

ABSTRACT

BUSHOUSE, MICAH. Cloud-ready Hypervisor-based Security. (Under the direction of Douglas Reeves.)

Improving host security through virtualization has led to many novel “out-of-guest” Hypervisor-Based Security (HBS) systems. Unlike traditional operating-system-based security mechanisms, HBS systems are both isolated from the guest operating system and operate at a higher privilege level, making it difficult for in-guest malicious code to disable or even detect an HBS system. However, although HBS systems have been an active area of research for fifteen years, they have not yet been adopted as a routine security practice in production systems.

In this dissertation, we investigate several HBS shortfalls and propose solutions for scaling limitations, development and integration challenges, and lack of a safe cloud framework. We begin by introducing two scalable, low-overhead HBS systems. First, *Goalkeeper* enforces guest process-level security policies and scales across tens to hundreds of guests per hypervisor by focusing on asynchronous, stateless, and lightweight Virtual Machine Introspection (VMI) techniques. Goalkeeper minimizes overhead by limiting inspections to recently-changed guest processes. Second, we discuss *Arav*, an HBS system that leverages a new VMI-based security monitoring design in lieu of in-guest agents. Arav inexpensively monitors guests unsuitable for traditional security monitoring. Next, we address HBS development and integration by presenting a methodology to re-engineer existing security agents into *hyperagents*, hypervisor-based agents which gain the benefits of HBS while retaining their original in-guest capabilities. Hyperagents integrate easily with established security operations because they inherit the years of practitioner experience and best practices inherent in the original agent. When agents are consolidated from multiple adjacent guests and centralized on the hypervisor, their hyperagent form is more resource-efficient than its predecessor and allows for new agent features. Finally, we introduce *Furnace*, an open source cloud VMI framework for multi-hypervisor, multi-tenant cloud ecosystems. Furnace enables a tenant to safely run self-submitted VMI applications underneath their cloud VMs. As opposed to prior frameworks, Furnace takes a sandbox-based approach, enabling tenant VMI applications to run faster and require fewer resources. Furnace’s ease of use is demonstrated through four essential tools for memory analysis. Overall, this dissertation sets the conditions for cloud-compatible HBS systems that are more practical, easier to develop, and more widely applicable.

© Copyright 2018 by Micah Bushouse

All Rights Reserved

Cloud-ready Hypervisor-based Security

by
Micah Bushouse

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2018

APPROVED BY:

William Enck

Vincent Freeh

Alexandros Kapravelos

Douglas Reeves
Chair of Advisory Committee

DEDICATION

To Heather, Zachary, and Matthew.

ACKNOWLEDGEMENTS

I am immensely grateful to my advisor, Dr. Doug Reeves. His patience, understanding, and advice are the main drivers for my academic success. I am in awe of his ability to ask the perfect question at the precisely-timed moment to either induce a leap forward or a gentle nudge down a better path.

Over the years I was fortunate to have close relationships with my committee, Drs. W. Enck, A. Kapravelos, V. Freeh, and M. Kang. I considered myself lucky that it was often they who reached out to check in on me instead of vice versa. Thank you for challenging me.

I am especially thankful for the NCSU CSC department's administrative staff, including Dr. G. Rouskas, K. Luca, A. Sleeth, T. Gardner, C. Benavente, and T. Murdoch. The good will you've shown me has provided the steady base (and a good amount of electricity) for my work. I also enjoyed collaborating with A. Workman and the rest of A. Kotynski's Security and Compliance team.

During my time in the WSPR lab, Dr. J. Staddon, Dr. A. Scafuro, Dr. A. Nadkarni, Dr. B. Reaves, L. Deshotels, I. Polinsky, S. Ahn, B. Andow, A. Verma, A. Gorski, L. Enloe, J. Jueckstock, S. Elder, and A. Haque kept me continuously on my feet. Over time, an individual's values are shaped by those of his or her organization, and in my case this is the US military. I hope that over the years we've worked together as a lab some of these values and attitudes have helped you. Despite this, I modestly acknowledge the reality that it's equally likely that I'll simply be remembered as that one salty guy who came in early every day and complained loudly about containers.

I would not be in this position if key leaders in the US Army had not advocated on my behalf, especially J. Dillon, I. MacLeod, LTC C. Saie, LTC T. OConnor, MAJ I. Faber, CW3(R) J. Babesh, and LTC S. Applegate. LTC P. Godbey provided the initial encouragement in 2014, while more recently LTC C. Saie and CPT T. Schindler performed untold amounts of blocking and tackling on my behalf.

I am grateful for my summer in Albuquerque with Dr. K. Dixon, R. Vestal, Dr. J. Haas, C. Babcock, C. Loverro, J. Ingram, and Dr. M. Smith. Sandia is truly a great place to work.

Many of those mentioned above assisted in reviewing my work—willingly or otherwise—as did my colleagues CPT N. Hicks and M. Kaar, the chair of my preliminary exam, Dr. E. Gehringer, and the anonymous reviewers around the world.

