frame MFNs are to be stored upon successful completion of the *GNTTABOP_get_status_frames* operation.

Output:[3] On success, a return code is stored in *(struct gnttab_get_status_frames) status* and the list starting at the address stored in *struct gnttab_get_status_frames) frame_list* is populated with status frame MFNs. On failure, *XENMEM_populate_physmap* returns an error code (typically a negative integer value).

Workflow of the vulnerable hypercall handler:[3]

```
compat_grant_table_op(GNTTABOP_get_status_frames, (struct gnttab_get_status_frames) gf,
      int count = 1)
  rc = 0
  i = 0
  for i < count and rc = 0
    . . .
    if count = 1
        call gnttab_get_status_frames(gf, ...)
            . . .
            if gf.nr_frames > the number of status frames of domain gf.dom
                gf.status = GNTST_general_error
            else
                . . .
                gf.status = GNTST_okay
        return
        if gf.status = GNTST_okay
            increment i to gf.nr_frames
        . . .
return
```

Description of the vulnerability: In the hypercall handler *compat_grant_table_op*, a *for* cycle loops until the value of the variable *i*, which is initialized to 0, is smaller than the value of the input parameter *count*, which has to be 1. In *compat_grant_table_op*, the value of the variable *i* is incremented to the value of the input parameter *(struct gnttab_get_status_frames) nr_frames* only if *(struct gnttab_get_status_frames) status* stores the value of the constant variable *GNTST_okay*. The value of *(struct gnttab_get_status_frames) status* is set in the function *gnttab_get_status_frames*, which is invoked in *compat_grant_table_op*. *gnttab_get_status_frames* sets the value of *(struct gnttab_get_status_frames) status* to the value of *GNTST_okay* only if the value of the input parameter *(struct gnttab_get_sta-*

tus_frames) nr_frames is smaller than the number of status frames of the domain whose identification number is stored in the parameter *(struct gnttab_get_status_frames) dom*.

CVE-2012-4539 can be triggered by invoking *GNTTABOP_get_status_frames* such that the value of the input parameter *(struct gnttab_get_status_frames) nr_frames* is greater than the number of status frames of the domain whose identification number is stored in the parameter *(struct gnttab_get_status_frames) dom*. This results in infinite looping of the *for* cycle in *compat_grant_table_op*.

In order to trigger CVE-2012-4539, one has to set the value of *(struct gnttab_get_status_frames) nr_frames* to a value greater than $\lceil \frac{nr\_grants \times sizeof(uint16\_t)}{PAGE\_SIZE} \rceil$, where *nr_grants* is the number of grants issued by the domain whose identification number is stored in *(struct gnttab_get_status_frames) dom*, *PAGE_SIZE* is the size of a single page of the domain, and *uint16_t* is the size of a variable of type unsigned 16-bit integer. Since the erroneous code is in the handler *compat_grant_table_op*, CVE-2012-4539 can be triggered only from a 64-bit guest VM running on top of a 32-bit host VM.

Vulnerability fix: A patch fixing the vulnerability CVE-2012-4539 was released on 13 November 2012 and is available at [patd]. The patch modifies *compat_grant_table_op* such that the value of *i* is set to 1, which is equal to the value of *count*, if the value of *(struct gnttab_get_status_frames) status* is not equal to the value of *GNTST_okay*. This prevents the *for* cycle in *compat_grant_table_op* from looping indefinitely.

Triggering CVE-2012-4539: We triggered CVE-2012-4539 in the following environment:

- guest VM - OS: Ubuntu Precise (32 bit), kernel 3.8.0-29-generic;
- host VM - OS: Ubuntu Precise (64 bit), kernel 3.8.0-29-generic;
- hypervisor - Xen 4.1.2.

The attack that we executed is depicted in Figure A.3.

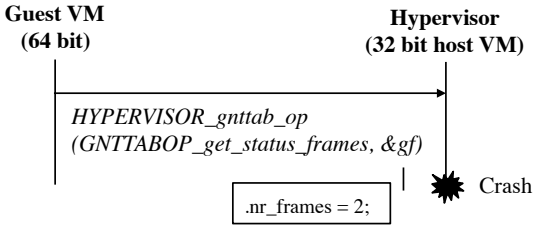


Figure A.3: An attack triggering CVE-2012-4539.

Post-attack state of the hypervisor: When we triggered CVE-2012-4539 in our testbed environment, the guest VM from where we invoked *GNTTABOP_get_status_frames* hanged. When we issued the *xm/xl destroy* command to shutdown the non-responsive guest VM, the hypervisor crashed. The hypervisor did not crash when we issued the

xm/xl shutdown command to shutdown, and the *xm/xl reboot* command to reboot, the non-responsive guest VM.

A.2.2 Vulnerability CVE-2012-5510

"Xen 4.x, when downgrading the grant table version, does not properly remove the status page from the tracking list when freeing the page, which allows local guest OS administrators to cause a denial of service (hypervisor crash) via unspecified vectors." [CVEe]

The *GNTTABOP_set_version* is an operation of the *grant_table_op* hypercall, which is used for downgrading (from version 2 to version 1) or upgrading (from version 1 to version 2) grant tables.

Input:[4] *GNTTABOP_set_version* takes as input a structure of type *gnttab_set_version* defined as:

```
struct gnttab_set_version {
  uint32_t version;
}
```

version stores the version to which the grant table of the domain from where *GNTTABOP_set_version* is invoked is to be set.

Output:[4] On success, *GNTTABOP_set_version* returns 0 and the version of the grant table from where *GNTTABOP_set_version* has been invoked is stored in *(struct gnttab_set_version) version*. On failure, *GNTTABOP_set_version* returns an error code (typically a negative integer value).

Workflow of the vulnerable hypercall handler:[4]

```
do_grant_table_op(GNTTABOP_set_version, ...)
  call gnttab_set_version(...)
    . . .
      if upgrading grant table
        call gnttab_populate_status_frames(...)
            allocate status frames
        return
      if downgrading grant table
        call gnttab_unpopulate_status_frames(...)
            release status frames
        return
    . . .
```

[4] As in Xen of version 4.1.2.

```
  return
return
```

Description of the vulnerability: The function *gnttab_unpopulate_status_frames*, which is invoked in the handler of the *GNTTABOP_set_version* hypercall operation, releases allocated status frames when a grant table is downgraded. However, this function does not fully perform the procedure for releasing status frames; that is, it does not remove the nodes that are associated with the status frames being released from the *xenpage_list* linked list. *xenpage_list* is a list of nodes that contain information about frames allocated from the hypervisor's heap memory space for the needs of a given guest VM.

Since *gnttab_unpopulate_status_frames* does not remove from *xenpage_list* the nodes associated with the status frames, subsequent allocation of the same frames leads to adding nodes to *xenpage_list* that are duplicates of the nodes that have not been removed by *gnttab_unpopulate_status_frames*. This is effectively a corruption of *xenpage_list*. The *gnttab_populate_status_frames* function, which is invoked in the handler of *GNTTABOP_set_version* when a grant table is upgraded, may be used for allocating the same frames that have been released when a grant table has been downgraded.

CVE-2012-5510 can be triggered by continuously allocating and releasing status frames, which eventually leads to corruption of *xenpage_list*; that is, CVE-2012-5510 can be triggered by continuously upgrading and downgrading a grant table. When a corruption of *xenpage_list* occurs depends on the amount of free heap memory of the targeted hypervisor as well as the memory allocating mechanism used.

Vulnerability fix: A patch fixing the vulnerability CVE-2012-5510 was released on 3 December 2012 and is available at [pate]. The patch modifies the function *gnttab_unpopulate_status_frames* such that it inserts an invocation of the function *put_page*. *put_page* removes from *xenpage_list* the nodes associated with the status frames being released when a grant table is downgraded.

Triggering CVE-2012-5510: We triggered CVE-2012-5510 in the following environment:

- guest VM - OS: Ubuntu Precise (32 bit), kernel 3.8.0-29-generic;
- host VM - OS: Ubuntu Precise (32 bit), kernel 3.8.0-29-generic;
- hypervisor - Xen 4.1.2.

The attack that we executed is depicted in Figure A.4.

Post-attack state of the hypervisor: Depending on the use of *xenpage_list* after it has been corrupted, triggering CVE-2012-5510 may result in crash of the targeted hypervisor or may corrupt its state. The hypervisor crashed when we triggered CVE-2012-5510 in our testbed environment.

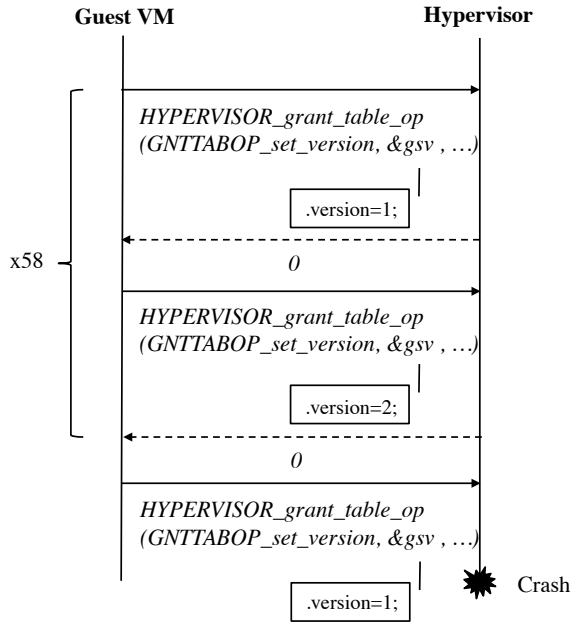


Figure A.4: An attack triggering CVE-2012-5510.

A.2.3 Vulnerability CVE-2013-1964

"Xen 4.0.x and 4.1.x incorrectly releases a grant reference when releasing a non-v1, non-transitive grant, which allows local guest administrators to cause a denial of service (host crash), obtain sensitive information, or possible have other impacts via unspecified vectors." [CVEh]

GNTTABOP_copy is an operation of the *grant_table_op* hypercall, which is used for copying memory pages from a source domain (SD) (i.e., the domain to which the page being copied is allocated) to a destination domain (DD) (i.e., the domain to which the page is copied) with respect to the data read and write permissions set by the SD and/or the DD using grant tables. *GNTTABOP_copy* can be invoked from the SD, the DD, or a domain that is neither the SD or the DD. The domain from where *GNTTABOP_copy* is invoked is called the *local* domain, whereas the other domains involved in copying pages are called *remote* domains.