I owe a great deal to the open source community for providing the foundation for many of my projects, specifically those involved in the Linux kernel, Fedora, Xen, LibVMI, DRAKVUF, GRR, Rekall, Python, SaltStack, ZeroMQ, Protocol Buffers, and SystemTap.

Finally, thank you to my family, my beautiful wife Heather and our boys Zachary and Matthew. You provided encouragement, patiently endured my tribulations, and sometimes (rather cheerfully) took day trips to Nana and Papa's house on Saturdays so I could catch up. Grandma, Grandpa, uncles, and aunts were also constantly involved, allowing me to steal hours hiding in a corner to type.

Rangers lead the way!

TABLE OF CONTENTS

LIST OF TABLES	viii
LIST OF FIGURES	ix
Chapter 1 Introduction	1
1.1 The Rise of Virtualization	1
1.2 Virtualization as a Security Enabler	2
1.3 Problem Statement	2
1.4 Thesis Statement	3
1.5 Dissertation Contributions	4
1.6 Dissertation Organization	6
Chapter 2 Foundations and Related Work	8
2.1 Virtualization	8
2.2 Virtual Machine Introspection	12
2.3 VMI Primitives	13
2.3.1 Reading and Writing	14
2.3.2 EPT Violations	15
2.3.3 Register Updates	15
2.3.4 Interrupt Injection	16
2.4 VMI and Hypervisor Types	16
2.5 Below the Hypervisor	19
2.6 Why Hypervisor-based Security?	20
2.6.1 The Isolation Security Argument	20
2.6.2 Other Arguments	22
2.6.3 VMI Challenges	23
Chapter 3 Hypervisor-based Process Enforcement	26
3.1 Introduction	27
3.2 Motivation	28
3.3 Background	31
3.3.1 Threat Model	33
3.3.2 Assumptions	33
3.4 Design	33
3.4.1 Goalkeeper’s Concept and State Machine	34
3.4.2 Process Identification and Vetting	34
3.4.3 Process Termination	37
3.4.4 Implementation Details	39
3.5 Evaluation and Results	39
3.5.1 Evaluation Environment	40
3.5.2 100 Guest Experiment	40
3.5.3 MOV-to-CR3 Profiling	42
3.5.4 Goalkeeper Profiling	44

3.5.5	Malware Validation	45
3.6	Discussion	46
3.6.1	Goalkeeper's Policy Language	46
3.6.2	Resistance to DKOM	47
3.6.3	Resistance to Anti-Virtualization	47
3.6.4	Guest Kernel Sensitivity	47
3.6.5	Limitations	48
3.6.6	Related Work	48
3.7	Summary	49
Chapter 4	Hypervisor-based Monitoring	50
4.1	Introduction	51
4.2	Virtual Router Security	52
4.2.1	Background	52
4.2.2	VR Monitoring Challenges	54
4.2.3	VR Monitoring Requirements	54
4.2.4	Replacing Agents with VMI	55
4.2.5	Threat Model	55
4.3	Design and Architecture	56
4.3.1	Gaining Equivalent Context with VMI	57
4.3.2	Analyst Preparation	57
4.3.3	Inferring Security Events from Traces	58
4.3.4	VR and Network Service Selection	59
4.3.5	Attacks and Responses	61
4.3.6	Implementation Details	62
4.4	Evaluation	62
4.4.1	Experiment Environment	63
4.4.2	Experiments	63
4.5	Results	64
4.6	Discussion	65
4.6.1	VMI Breakpoint Protection	65
4.6.2	Analyst Effort with New VR Releases	65
4.6.3	Limitations	66
4.6.4	Related Work	66
4.7	Summary	67
Chapter 5	Developing Hypervisor-based Agents	68
5.1	Introduction	69
5.2	Motivation	70
5.3	Background	71
5.3.1	Host Agents	71
5.3.2	Hypervisor-based Security Agents	72
5.3.3	Live Forensics	73
5.3.4	Assumptions and Threat Model	73
5.4	Design and Methodology	74

5.4.1	Methodology Goals	75
5.4.2	Phase 1: Porting Agent Actuators	75
5.4.3	Phase 2: Consolidating Adjacent Agents	78
5.4.4	Phase 3: New Agent Functionality	79
5.5	Re-Engineering GRR	80
5.5.1	Phase 1	80
5.5.2	Phase 2	81
5.5.3	Phase 3	81
5.6	Evaluation and Results	84
5.6.1	Evaluation Environment	84
5.6.2	Development Work Factor	84
5.6.3	Execution Time and Network Bandwidth	85
5.6.4	Performance and Overhead	86
5.7	Discussion	87
5.7.1	Other Agents	87
5.7.2	Tradeoffs	88
5.7.3	Beyond the Datacenter	88
5.7.4	Memory Deduplication	88
5.7.5	Adding VMI Functionality	88
5.7.6	Related Work	89
5.8	Summary	90
Chapter 6	Tenant-driven Cloud VMI	91
6.1	Introduction	91
6.2	Background	93
6.2.1	Modern Clouds	94
6.3	Design Goals	95
6.4	Design	98
6.4.1	Security	101
6.4.2	Performance	107
6.5	Case Studies	109
6.6	Evaluation and Results	111
6.6.1	Security Evaluation	111
6.6.2	Performance Evaluation	113
6.7	Discussion	114
6.7.1	Related Work	116
6.8	Summary	116
Chapter 7	Directions for Hypervisor-based Security	118
7.1	Dissertation Summary	118
7.2	Trends	119
7.3	Future Work	121
7.4	Concluding Remarks	122
BIBLIOGRAPHY		124