Input:[5] GNTTABOP_copy takes as input a structure of type *gnttab_copy* defined as:

```
struct gnttab_copy {
```

[5] As in Xen of version 4.1.2.

```
  struct {
    union {
      grant_ref_t ref;
      xen_pfn_t gmfn;
    } u;
    domid_t domid;
    ...
  } source, dest;
  uint16_t len;
  uint16_t flags;
  int16_t status;
}
```

source.u.gmfn stores the GMFN of the page that is to be copied if the SD is a local domain; *dest.u.gmfn* stores the GMFN of the page of the DD to which a page of the SD is to be copied if the DD is a local domain; *source.u.ref* stores the grant reference of the grant that grants access to the page that is to be copied if the SD is a remote domain; *dest.u.ref* stores the grant reference of the grant that grants access to the page of the DD to which a page from the SD is to be copied in case the DD is a remote domain; *(source./dest.)u.domid* stores the identification number of the SD/DD; *len* stores the number of bytes to be copied; *flags* stores a value indicating whether the SD and the DD are local or remote domains.

Output:[5] On success, *GNTTABOP_copy* returns 0. On failure, *GNTTABOP_copy* returns an error code (typically a negative integer value). *(struct gnttab_copy) status* stores a value indicating the status of the page copying operation.

Workflow of the vulnerable hypercall handler:[5]

```
i ← the domain invoking GNTTABOP_copy
d ← the DD

do_grant_table_op(GNTTABOP_copy, struct grant_table_op op, ...)
  call gnttab_copy(op, ...)
    call __gnttab_copy(op, ...)
      ...
      if the DD is remote
        call __acquire_grant_for_copy
          ...
          act = active grant table entry (op.dest.ref)
          ...
          if the grant to be acquired is non−transitive
            ...
            act.trans_domain = i
            act.trans_gref = 0
          ...
        return
```

```
      . . .
      if the DD is remote
        call __release_grant_for_copy(d, op.dest.ref, ...)
          . . .
          act = active grant table entry (op.dest.ref)
          . . .
          if the grant to be released is of version 2
            if act.trans_domain != d
              call __release_grant_for_copy(act.trans_domid, act.trans_gref, ...)
        return
      . . .
    return
  return
return
```

Description of the vulnerability: In the handler of the hypercall operation *GNTTABOP_copy*, the function *__acquire_grant_for_copy* is used for acquiring grants and *__release_grant_for_copy(d, gref, ...)* for releasing grants, where *d* is the domain that has issued the grant to be released and *gref* is the reference of the grant to be released. In case a grant of version 2 is acquired, the hypervisor creates an active grant and sets the values of its fields *trans_domid* and *trans_gref* to the identification number of the domain from where *GNTTABOP_copy* has been invoked and 0, respectively. The reason for the latter is to enable scenarios involving, as described in the source code of the handler of *GNTTABOP_copy*: "grant being issued by one domain, sent to another one, and then transitively granted back to the original domain".

The way in which the scenario mentioned above is supported causes non-transitive grants of version 2 to be released as if they were transitive grants (i.e., in a recursive manner). The culprit of this error is that when releasing a grant in the handler of *GNTTABOP_copy*, it is assumed that a transitive grant is a grant whose *trans_dom* field stores a domain identification number that is not equal to the identification number of the domain that has issued the grant being released. However, since the value of the field *trans_domid* of a non-transitive grant is set to the identification number of the domain from where *GNTTABOP_copy* has been invoked when the grant has been acquired, the previously mentioned condition is also true for non-transitive grants of version 2. As a result, when a non-transitive (active) grant of version 2 is released in the handler of *GNTTABOP_copy*, at least one more grant release takes place, where the grant with a grant reference 0, issued by the domain from where *GNTTABOP_copy* has been invoked, is released.

An attacker can trigger CVE-2013-1964 by invoking *GNTTABOP_copy* such that, for example, a page is copied from a local SD to a remote DD, which has issued a non-transitive grant of version 2.

Vulnerability fix: A patch fixing the vulnerability CVE-2013-1964 was released on 18 April 2013 and is available at [path]. The patch modifies *__acquire_grant_for_copy* such that the value of *trans_domid* is set to the identification number of the domain that

issued the grant that is acquired. Further, the value of *trans_gref* is set to the reference of the grant that is acquired. These modifications of *__acquire_grant_for_copy* prevent the recursive release of non-transitive grant of version 2.

Triggering CVE-2013-1964: We triggered CVE-2013-1964 in the following environment:

- guest VM - OS: Ubuntu Precise (32 bit), kernel 3.8.0-29-generic;
- host VM - OS: Ubuntu Precise (32 bit), kernel 3.8.0-29-generic;
- hypervisor - Xen 4.1.2.

The attack that we executed is depicted in Figure A.5.

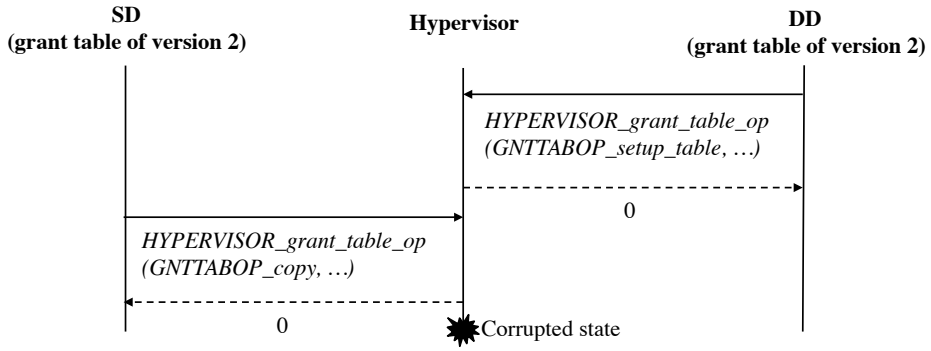


Figure A.5: An attack triggering CVE-2013-1964.

Post-attack state of the hypervisor: Triggering CVE-2013-1964 results in a release of the grant with reference 0 issued by the domain from where *GNTTABOP_copy* is invoked. Triggering CVE-2013-1964 may disrupt the operation of the hypervisor if the grant released due to the triggering of CVE-2013-1964 is in use (i.e., acquired) at the time of execution of the attack. When we triggered CVE-2013-1964 in our testbed environment, the hypervisor continued operating in a corrupted state.

A.3 Hypercall set_debugreg

A.3.1 Vulnerability CVE-2012-3494

"The set_debugreg hypercall in include/asm-x86/debugreg.h in Xen 4.0, 4.1, and 4.2, and Citrix XenServer 6.0.2 and earlier, when running on x86-64 systems, allows local OS guest users to cause a denial of service (host crash) by writing to the reserved bits of the Debug Register (DR) 7 debug control register." [CVEa]

The *set_debugreg* hypercall is used for setting the value of the DR7 register of a CPU allocated to a guest VM. The DR7 register is used for controlling the actions of a CPU when program debugging is performed (e.g., for setting data and/or instruction breakpoints). The addresses at which breakpoints are set in a given debugging session are stored in the registers DR0 – DR3.

The layout of the DR7 register of a 64-bit machine is as follows: ^{bit63} 0 0 0 ... 0 ^{bit31} [LEN3][R/W3] ... [LEN0][R/W0]^{bit15} 0 0 ^{bit13} GD^{bit11} 0 0 ^{bit9} GE LE ^{bit7} [G3][L3] .. [G0][L0]. The upper 32 bits are reserved and should always be cleared. The *LENx* and *R/Wx* fields are used for specifying the length of the monitored data items when a data breakpoint is set (e.g., 00: one-byte length - also when an instruction breakpoint is set, 01: two-byte length) and the type of program execution break set (e.g., 00 - instruction break, 01 - break on data write, 11 - break on data read and write), respectively. The *global exact (GE)* and/or the *local exact (LE)* bits are set when a data breakpoint is set and instruct the CPU to slow down the execution of the program being debugged so that the exact instruction that triggers the data breakpoint can be reported to the debugging program. The *Gx* and *Lx* bits are used for enabling or disabling breakpoints set at the addresses stored in the registers DR0 - DR3.

Input:[6] *set_debugreg* takes as input a number of a register (an integer value, 7 is used for specifying the DR7 register) and a value that is to be stored in the register (an unsigned long integer value).

Output:[6] On success, *set_debugreg* returns 0. On failure, *set_debugreg* returns an error code (typically a negative integer value).

Workflow of the vulnerable hypercall handler:[6]

```
do_set_debugreg (int reg_nr, unsigned long value)
  call set_debugreg(reg_nr, value)
    if reg_nr = 7
      value &= ~DR_CONTROL_RESERVED_ZERO
      . . .
      store value in DR7
  return
return
```

Description of the vulnerability: In the handler of the *set_debugreg* hypercall, the value of the variable *~DR_CONTROL_RESERVED_ZERO* is applied as a mask with the binary bitwise AND operator to the value of the second parameter of *set_debugreg*. The latter is performed so that the upper 32 bits of the value that is to be stored in the DR7 register are cleared. *DR_CONTROL_RESERVED_ZERO*, which stores the value of *0x0000d800ul*, translates to the binary value of ^{bit63}(0...0)^{bit31}(0000) (0000) (0000) (0000) (1101) (1000) (0000) (0000)^{bit0}. The complement form of the previously mentioned binary number is: ^{bit63}(1...1)^{bit31}(1111) (1111) (1111) (1111) (0010) (0111)

[6] As in Xen of version 4.1.2.

(1111) (1111)^{bit0}, which is stored in the variable *~DR_CONTROL_RESERVED_ZERO*. Since they are set to 1, the upper 32 bits of *~DR_CONTROL_RESERVED_ZERO* do not clear the upper 32 bits of the value that is to be stored in the DR7 register when applied as a mask with the binary bitwise AND operator. This results in setting one or multiple bits of the upper 32 bits of the DR7 register to 1, which is not allowed according to hardware specifications.

CVE-2012-3494 can be triggered by invoking *set_debugreg* in a way such that one or multiple bits of the upper 32 bits of the value of the second parameter of *set_debugreg* are set to 1. The bits of the second parameter of *set_debugreg* that are used for setting data or instruction breakpoints (e.g., the bits of the *LENx* fields) should store binary values for setting an instruction breakpoint.

Vulnerability fix: A patch fixing the vulnerability CVE-2012-3494 was released on 5 September 2012 and is available at [pata]. The patch assigns the value of *~0xffff27fful* to *DR_CONTROL_RESERVED_ZERO*, and thus, the variable *~DR_CONTROL_RESERVED_ZERO*, which is applied as a mask to the value of the second parameter of *set_debugreg*, has the binary value of ^{bit63}(0...0)^{bit31}(0000) (0000) (0000) (0000) (0010) (0111) (1111) (1111)^{bit0}. Since the upper 32 bits of *~DR_CONTROL_RESERVED_ZERO* are cleared, applying *~DR_CONTROL_RESERVED_ZERO* as a mask to the value of the second parameter of *set_debugreg* with the binary bitwise AND operator clears the upper 32 bits of the parameter.

Triggering CVE-2012-3494: We triggered CVE-2012-3494 in the following environment:

- guest VM - OS: Ubuntu Precise (32 bit), kernel 3.8.0-29-generic;
- host VM - OS: Ubuntu Precise (32 bit), kernel 3.8.0-29-generic;
- hypervisor - Xen 4.1.2.

The attack that we executed is depicted in Figure A.6.

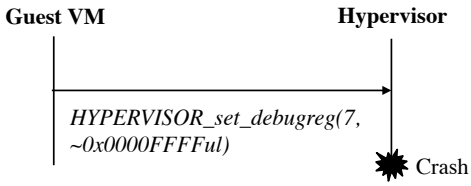


Figure A.6: An attack triggering CVE-2012-3494.

Post-attack state of the hypervisor: Given that in current systems the upper 32 bits of the DR7 register are reserved and should be cleared, triggering CVE-2012-3494 results in crash of the vulnerable hypervisor. However, an outcome different than crash of the hypervisor may be possible if a vulnerable hypervisor is run on future hardware, as stated in [CVEa]: "if the vulnerable hypervisor is run on future hardware, the impact

of the vulnerability might be widened depending on the future assignment of the currently-reserved debug register bits."

A.4 Hypercall physdev_op

A.4.1 Vulnerability CVE-2012-3495

"The physdev_get_free_pirq hypercall in arch/x86/physdev.c in Xen 4.1.x and Citrix XenServer 6.0.2 and earlier uses the return value of the get_free_pirq function as an array index without checking that the return value indicates an error, which allows guest OS users to cause a denial of service (invalid memory write and host crash) and possibly gain privileges via unspecified vectors." [CVEb]

PHYSDEVOP_get_free_pirq is an operation of the *physdev_op* hypercall, which is used for allocating Peripheral Component Interconnect Interrupt Requests (PIRQs) for the needs of a given guest VM. The Xen hypervisor maintains an array called *pirq_irq* for each guest VM that it hosts. *pirq_irq* is used for marking a given PIRQ as allocated such that the value of the constant variable PIRQ_ALLOCATED (i.e., -1) is stored in the node of *pirq_irq* of index equal to the allocated PIRQ.

Input:[7] *PHYSDEVOP_get_free_pirq* takes as input structure of type *physdev_get_free_pirq* defined as:

```
struct physdev_get_free_pirq {
  int type;
  uint32_t pirq;
}
```

type stores the type of the PIRQ to be allocated (i.e., *MAP_PIRQ_TYPE_GSI* or *MAP_PIRQ_TYPE_MSI*).

Output:[7] On success, *PHYSDEVOP_get_free_pirq* returns 0 and the allocated PIRQ is stored in *(struct physdev_get_free_pirq) pirq*. On failure, -28 is stored in *(struct physdev_get_free_pirq) pirq* and *XENMEM_populate_physmap* returns an error code (typically a negative integer value).

Workflow of the vulnerable hypercall handler:[7]

```
do_physdev_op (PHYSDEVOP_get_free_pirq, (struct physdev_get_free_pirq) gfp)
  . . .
  call gfp.pirq = get_free_pirq(...)
```

[7] As in Xen of version 4.1.2.

```
    allocate a PIRQ
  return
  pirq_irq[gfp.pirq] = PIRQ_ALLOCATED
  ...
return
```

Description of the vulnerability: In the handler of the *PHYSDEVOP_get_free_pirq* hypercall operation, the function *get_free_pirq* is invoked for allocating a PIRQ. The return value of *get_free_pirq* is the allocated PIRQ, if a PIRQ has been succesfully allocated, or an error code (i.e., -28) if a PIRQ could not be allocated. The return value of *get_free_pirq* is used as an index to access an element of the array *pirq_irq* for marking a PIRQ as allocated. However, the return value of *get_free_pirq* is not checked whether it is a PIRQ or an error code. In case *get_free_pirq* returns an error code, the error code is used as an array index and as a result the value of the constant variable *PIRQ_ALLOCATED* (i.e., -1) is written at the memory address &*pirq_irq* - 28, which is a location in hypervisor's memory.

CVE-2012-3495 can be triggered by attempting to allocate a PIRQ when there are no available PIRQs. This can be achieved by invoking the hypercall operation *PHYSDEVOP_get_free_pirq* multiple times until all available PIRQs are allocated and an attempt is made to allocate a PIRQ when there are no available PIRQs. Since PIRQs that can be allocated to a given VM are in the range of 16 to the value of the variable *nr_pirqs_gsi*, a variable in hypervisor context that stores the largest PIRQ that can be allocated to a given VM, invoking *PHYSDEVOP_get_free_pirq* ((*nr_pirqs_gsi* - 16) + 2) times is sufficient for triggering CVE-2012-3495.

Vulnerability fix: A patch fixing the vulnerability CVE-2012-3495 was released on 5 September 2012 and is available at [patb]. The patch inserts an *if* clause that checks whether the return value of the *get_free_pirq* function is a PIRQ. If *get_free_pirq* returns an error code, the error code is not used as an index for accessing an element of the *pirq_irq* array.

Triggering CVE-2012-3495: We triggered CVE-2012-3495 in the following environment:

- guest VM - OS: Ubuntu Precise (32 bit), kernel 3.8.0-29-generic;
- host VM - OS: Ubuntu Precise (32 bit), kernel 3.8.0-29-generic;
- hypervisor - Xen 4.1.2.

The attack that we executed is depicted in Figure A.7.

Post-attack state of the hypervisor: Triggering CVE-2012-3495 results in overwriting the hypervisor's memory at the memory address &*pirq_irq* - 28 with the value of the variable *PIRQ_ALLOCATED* (i.e., -1). An attacker cannot control the value written in the hypervisor's memory. Depending on the memory layout of the hypervisor, the hypervisor may crash or continue operating in a corrupted state.

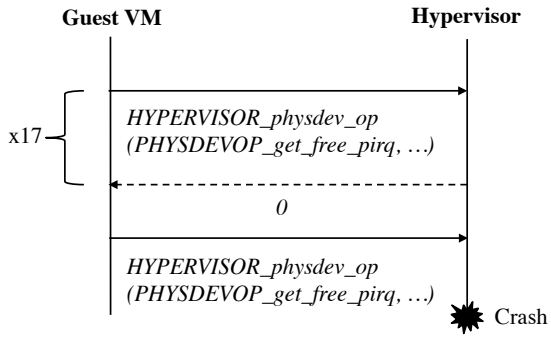


Figure A.7: An attack triggering CVE-2012-3495.

A.5 Hypercall mmuext_op

A.5.1 Vulnerability CVE-2012-5525

"The *get_page_from_gfn* hypercall function in Xen 4.2 allows local PV guest OS administrators to cause a denial of service (crash) via a crafted GFN that triggers a buffer over-read." [CVEg]

The *get_page_from_gfn* function provides information about a given memory page. It is invoked in the handlers of multiple hypercalls of the Xen hypervisor, one of which is the handler of the *MMUEXT_CLEAR_PAGE* operation of the *mmuext_op* hypercall. *MMUEXT_CLEAR_PAGE* is an operation of the *mmuext_op* hypercall, which is used for clearing memory pages/frames.

Input:[8] *MMUEXT_CLEAR_PAGE* takes as input structure of type *mmuext_op* defined as:

```
struct mmuext_op {
  unsigned int cmd;
  union {
  xen_pfn_t mfn;
  . . .
  } arg1;
  . . .
}
```

cmd stores a number identifying an operation of the *mmuext_op* hypercall (e.g., *MMUEXT_CLEAR_PAGE*); *arg1.mfn* stores the MFN of the page that is to be cleared.

[8] As in Xen of version 4.2.0.

Output:[8] On success, *MMUEXT_CLEAR_PAGE* returns 0. On failure, it returns an error code (typically a negative integer value).

Workflow of the vulnerable hypercall handler:[8]

```
do_mmuext_op ((struct mmuext_op) op, ...)
  struct page_info page;
  call page = get_page_from_gfn(op.arg1.mfn, ...)
  . . .
return
```

Description of the vulnerability: *get_page_from_gfn* reads information about a page allocated to a guest VM from the frame table of the VM using the MFN of the page as offset. A frame table is a memory area shared between the hypervisor and a guest VM where information about each page allocated to the guest VM is stored in the format of a structure of type *page_info*.

The MFN used by *get_page_from_gfn* for reading page information is provided to *get_page_from_gfn* as an input parameter. The value of the MFN provided as input parameter to *get_page_from_gfn* is not checked for validity. Since *get_page_from_gfn* uses a MFN as an offset for reading from the frame table of a given guest VM, an invalid MFN is a MFN that causes a buffer over-read (i.e., that is larger than the largest MFN at which a page of the guest VM is allocated). An attacker can provide an invalid MFN as an input parameter to *get_page_from_gfn*, in which case *get_page_from_gfn* returns invalid page information.

In the handler of the *MMUEXT_CLEAR_PAGE* hypercall operation, the MFN stored in the input parameter *(struct mmuext_op) arg1.mfn* is provided to *get_page_from_gfn* for reading page information. CVE-2012-5525 can be triggered by invoking *MMUEXT_CLEAR_PAGE* such that an invalid MFN is stored in *(struct mmuext_op) arg1.mfn*.

Vulnerability fix: A patch fixing the vulnerability CVE-2012-5525 was released on 3 December 2012 and is available at [patg]. The patch inserts an invocation of the function *mfn_valid* in *get_page_from_gfn*, which verifies the validity of the MFN provided as input to *get_page_from_gfn*. The patch modifies *get_page_from_gfn* such that if the MFN used for reading page information is not valid, *get_page_from_gfn* returns NULL instead of invalid page information.

Triggering CVE-2012-5525: We triggered CVE-2012-5525 in the following environment:

- guest VM - OS: Ubuntu Precise (32 bit), kernel 3.8.0-29-generic;
- host VM - OS: Ubuntu Precise (32 bit), kernel 3.8.0-29-generic;
- hypervisor - Xen 4.2.0.

The attack that we executed is depicted in Figure A.8.

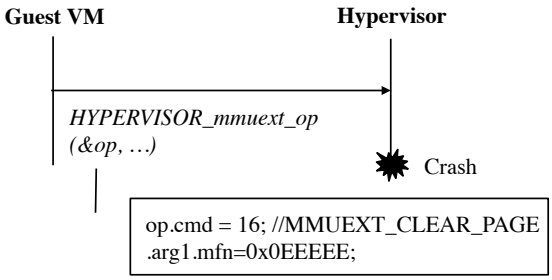


Figure A.8: An attack triggering CVE-2012-5525.

Post-attack state of the hypervisor: Triggering CVE-2012-5525 may result in a crash of the targeted hypervisor or may corrupt its state. Whether the hypervisor crashes depends on the use of the invalid page information returned from *get_page_from_gfn* when CVE-2012-5525 is triggered. The hypervisor crashed when we triggered CVE-2012-5525 in our testbed environment.

List of Figures

List of Tables

Acronyms

AC access control.
ACM Access Control Mechanism.
ACPS Advanced Cloud Protection System.
AMI Amazon Machine Image.
APT advanced persistent threat.

C^2 Covert Channel.
CAIDA Cooperative Association for Internet Data Analysis.
CR Control Register.
CVE Common Vulnerabilities and Exposures.

DARPA Defense Advanced Research Projects Agency.
DD destination domain.
DEFCON Defense Readiness Condition.
DES Data Encryption Standard.
DesVM designated VM.
DoS denial-of-service.
DR Debug Register.
DVM destination VM.

EAX Extended Accumulator Register.
EC2 Amazon Elastic Compute Cloud.

GE global exact.
GMFN Guest Machine Frame Number.
GPFN Guest Pseudo-Physical Frame Number.

HF hypervisor factor.
hid hypercall identification.
HTTP Hyper Text Transfer Protocol.

I/O input/output.

IBM International Business Machines Corporation.
ID identification number.
IDPS intrusion detection and prevention system.
IDS intrusion detection system.
IMAP Internet Message Access Protocol.
ioctl input/output control.
IP Internet Protocol.
ISA Internet Security and Acceleration.
ITA Internet Traffic Archive.

J2EE Java 2 Enterprise Edition.

KDD Knowledge Discovery and Data Mining.
KVM Kernel-based Virtual Machine.

LAN local area network.
LARIAT Lincoln Adaptable Real-time Information Assurance Testbed.
LBNL/ISCI Lawrence Berkeley National Laboratory/International Computer Science Institute.
LE local exact.
LKM loadable kernel module.

MAC/HAT Message Authentication Code/Hypercall Access Table.
MAWI Measurement and Analysis on the WIDE Internet.
MFN Machine Frame Number.
MIT Massachusetts Institute of Technology.
MVM malicious VM.

NIC network interface card.
NIST National Institute of Standards and Technology.
NPV negative predictive value.

OSEC Open Security Evaluation Criteria.
OSSEC Open Source Security.
OSVDB Open Sourced Vulnerability Database.

PIRQ Peripheral Component Interconnect Interrupt Request.
PPV positive predictive value.
PV paravirtualized.

ROC Receiver Operating Characteristic.
RSA Rivest Shamir Adleman.

SD source domain.
SNMP Simple Network Management Protocol.
SPEC Standard Performance Evaluation Corporation.
SQL Standard Query Language.
SUT system under test.
SVM source VM.

TCP/IP Transport Control Protocol/Internet Protocol.

UC University of California.

VAIT Vulnerability and Attack Injector Tool.
vCPU virtual CPU.
VM virtual machine.
VMI virtual machine introspection.
VNF virtual network function.

WIDE Widely Integrated Distributed Environment.
WIND Workload-aware Intrusion Detection.
WINE Worldwide Intelligence Network Environment.

XML Extensible Markup Language.
XSM-FLASK Xen Security Modules - Flux Advanced Security Kernel.

ZRC zero reference curve.

Bibliography

[AAA+10] Faeiz Alserhani, Monis Akhlaq, Irfan U. Awan, Andrea J. Cullen, and Pravin Mirchandani. MARS: Multi-stage Attack Recognition System. In *Proceedings of the 24th IEEE International Conference on Advanced Information Networking and Applications (AINA)*, pages 753–759, Washington, DC, USA, 2010. IEEE Computer Society. [see pages 5 and 73]

[aAHSBT] ab Apache HTTP Server Benchmarking Tool. `http://httpd.apache.org/docs/2.2/programs/ab.html`. [see page 28]

[AAL+03] Nicholas Athanasiades, Al Abler, John Levine, Henry Owen, and George Riley. Intrusion detection testing and benchmarking methodologies. In *Proceedings of First IEEE International Workshop on Information Assurance*, pages 63–72, 2003. [see page 25]

[Abh15] Abhik Chaudhuri, Heberto Ferrer, Hemma Prafullchandra, JD Sherry, Kelvin Ng, Xiaoyu Ge, Yao Sing and Yiak Por (Main Contributors). Aleksandar Milenkoski (Minor Contributor). Best Practices for Mitigating Risks in Virtualized Environments. Cloud Security Alliance - Virtualization Working Group, April 2015. [see pages xii and 1]

[ABYSS10] Miriam Allalouf, Muli Ben-Yehuda, Julian Satran, and Itai Segall. Block storage listener for detecting file-level intrusions. In *26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, May 2010. [see pages 28, 30, and 72]

[AHPP10] Eyad Alkassar, Mark A. Hillebrand, Wolfgang Paul, and Elena Petrova. Automated Verification of a Small Hypervisor. In *Verified Software: Theories, Tools, Experiments*, volume 6217, pages 40–54. Springer Berlin Heidelberg, 2010. [see page 91]

[ALD11] Abdulbasit Ahmed, Alexei Lisitsa, and Clare Dixon. A Misuse-based Network Intrusion Detection System using Temporal Logic and Stream Processing. In *5th International Conference on Network and System Security (NSS)*, pages 1–8, 2011. [see page 52]

[Ale16] Aleksandar Milenkoski, Bernd Jaeger, Kapil Raina, Mason Harris, Saif Chaudhry, Sivadon Chasiri, Veronica David, and Wenmao Liu. Security Position Paper: Network Function Virtualization. Cloud Security Alliance - Virtualization Working Group, March 2016. [see pages xi and 2]

[ama] Amazon Elastic Computing Cloud: Virtualization Types. `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/virtualization_types.html`. [see page 99]

[AMIA] Amazon Machine Images (AMIs). `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html`. [see page 69]

[ASS11] Saurabh Amin, Galina A. Schwartz, and Shankar S. Sastry. On the interdependence of reliability and security in Networked Control Systems. In *50th IEEE Conference on Decision and Control and European Control Conference (CDC-ECC)*, pages 4078–4083, Dec 2011. [see page 86]

[ATJ+10] Alberto Avritzer, Rajanikanth Tanikella, Kiran James, Robert G. Cole, and Elaine J. Weyuker. Monitoring for security intrusion using performance signatures. In *WOSP/SIPEW*, pages 93–104, 2010. [see pages 19, 26, and 52]

[Axe00] Stefan Axelsson. The base-rate fallacy and its implications for the difficulty of intrusion detection. *ACM Transactions on Information and Systems Security*, 3(3):186–205, August 2000. [see pages 44, 103, and 120]

[BBCL11] Gilles Barthe, Gustavo Betarte, Juan D. Campo, and Carlos Luna. Formally Verifying Isolation and Availability in an Idealized Model of Virtualization. In *FM 2011: Formal Methods*, volume 6664, pages 231–245. Springer Berlin Heidelberg, 2011. [see page 91]

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. *SIGOPS Operating Systems Review*, 37(5):164–177, October 2003. [see pages 78 and 153]

[BSNS11a] Saketh Bharadwaja, Weiqing Sun, Mohammed Niamat, and Fangyang Shen. Collabra: A Xen Hypervisor Based Collaborative Intrusion Detection System. In *Proceedings of the Eighth International Conference on Information Technology: New Generations (ITNG)*, pages 695–700, Washington, DC, USA, 2011. IEEE Computer Society. [see pages 3, 4, 26, and 149]

[BSNS11b] Saketh Bharadwaja, Weiqing Sun, Mohammed Niamat, and Fangyang Shen. Collabra: A Xen Hypervisor Based Collaborative Intrusion Detection System. In *Proceedings of the 2011 Eighth International Conference on Information Technology: New Generations*, pages 695–700. IEEE Computer Society, 2011. [see pages 92, 93, and 99]

[Bug] BugTraq. `http://www.securityfocus.com/archive/1`. [see page 31]

[Bur13] Anton Burtsev. *Deterministic Systems Analysis*. PhD thesis, University of Utah, 2013. [see pages 104 and 148]

[cai] The Cooperative Association for Internet Data Analysis (CAIDA). `http://www.caida.org/data/`. [see pages 5 and 36]

[CAV+15] Diogo Carvalho, Nuno Antunes, Marco Vieira, Aleksandar Milenkoski, and Samuel Kounev. Challenges of Assessing the Hypercall Interface Robustness (Fast Abstract). In *The 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2015)*. IEEE, June 2015. [see page x]

[Chi07] David Chisnall. *The Definitive Guide to the Xen Hypervisor*. Prentice Hall

Press, Upper Saddle River, NJ, USA, first edition, 2007. [see pages 154 and 160]

[CLC+10] Chien-Yi Chiu, Yuh-Jye Lee, Chien-Chung Chang, Wen-Yang Luo, and Hsiu-Chuan Huang. Semi-supervised Learning for False Alarm Reduction. In Petra Perner, editor, *Advances in Data Mining. Applications and Theoretical Aspects*, volume 6171 of *Lecture Notes in Computer Science*, pages 595–605. Springer Berlin Heidelberg, 2010. [see page 46]

[CLF+99] Robert K. Cunningham, R. P. Lippmann, D. J. Fried, S. L. Garfinkel, I. Graf, K. R. Kendall, S. E. Webster, D. Wyschogrod, and M. A. Zissman. Evaluating Intrusion Detection Systems without Attacking your Friends: The 1998 DARPA Intrusion Detection Evaluation. In *SANS*, 1999. [see pages 21, 38, and 74]

[CLLL12] Tsung-Huan Cheng, Ying-Dar Lin, Yuan-Cheng Lai, and Po-Ching Lin. Evasion Techniques: Sneaking through Your Intrusion Detection/Prevention Systems. *IEEE Communications Surveys and Tutorials*, 14(4):1011–1020, 2012. [see page 31]

[CM06] Simon P. Chung and Aloysius K. Mok. On random-inspection-based intrusion detection. In *Proceedings of the 8th international conference on Recent Advances in Intrusion Detection (RAID)*, pages 165–184, 2006. [see pages 28 and 52]

[CPX+13] Chun-Jen Chung, Khatkar Pankaj, Tianyi Xing, Jeongkeun Lee, and Dijiang Huang. NICE: Network Intrusion Detection and Countermeasure Selection in Virtual Network Systems. *IEEE Transactions on Dependable and Secure Computing*, 10(4):198–211, 2013. [see pages 30 and 52]

[CtCtF] Capture the Capture the Flag. `http://cctf.shmoo.com/`. [see pages 5 and 36]

[CVEa] CVE-2012-3494. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-3494`. [see pages 90, 153, 168, and 170]

[CVEb] CVE-2012-3495. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-3495`. [see pages 10, 82, 107, 153, and 171]

[CVEc] CVE-2012-3496. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-3496`. [see pages 86, 153, and 154]

[CVEd] CVE-2012-4539. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-4539`. [see pages 153 and 160]

[CVEe] CVE-2012-5510. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-5510`. [see pages 84, 153, and 163]

[CVEf] CVE-2012-5513. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-5513`. [see pages 10, 83, 153, and 157]

[CVEg] CVE-2012-5525. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-5525`. [see pages 10, 82, 153, and 173]

[CVEh] CVE-2013-1964. `http://web.nvd.nist.gov/view/vuln/detail?`

vulnId=CVE-2013-1964. [see pages 85, 153, and 165]

[CVEi] CVE-2013-4494. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-4494. [see pages 85 and 142]

[CVEj] CVE Details. http://www.cvedetails.com/. [see pages 9, 77, and 78]

[CVEC] Common Vulnerabilities and Exposures (CVE). http://cve.mitre.org/. [see page 31]

[CWM+07] Scott E. Coull, Charles V. Wright, Fabian Monrose, Michael P. Collins, and Michael K. Reiter. Playing Devil's Advocate: Inferring Sensitive Information from Anonymized Network Traces. In *Proceedings of the Network and Distributed System Security Symposium*, pages 35–47, 2007. [see page 35]

[dar] DARPA Intrusion Detection Evaluation: 1998 Testing Data - First Week Truth. http://www.ll.mit.edu/mission/communications/cyber/CSTcorpora/ideval/data/1998/Truth_Week_1.llist.tar.gz. [see page 57]

[DCW+99] Robert Durst, Terrence Champion, Brian Witten, Eric Miller, and Luigi Spagnuolo. Testing and evaluating computer intrusion detection systems. *ACM Communications*, 42(7):53–61, July 1999. [see pages 44, 45, 73, and 74]

[DD] Deep Discovery. http://www.trendmicro.com/us/enterprise/security-risk-management/deep-discovery/. [see page 146]

[DDW99] Hervé Debar, Marc Dacier, and Andreas Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(9):805–822, 1999. [see page 16]

[DDWL98] Hervé Debar, Marc Dacier, Andreas Wespi, and Stefan Lampart. An Experimentation Workbench for Intrusion Detection Systems. Technical report, IBM Research, Zurich Research Laboratory, 1998. [see pages 21, 26, and 72]

[Deh12] Alex Dehnert. Intrusion Detection Using VProbes. *VMware Technical Journal*, 1(2):28–31, 2012. [see pages 52 and 65]

[DFPS08] Holger Dreger, Anja Feldmann, Vern Paxson, and Robin Sommer. Predicting the resource consumption of network intrusion detection systems. In *Proceedings of the 2008 International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 437–438, New York, NY, USA, 2008. ACM. [see page 59]

[DKC+02] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 211–224, New York, NY, USA, 2002. ACM. [see pages 28, 30, and 72]

[Dkf] Dkftpbench. http://www.kegel.com/dkftpbench/. [see page 28]

[DM02] Hervé Debar and Benjamin Morin. Evaluation of the Diagnostic Capabil-

ities of Commercial Intrusion Detection Systems. In *Recent Advances in Intrusion Detection (RAID)*, volume 2516, pages 177–198. Springer Berlin / Heidelberg, 2002. [see page 26]

[DS11] Tudor Dumitras and Darren 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)*, pages 89–96, New York, NY, USA, 2011. ACM. [see pages 21 and 74]

[Dun] George Dunlap. The Paravirtualization Spectrum, Part 2: From poles to a spectrum. `https://blog.xenproject.org/2012/10/31/the-paravirtualization-spectrum-part-2-from-poles-to-a-spectrum/#more-5564`. [see page 98]

[evi16] EvIDencE: Testing Intrusion Detection Systems in Virtualized Environments; EvIDencE: Testen von Systemen zur Angriffserkennung in virtualisierten Umgebungen (orig., ger.), 2016. Awarded by the German Research Foundation; Deutsche Forschungsgemeinschaft (DFG). [see page x]

[exp] Exploit Database by Offensive Security. `http://www.exploit-db.com/`. [see page 4]

[FBAF10] Romain Fontugne, Pierre Borgnat, Patrice Abry, and Kensuke Fukuda. MAWILab: Combining Diverse Anomaly Detectors for Automated Anomaly Labeling and Performance Benchmarking. In *ACM CoNEXT '10*, December 2010. [see pages 23 and 36]

[FHSL96] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A Sense of Self for Unix Processes. In *IEEE Symposium on Security and Privacy*, pages 120–128, May 1996. [see page 109]

[FJGS+00] Y. F. Jou, F. Gong, C. Sargor, X. Wu, S.F. Wu, H. C. Chang, and F. Wang. Design and implementation of a scalable intrusion detection system for the protection of network infrastructure. In *Proceedings of DARPA Information Survivability Conference and Exposition, 2000*, volume 2, pages 69–83 vol.2, 2000. [see pages 30, 52, and 74]

[FM11] Leo Freitas and John McDermott. Formal methods for security in the Xenon hypervisor. *International Journal on Software Tools for Technology Transfer*, 13(5):463–489, 2011. [see page 91]

[Fos07] James C. Foster. *Metasploit Toolkit for Penetration Testing, Exploit Development, and Vulnerability Research*. Syngress Publishing, 2007. [see page 56]

[FVM09] Josè Fonseca, Marco Vieira, and Henrique Madeira. Vulnerability and attack injection for web applications. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 93–102, July 2009. [see page 4]

[FVM14] Josè Fonseca, Marco Vieira, and Henrique Madeira. Evaluation of Web Security Mechanisms using Vulnerability and Attack Injection. *IEEE*

Transactions on Dependable and Secure Computing, PrePrints(99):1–1, 2014. [see pages 23, 32, 34, 73, and 100]

[GER08] Mohammed Gad El Rab. *Evaluation des systèmes de détection d'intrusion*. PhD thesis, Université Paul Sabatier - Toulouse III, December 2008. [see pages 32 and 73]

[GFD+06] Guofei Gu, Prahlad Fogla, David Dagon, Wenke Lee, and Boris Skorić. Measuring intrusion detection capability: an information-theoretic approach. In *Proceedings of the 2006 ACM Symposium on Information, computer and communications security (ASIACCS)*, pages 90–101, New York, NY, USA, 2006. ACM. [see pages 7, 23, 26, 43, 44, 45, 46, 48, 49, 75, and 121]

[GKRB09] Nico Görnitz, Marius Kloft, Konrad Rieck, and Ulf Brefeld. Active learning for network intrusion detection. In *Proceedings of the 2nd ACM workshop on Security and artificial intelligence (AISec)*, pages 47–54, New York, NY, USA, 2009. ACM. [see page 4]

[GMV+10] Frank Gens, Robert Mahowald, Richard L. Villars, David Bradshaw, and Chris Morris. Cloud Computing 2010: An IDC Update, 2010. [see pages 1, 2, and 63]

[GPB+03] John L. Griffin, Adam Pennington, John S. Bucy, Deepa Choundappan, Nithya Muralidharan, and Gregory R. Ganger. On the Feasibility of Intrusion Detection inside Workstation Disks, 2003. [see pages 28 and 30]

[GR03] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the Network and Distributed Systems Security Symposium*, pages 191–206, 2003. [see pages 18, 52, 64, and 72]

[GU01] John E. Gaffney and Jacob W. Ulvila. Evaluation of intrusion detectors: a decision theory approach. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 50–61, 2001. [see pages 7, 23, 43, 46, 75, 116, 121, and 127]

[HBK+17] Nikolas Herbst, Steffen Becker, Samuel Kounev, Heiko Koziolek, Martina Maggio, Aleksandar Milenkoski, and Evgenia Smirni. Metrics and Benchmarks for Self-Aware Computing Systems. In Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu, editors, *Self-Aware Computing Systems*. Springer Verlag Berlin Heidelberg, Germany, 2017. To Appear. [see page xi]

[HH05] Simon Hansman and Ray Hunt. A taxonomy of network and computer attacks. *Computers and Security*, 24(1):31 – 43, 2005. [see page 13]

[HKR13] Nikolas R. Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in Cloud Computing: What it is, and What it is Not (Short Paper). In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 2013)*. USENIX, June 2013. [see page 6]

[HL09] Cuong H. Le. Protecting Xen Hypercalls: Intrusion Detection/Prevention in a Virtualization Environment. Master's thesis, University of British

Columbia, Vancouver, Canada, 2009. [see pages 8, 92, 93, 97, 99, 143, 148, and 149]

[hon] Honeyd. `http://www.honeyd.org/`. [see page 5]

[HS11] Amin Hassanzadeh and Radu Stoleru. Towards Optimal Monitoring in Cooperative IDS for Resource Constrained Wireless Networks. In *Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN)*, pages 1–8, August 2011. [see pages 51, 52, and 53]

[Htt] Httpbench. `http://freecode.com/projects/httpbench`. [see page 28]

[HVO06] William G. J. Halfond, Jeremy Viegas, and Alessandro Orso. A Classification of SQL-Injection Attacks and Countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, Arlington, VA, USA, March 2006. [see page 14]

[HW02] Mike Hall and Kevin Wiley. Capacity verification for high speed network intrusion detection systems. In *Proceedings of the 5th International Conference on Recent Advances in Intrusion Detection (RAID)*, pages 239–251, Berlin, Heidelberg, 2002. Springer-Verlag. [see pages 6, 7, 61, and 122]

[Hyp] Hypervisor Top-Level Functional Specification: Windows Server 2012. `https://msdn.microsoft.com/en-us/virtualization/hyperv_on_windows/develop/tlfs`. [see page 83]

[IDE] DARPA Intrusion Detection Evaluation. `https://www.ll.mit.edu/ideval/`. [see pages 5, 25, and 36]

[IFB] Iozone Filesystem Benchmark. `http://www.iozone.org/`. [see pages 28 and 62]

[inj] 1337day Exploit Database. `http://0day.today/`. [see page 4]

[ISA] Microsoft Internet Security and Acceleration Server 2006. `https://technet.microsoft.com/de-de/library/bb898433.aspx`. [see page 20]

[ita] The Internet Traffic Archive (ITA). `http://ita.ee.lbl.gov/html/traces.html`. [see pages 5 and 36]

[JADAD06] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Antfarm: tracking processes in a virtual machine environment. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 1–1, Berkeley, CA, USA, 2006. USENIX Association. [see page 64]

[JMK17] K. R. Jayaram, Aleksandar Milenkoski, and Samuel Kounev. Software Architectures for Self-Protection in IaaS Clouds. In Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu, editors, *Self-Aware Computing Systems*. Springer Verlag Berlin Heidelberg, Germany, 2017. To Appear. [see page xi]

[JXZ+09] Hai Jin, Guofu Xiang, Feng Zhao, Deqing Zou, Min Li, and Lei Shi. VM-Fence: a customized intrusion prevention system in distributed virtual

computing environment. In *Proceedings of the 3rd International Conference on Ubiquitous Information Management and Communication (ICUIMC)*, pages 391–399, New York, NY, USA, 2009. ACM. [see pages 16, 19, 28, 52, and 64]

[JXZ+11] Hai Jin, Guofu Xiang, Deqing Zou, Song Wu, Feng Zhao, Min Li, and Weide Zheng. A VMM-based intrusion prevention system in cloud computing environment. *The Journal of Supercomputing*, pages 1–19, 2011. [see pages 1, 26, 28, 30, 52, 63, 72, and 145]

[Kat97] Jeffrey Katcher. PostMark: a new file system benchmark. Network Appliance Tech Report TR3022, October 1997. [see page 28]

[KGL04] Anita Komlodi, John R. Goodall, and Wayne G. Lutters. An Information Visualization Framework for Intrusion Detection. In *CHI '04 Extended Abstracts on Human Factors in Computing Systems*, page 1743, New York, NY, USA, 2004. ACM. [see page 21]

[Kiv07] Avi Kivity. KVM: the Linux Virtual Machine Monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, July 2007. [see pages 78 and 153]

[KKMZ17] Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu. Self-Aware Computing Systems. Springer Verlag Berlin Heidelberg, Germany, 2017. To Appear. [see page ix]

[Kon11] Jinzhu Kong. AdjointVM: a new intrusion detection model for cloud computing. *Energy Procedia*, 13(0):7902–7911, 2011. International Conference on Energy Systems and Electrical Power (ESEP). [see pages 1 and 63]

[KVV05] Christopher Kruegel, Fredrik Valeur, and Giovanni Vigna. *Intrusion Detection and Correlation - Challenges and Solutions*, volume 14 of *Advances in Information Security*. Springer, 2005. [see page 15]

[KZ05] Pradeep Kannadiga and Mohammad Zulkernine. DIDMA: A Distributed Intrusion Detection System Using Mobile Agents. In *Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, pages 238–245, 2005. [see page 52]

[lbn] LBNL/ICSI Enterprise Tracing Project. `http://www.icir.org/enterprise-tracing/`. [see pages 5, 35, and 36]

[LDP11] Flavio Lombardi and Roberto Di Pietro. Secure virtualization for cloud computing. *Journal of Network and Computer Applications*, 34(4):1113–1122, July 2011. [see pages 1, 5, 20, 28, 29, 52, 63, 64, 69, and 72]

[LGT08] Michael Le, Andrew Gallagher, and Yuval Tamir. Challenges and Opportunities with Fault Injection in Virtualized Systems. In *VPACT*, 2008. [see page 100]

[LHF+00] Richard Lippmann, Joshua W. Haines, David J. Fried, Jonathan Korba, and Kumar Das. The 1999 DARPA off-line intrusion detection evaluation. *Computer Networks*, 34(4):579–595, October 2000. [see pages 5 and 26]

[LL13] Po-Ching Lin and Jia-Hau Lee. Re-examining the performance bottle-

neck in a NIDS with detailed profiling. *Journal of Network and Computer Applications*, 36(2):768 – 780, 2013. [see page 144]

[LMJ07] Marcus Laureano, Carlos Maziero, and Edgard Jamhour. Protecting host-based intrusion detectors through virtual machines. *Computer Networks*, 51(5):1275–1283, April 2007. [see pages 30, 52, and 53]

[LP13] Hoang Le and Viktor K. Prasanna. A Memory-Efficient and Modular Approach for Large-Scale String Pattern Matching. *IEEE Transactions on Computers*, 62(5):844–857, May 2013. [see page 144]

[LS09] Dirk Leinenbach and Thomas Santen. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *Proceedings of the 2nd World Congress on Formal Methods*, pages 806–809. Springer-Verlag, 2009. [see page 90]

[LTfPA] LMbench Tools for Performance Analysis. `http://www.bitmover.com/lmbench/`. [see page 30]

[MAW] MAWILab. `http://www.fukuda-lab.org/mawilab/index.html`. [see page 23]

[McH00] John McHugh. Testing Intrusion Detection Systems: a critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by Lincoln Laboratory. *ACM Transactions on Information and System Security*, 3(4):262–294, November 2000. [see pages 3 and 36]

[Men12] Yuxin Meng. Measuring intelligent false alarm reduction using an ROC curve-based approach in network intrusion detection. In *IEEE International Conference on Computational Intelligence for Measurement Systems and Applications (CIMSA)*, pages 108–113, July 2012. [see pages 46, 47, and 75]

[MFER] Metasploit Framework: Exploit Ranking. `https://github.com/rapid7/metasploit-framework/wiki/Exploit-Ranking`. [see page 32]

[MHL+03] Peter Mell, Vincent Hu, Richard Lippmann, Josh Haines, and Marc Zissman. An Overview of Issues in Testing Intrusion Detection Systems, 2003. [see pages 6, 7, 23, 29, 54, 74, 119, and 122]

[mic] Internet Security and Acceleration Server 2006: Common Criteria Evaluation. `https://www.commoncriteriaportal.org/files/epfiles/0453b.pdf`. [see page 20]

[MIK+13] Aleksandar Milenkoski, Alexandru Iosup, Samuel Kounev, Kai Sachs, Piotr Rygielski, Jason Ding, Walfredo Cirne, and Florian Rosenberg. Cloud Usage Patterns: A Formalism for Description of Cloud Usage Scenarios. Technical Report SPEC-RG-2013-001 v.1.0.1, SPEC Research Group - Cloud Working Group, Standard Performance Evaluation Corporation (SPEC), 7001 Heritage Village Plaza Suite 225, Gainesville, VA 20155, USA, May 2013. [see page xi]

[MIK+16] Aleksandar Milenkoski, Alexandru Iosup, Samuel Kounev, Kai Sachs, Diane E. Mularz, Jonathan A. Curtiss, Jason J. Ding, Florian Rosenberg, and Piotr Rygielski. CUP: A Formalism for Expressing Cloud Usage Patterns for Experts and Non-Experts. *IEEE Cloud Computing*, 2016. To

Appear. [see page x]

[MJA+16] Aleksandar Milenkoski, K. R. Jayaram, Nuno Antunes, Marco Vieira, and Samuel Kounev. Quantifying the Attack Detection Accuracy of Intrusion Detection Systems in Virtualized Environments. In *Proceedings of The 27th IEEE International Symposium on Software Reliability Engineering (ISSRE 2016)*, Washington DC, USA, October 2016. IEEE, IEEE Computer Society. To Appear. [see pages ix and 120]

[MJK17] Aleksandar Milenkoski, K. R. Jayaram, and Samuel Kounev. Benchmarking Intrusion Detection Systems with Adaptive Provisioning of Virtualized Resources. In Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu, editors, *Self-Aware Computing Systems*. Springer Verlag Berlin Heidelberg, Germany, 2017. To Appear. [see pages x and 120]

[MK12] Aleksandar Milenkoski and Samuel Kounev. Towards Benchmarking Intrusion Detection Systems for Virtualized Cloud Environments (Work-in-Progress Paper). In *Proceedings of the 7th International Conference for Internet Technology and Secured Transactions (ICITST 2012)*, pages 562–563, New York, USA, December 2012. IEEE. [see pages x and 6]

[MKA+13] Aleksandar Milenkoski, Samuel Kounev, Alberto Avritzer, Nuno Antunes, and Marco Vieira. On Benchmarking Intrusion Detection Systems in Virtualized Environments. Technical Report SPEC-RG-2013-002 v.1.0, SPEC Research Group - IDS Benchmarking Working Group, Standard Performance Evaluation Corporation (SPEC), 7001 Heritage Village Plaza Suite 225, Gainesville, VA 20155, USA, June 2013. [see pages xi and 2]

[ML12] Yuxin Meng and Wenjuan Li. Adaptive Character Frequency-Based Exclusive Signature Matching Scheme in Distributed Intrusion Detection Environment. In *IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 223–230, June 2012. [see pages 5, 20, 30, and 40]

[MM11] Carlo Maiero and Marino Miculan. Unobservable intrusion detection based on call traces in paravirtualized systems. In *Proceedings of the International Conference on Security and Cryptography (SECRYPT)*, 2011. [see pages 3, 8, 10, 98, 99, 109, 116, 143, and 149]

[MOL+11] Noman Mohammed, Hadi Otrok, Wang Lingyu, Mourad Debbabi, and Prabir Bhattacharya. Mechanism Design-Based Secure Leader Election Model for Intrusion Detection in MANET. *IEEE Transactions on Dependable and Secure Computing*, 8(1):89–103, January-February 2011. [see pages 5 and 20]

[MP13] Chirag Modi and Dhiren Patel. A Novel Hybrid-network Intrusion Detection System (H-NIDS) in Cloud Computing. In *IEEE Symposium on Computational Intelligence in Cyber Security (CICS)*, pages 23–30, April 2013. [see page 19]

[MPA+13] Aleksandar Milenkoski, Bryan D. Payne, Nuno Antunes, Marco Vieira, and Samuel Kounev. HInjector: Injecting Hypercall Attacks for Evaluating VMI-based Intrusion Detection Systems. In *Poster Reception at the 2013 Annual Computer Security Applications Conference (ACSAC 2013)*, Maryland, USA, 2013. Applied Computer Security Associates (ACSA). [see pages x, 78, and 93]

[MPA+14] Aleksandar Milenkoski, Bryan D. Payne, Nuno Antunes, Marco Vieira, and Samuel Kounev. An Analysis of Hypercall Handler Vulnerabilities. In *Proceedings of The 25th IEEE International Symposium on Software Reliability Engineering (ISSRE 2014)*. IEEE, 2014. [see pages ix, 8, 78, 97, 99, 107, and 143]

[MPA+15] Aleksandar Milenkoski, Bryan D. Payne, Nuno Antunes, Marco Vieira, Samuel Kounev, Alberto Avritzer, and Matthias Luft. Evaluation of Intrusion Detection Systems in Virtualized Environments Using Attack Injection. In *The 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID 2015)*. Springer, November 2015. [see pages ix and 98]

[MR04] Roy A. Maxion and Rachel R. Roberts. Proper Use of ROC Curves in Intrusion/Anomaly detection. Technical Report CS-TR-871, School of Computing Science, University of Newcastle upon Tyne, November 2004. [see page 6]

[MS01] Patrick Mueller and Greg Shipley. To Catch a Thief, August 2001. [see page 26]

[MVK+15] Aleksandar Milenkoski, Marco Vieira, Samuel Kounev, Alberto Avritzer, and Bryan D. Payne. Evaluating Computer Intrusion Detection Systems: A Survey of Common Practices. *ACM Computing Surveys*, 48(1):12:1–12:41, Sep 2015. [see pages ix, 25, 124, 139, and 141]

[MVP+14] Aleksandar Milenkoski, Marco Vieira, Bryan D. Payne, Nuno Antunes, and Samuel Kounev. Technical Information on Vulnerabilities of Hypercall Handlers. Technical Report SPEC-RG-2014-001 v.1.0, SPEC Research Group - IDS Benchmarking Working Group, Standard Performance Evaluation Corporation (SPEC), 7001 Heritage Village Plaza Suite 225, Gainesville, VA 20155, USA, August 2014. [see pages xi and 154]

[MWBI] MindCraft WebStone Benchmark Information. `http://www.mindcraft.com/webstone/`. [see page 30]

[NAEKF11] Khalid Nasr, Anas Abou El Kalam, and Christian Fraboul. An IDS Evaluation-Centric Taxonomy of Wireless Security Attacks. In *Advances in Network Security and Applications*, volume 196 of *Communications in Computer and Information Science*, pages 402–413. Springer Berlin Heidelberg, 2011. [see page 13]

[NBH08] Kara Nance, Matt Bishop, and Brian Hay. Virtual Machine Introspection: Observation or Interference? *IEEE Security and Privacy*, 6:32–37, 2008.

[see page 64]

[neo] Neohapsis Open Security Evaluation Criteria (OSEC). http://osec.neohapsis.com/. [see page 26]

[Nes] Nessus. http://www.tenable.com/products/nessus-vulnerability-scanner. [see page 32]

[NGI] NGIPS Test Methodology: v2.0. https://www.nsslabs.com/research-advisory/library/industry/methodologies/ngips-test-methodology-v2-0/. [see page 23]

[nid] Nidsbench - a network intrusion detection system testing suite. http://www.ussrback.com/UNIX/IDS/nidsbench/nidsbench.html. [see page 25]

[Nik] Nikto2. http://cirt.net/nikto2. [see page 31]

[NKF12] Khalid Nasr, Anas A. Kalam, and Christian Fraboul. Performance Analysis of Wireless Intrusion Detection Systems. In *Internet and Distributed Computing Systems*, pages 238–252. Springer, 2012. [see pages 4, 32, and 44]

[nssa] NSSLabs. https://www.nsslabs.com/. [see page 20]

[NSSb] NSSLabs. Breach Detection Test Methodology. https://www.nsslabs.com/linkservid/E12A4AD5-5056-9046-931193F7191C0AF7/. [see page 146]

[oss] Open Source SEcurity (OSSEC). http://www.ossec.net/. [see pages 3, 8, 16, 18, 19, 99, 143, 148, 149, and 150]

[OSVDO] Open Sourced Vulnerability Database (OSVDB). http://www.osvdb.org/. [see page 31]

[pata] Xen Security Advisory 12 (CVE-2012-3494). http://lists.xen.org/archives/html/xen-announce/2012-09/msg00000.html. [see page 170]

[patb] Xen Security Advisory 13 (CVE-2012-3495). http://lists.xen.org/archives/html/xen-announce/2012-09/msg00001.html. [see page 172]

[patc] Xen Security Advisory 14 (CVE-2012-3496). http://lists.xen.org/archives/html/xen-announce/2012-09/msg00002.html. [see page 156]

[patd] Xen Security Advisory 24 (CVE-2012-4539). http://lists.xen.org/archives/html/xen-announce/2012-11/msg00002.html. [see page 162]

[pate] Xen Security Advisory 26 (CVE-2012-5510). http://lists.xen.org/archives/html/xen-announce/2012-12/msg00001.html. [see page 164]

[patf] Xen Security Advisory 29 (CVE-2012-5513). http://lists.xen.org/archives/html/xen-announce/2012-12/msg00004.html. [see page 158]

[patg] Xen Security Advisory 32 (CVE-2012-5525). `http://lists.xen.org/archives/html/xen-announce/2012-12/msg00002.html`. [see page 174]

[path] Xen Security Advisory 50 (CVE-2013-1964). `http://lists.xen.org/archives/html/xen-announce/2013-04/msg00006.html`. [see page 167]

[PBSL13] Diego Perez-Botero, Jakub Szefer, and Ruby B. Lee. Characterizing Hypervisor Vulnerabilities in Cloud Computing Servers. In *Proceedings of the 2013 International Workshop on Security in Cloud Computing*, pages 3–10. ACM, 2013. [see page 1]

[PCKI11] Cuong Pham, Daniel Chen, Zbigniew Kalbarczyk, and Ravishankar Iyer. CloudVal: A Framework for Validation of Virtualization Environment in Cloud Infrastructure. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 189–196, 2011. [see page 100]

[PCOM97] Nicholas Puketza, Mandy Chung, Ronald A. Olsson, and Biswanath Mukherjee. A Software Platform for Testing Intrusion Detection Systems. *IEEE Software*, 14(5):43–51, September 1997. [see pages 21, 26, and 72]

[PKSZ04] Swapnil Patil, Anand Kashyap, Gopalan Sivathanu, and Erez Zadok. FS: An In-Kernel Integrity Checker and Intrusion Detection File System. In *Proceedings of the 18th USENIX Conference on System Administration (LISA)*, pages 67–78, Berkeley, CA, USA, 2004. USENIX Association. [see pages 28, 30, and 72]

[PR] TCPDUMP/LIBPCAP Public Repository. `http://www.tcpdump.org/`. [see page 26]

[pst] PacketStorm. `http://packetstormsecurity.com/`. [see page 4]

[PtsM] Penetration testing software: Metasploit. `http://www.metasploit.com/`. [see pages 4 and 32]

[PZC+96] Nicholas J. Puketza, Kui Zhang, Mandy Chung, Biswanath Mukherjee, and Ronald A. Olsson. A Methodology for Testing Intrusion Detection Systems. *IEEE Transactions on Software Engineering*, 22(10):719–729, October 1996. [see page 21]

[Ran01] Marcus J. Ranum. Experiences Benchmarking Intrusion Detection Systems. *White Paper*, 2001. [see pages 3 and 23]

[RAR12] Kannaiya N. Raja, Srinivasan Arulanandam, and Raja Rajeswari. Two-Level Packet Inspection Using Sequential Differentiate Method. In *International Conference on Advances in Computing and Communications (ICACC)*, pages 42–45, 2012. [see pages 5, 36, 52, 73, and 74]

[RCF+01] Lee M. Rossey, Robert K. Cunningham, David J. Fried, Jesse C. Rabek, Richard P. Lippmann, Joshua W. Haines, and Marc A. Zissman. LARIAT: Lincoln Adaptable Real-time Information Assurance Testbed. In *IEEE Proceedings of Aerospace Conference*, pages 2671–2682, 2001. [see page 25]

[RJX08] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 1–20, Berlin, Heidelberg, 2008. Springer-Verlag. [see pages 28, 30, and 52]

[Roe99] Martin Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX conference on System Administration (LISA)*, pages 229–238. USENIX Association, 1999. [see pages 2, 11, 16, 17, 18, 19, 49, 52, 54, 56, 144, 149, and 150]

[RRL+12] Jason Reeves, Ashwin Ramaswamy, Michael Locasto, Sergey Bratus, and Sean Smith. Intrusion detection for resource-constrained embedded control systems in the power grid. *International Journal of Critical Infrastructure Protection*, 5(2):74–83, 2012. [see pages 4, 26, 28, 30, 52, 72, and 74]

[RW] Joanna Rutkowska and Rafał Wojtczuk. Xen 0wning Trilogy: Part Two (presentation slides). `http://invisiblethingslab.com/resources/bh08/part2.pdf`. [see pages 8, 87, 94, and 99]

[Sam15] Samuel Kounev, Xiaoyun Zhu, Jeffrey O. Kephart, and Marta Kwiatkowska, editors. Aleksandar Milenkoski, assistant editor. Model-driven Algorithms and Architectures for Self-Aware Computing Systems. Dagstuhl Reports, 2015. Dagstuhl, Germany. [see page xi]

[SB10] IBM Smart Business. Dispelling the vapor around cloud computing. Whitepaper, 2010. [see page 67]

[SC] SC Magazine: IDS/IPS Group Test. `http://www.scmagazine.com/idsips/grouptest/241/`. [see page 23]

[seca] SecuriTeam Exploit Database. `http://www.securiteam.com/exploits/`. [see page 4]

[secb] SecurityFocus. `http://www.securityfocus.com/`. [see page 4]

[Shi99] R. Shirey. Internet Security Glossary (RFC 2828). `http://tools.ietf.org/html/draft-shirey-security-glossary-01`, January 1999. [see page 13]

[SHK+15] Simon Spinner, Nikolas R. Herbst, Samuel Kounev, Xiaoyun Zhu, Lei Lu, Mustafa Uysal, and Rean Griffith. Proactive Memory Scaling of Virtualized Applications. In *Proceedings of the 2015 IEEE 8th International Conference on Cloud Computing (IEEE CLOUD 2015)*, pages 277–284. IEEE, June 2015. [see page 119]

[SJP06] Sushant Sinha, Farnam Jahanian, and Jignesh M. Patel. WIND: Workload-aware INtrusion Detection. In *Proceedings of the 9th International Conference on Recent Advances in Intrusion Detection (RAID)*, pages 290–310, Berlin, Heidelberg, 2006. Springer Verlag. [see pages 5, 20, 26, 40, 52, 54, and 149]

[SK11] Sundararajan Subashini and Vinayav Kavitha. A survey on security issues in service delivery models of cloud computing. *Journal of Network and Computer Applications*, 34(1):1–11, 2011. [see page 1]

[SM] Snort Manual. `http://manual.snort.org/node16.html`. [see page 59]

[SM07] Karen Scarfone and Peter Mell. Guide to Intrusion Detection and Prevention Systems (IDPS), 2007. NIST Special Publication 900-94. [see pages 13, 16, and 141]

[SP07] Vidar E. Seeberg and Slobodan Petrovic. A New Classification Scheme for Anonymization of Real Data Used in IDS Benchmarking. In *The Second International Conference on Availability, Reliability and Security (ARES)*, pages 385–390, April 2007. [see page 35]

[SP10] Robin Sommer and Vern Paxson. Outside the Closed World: On Using Machine Learning For Network Intrusion Detection. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010. [see pages 36 and 53]

[SPB+] Reiner Sailer, Ronald Perez, Stefan Berger, Ramon Caceres, and Leendert van Doorn. Towards Enterprise-level Security with Xen – sHype Access Control module (ACM) (presentation slides). `http://pdub.net/proj/usenix08boston/xen_drive/resources/xensummit/slides/sailer_xensummit_sept07.pdf`. [see page 149]

[spea] SPECvirt sc 2013. `http://www.spec.org/virt_sc2013/`. [see pages 12 and 109]

[SPEb] SPEC CPU 2000. `http://www.spec.org/cpu2000/`. [see page 28]

[SPEc] SPECweb99. `http://www.spec.org/web99/`. [see page 30]

[SSG08] Abhinav Srivastava, Kapil Singh, and Jonathon Giffin. Secure Observation of Kernel Behavior. `http://hdl.handle.net/1853/25464`, 2008. [see pages 28, 30, 52, 53, 65, 66, 69, 72, 74, 99, 145, and 149]

[SSTG12] Ali Shiravi, Hadi Shiravi, Mahbod Tavallaee, and Ali Ghorbani. Toward developing a systematic approach to generate benchmark datasets for intrusion detection. *Computers and Security*, 31(3):357–374, 2012. [see pages 23, 37, 38, 39, and 74]

[SSWX13] Bo Sun, Xuemei Shan, Kui Wu, and Yang Xiao. Anomaly Detection Based Secure In-Network Aggregation for Wireless Sensor Networks. *IEEE Systems Journal*, 7(1):13–25, 2013. [see page 52]

[Sta02] William Stallings. *Cryptography and Network Security: Principles and Practice*. Pearson Education, 2002. [see page 13]

[SUBP08] Jaydip Sen, Arijit Ukil, Debasis Bera, and Arpan Pal. A distributed intrusion detection system for wireless ad hoc networks. In *16th IEEE International Conference on Networks (ICON)*, pages 1–6, 2008. [see pages 51, 52, and 53]

[sur] Suricata - Open source IDS / IPS / NSM engine. `http://suricata-ids.org/`. [see pages 11, 16, and 149]

[SYB04] Joel Sommers, Vinod Yegneswaran, and Paul Barford. A framework for malicious workload generation. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement (IMC)*, pages 82–87, New York, NY,

USA, 2004. ACM. [see page 3]

[TBASoU] The Berkeley Automounter Suite of Utilities. http://www.am-utils.org/. [see page 30]

[TBNSM] The Bro Network Security Monitor. http://www.bro.org/. [see pages 16, 144, and 149]

[TCM05] Katrina Tsipenyuk, Brian Chess, and Gary McGraw. Seven pernicious kingdoms: a taxonomy of software security errors. *IEEE Security and Privacy*, 3(6):81–84, Nov 2005. [see page 80]

[tcpa] tcpmkpub. http://www.icir.org/enterprise-tracing/tcpmkpub.html. [see page 35]

[Tcpb] Tcpreplay. http://tcpreplay.synfin.net/. [see pages 26 and 57]

[tLmp] top: Linux man page. http://linux.die.net/man/1/top. [see page 60]

[TMI] Trend Micro Inc. Detecting APT Activity with Network Traffic Analysis. http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp-detecting-apt-activity-with-network-traffic-analysis.pdf. [see page 146]

[tri] Tripwire, Inc. http://www.tripwire.com/. [see page 16]

[Uni] Unixbench. http://code.google.com/p/byte-unixbench/. [see page 28]

[UoC] University of California. KDD CUP'99 Data. http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html. [see pages 5 and 36]

[vpr] VProbe Toolkit. http://vprobe-toolkit.sourceforge.net/. [see page 65]

[w3a] w3af. http://w3af.org/. [see page 32]

[WCMX12] Feifei Wang, Ping Chen, Bing Mao, and Li Xie. RandHyp: Preventing Attacks via Xen Hypercall Interface. In *Information Security and Privacy Research*, volume 376, pages 138–149. Springer Berlin Heidelberg, 2012. [see pages 92, 93, 99, 148, and 149]

[WCS+02] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the 11th USENIX Security Symposium*, pages 17–31, Berkeley, CA, USA, 2002. USENIX Association. [see pages 28 and 30]

[WDW+14] Jing Z. Wu, Liping Ding, Yanjun Wu, Nasro Min-Allah, Samee U. Khan, and Yongji Wang. C^2Detector: A Covert Channel Detection Framework in Cloud Computing. *Security and Communication Networks*, 7(3):544–557, 2014. [see pages 99, 112, 143, and 149]

[WLR] Felix Wilhelm, Matthias Luft, and Enno Rey. Compromise-as-a-Service. https://www.ernw.de/download/ERNW_HITBAMS14_HyperV_fwilhelm_mluft_erey.pdf. [see page 99]

[Wor] Worldwide Intelligence Network Environment: Data Sharing. `http://securityresponse.symantec.com/about/profile/universityresearch/sharing.jsp`. [see page 21]

[WS02] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM conference on Computer and Communications security (CCS)*, pages 255–264, 2002. [see pages 103 and 118]

[xena] Manpage for XenTrace. `http://man.cx/xentrace`. [see pages 3 and 110]

[xenb] Xen Interface Manual. `http://pdub.net/proj/usenix08boston/xen_drive/resources/developer_manuals/interface.pdf`. [see page 154]

[xenc] Xen Wiki. `http://wiki.xenproject.org/wiki/Main_Page`. [see pages 154 and 160]

[xend] Xenaccess - A Virtual Machine Introspection Library for Xen. `http://code.google.com/p/xenaccess/`. [see page 64]

[XSM] XSM-FLASK. `http://wiki.xen.org/wiki/Xen_Security_Modules_:_XSM-FLASK`. [see pages 91, 92, 149, and 150]

[YD11] Senhua Yu and Dipankar Dasgupta. An effective network-based Intrusion Detection using Conserved Self Pattern Recognition Algorithm augmented with near-deterministic detector generation. In *2011 IEEE Symposium on Computational Intelligence in Cyber Security (CICS)*, pages 17–24, 2011. [see pages 5, 26, 36, and 73]

[YL13] William Young and Nancy Leveson. Systems Thinking for Safety and Security. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC 2013)*, pages 1–8. ACM, 2013. [see page 86]

[YMLFK13] Yuxin Meng and Lam-For Kwok. Towards an Information-Theoretic Approach for Measuring Intelligent False Alarm Reduction in Intrusion Detection. In *12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 241–248, July 2013. [see pages 49 and 75]

[Zan06] Stefano Zanero. My IDS is better than yours. Or is it? BlackHat Briefing, February 2006. [see page 25]

[ZWGW08] Youhui Zhang, Hongyi Wang, Yu Gu, and Dongsheng Wang. IDRS: Combining File-level Intrusion Detection with Block-level Data Recovery based on iSCSI. In *Third International Conference on Availability, Reliability and Security (ARES)*, pages 630–635, March 2008. [see pages 28, 30, and 52]