APPENDICES	136
Appendix A	Goalkeeper Whitelisting Mechanism
A.1	The Whitelisting Check
A.2	Grammar Specification
A.3	Example Policy
Appendix B	Other Potential Hyperagents
B.1	Security: ClamAV Scanner
B.2	Security: OSSEC Intrusion Detection
B.3	Monitoring: Nagios NRPE
B.4	Monitoring: Datadog

LIST OF TABLES

Table 3.1	Process Supervision Comparison	29
Table 3.2	Policy Rule Components	35
Table 3.3	Memory Manipulation Techniques	36
Table 3.4	Goalkeeper Profiling	44
Table 4.1	Security Monitoring Classes	56
Table 4.2	Virtual Routers (VRs)	60
Table 4.3	Target VR Services	61
Table 4.4	File Transfer Performance Tests	64
Table 4.5	pfSense Effectiveness Tests	65
Table 5.1	Example agents.	72
Table 5.2	A matrix of GRR agent actuators (columns) matched against the libhyperagent components (rows) used to function on a guest.	80
Table 5.3	Source Code Changes	85
Table 5.4	SysDiff Performance	85
Table 5.5	Agent Memory Usage	86
Table 5.6	In-guest Overheads	86
Table 5.7	Interrogation Latency. In Guest (IG) agent. HA (Hyperagent). Units in seconds. Statistics across 100 samples.	87
Table 6.1	Desirable Properties for a Cloud VMI Framework	96
Table 6.2	Sandbox Comparison	102
Table 6.3	Furnace Partitions	103
Table 6.4	VMI Performance Comparison	113

LIST OF FIGURES

Figure 1.1	Hypervisor-based security in the context of defense in depth. HBS systems are part of an integrated approach to defense.	4
Figure 1.2	A chronological progression of dissertation-related research, including each work's <u>P</u> roblem, <u>S</u> olution, and <u>M</u> ethodology. Techniques developed in earlier chapters are incorporated in later chapters.	7
Figure 2.1	A native hypervisor diagram based on Xen. The guest OS and processes run in CPLs 0 and 3, isolated from other guests by a hypervisor using hardware-accelerated virtualization support. The hypervisor and its management domain (here called dom0) run in VMX root mode. Beneath the virtualization layer exist two other layers which can support security applications: System Management Mode and the Intel Management Engine. To the right of the diagram are currently available products found at each layer.	10
Figure 2.2	A VMI tool (shown as the green HBS shield) can access guest memory, vCPUs, and the guest's VMCS through mechanisms provided by the hypervisor. . . .	12
Figure 2.3	The guest Reads and Writes VMI Primitive (left) and the EPT violations VMI Primitive (right).	14
Figure 2.4	Register Events VMI Primitive (left) and Interrupt Injection VMI Primitive (right).	16
Figure 2.5	Processing an injected breakpoint. When guest execution encounters a breakpoint, the guest vCPU is suspended. The hypervisor passes this event to the inspecting domain's VMI tool, which performs arbitrary action before removing the breakpoint and setting the vCPU to single-step mode. The guest vCPU single-steps across the breakpoint's former location, and immediately traps back into the hypervisor so that the breakpoint can be re-injected.	17
Figure 2.6	The semantic gap problem is typically addressed by a mapping that translates VMI-tool-provided semantic information in and out of raw guest memory. In this example, a semantic mapping is used to extract process information from a guest's main memory.	23
Figure 3.1	Guest Process Supervision	28
Figure 3.2	Process attributes are manifested as a new process is created and initialized. Over time (from top to bottom), the process's memory layout reflects new dynamically loaded libraries (CS 4, 7) and new page allocations (CS 8).	31
Figure 3.3	Goalkeeper spawns a monitoring process for each guest virtual machine to be monitored. Each monitoring process consists of a fast path and slow path, and both are driven by its corresponding guest's process context switches. The guest process being "switched to" is subject to Goalkeeper enumeration, inspection, and potential termination if it fails its policy check.	32
Figure 3.4	Goalkeeper's high-level state machine. Existing, stable processes are filtered out and quickly return execution to the guest. New or changed processes are enumerated and checked against their corresponding policy.	33

Figure 3.5	A summary of Goalkeeper’s performance overheads taken during the 100 guest experiment. Goalkeeper imposes a 3.1% in-guest performance overhead on guests it protects. Each per-guest Goalkeeper process requires an average of 15.1MB dom0 memory (RSS) and 5.6% dom0 CPU while inspecting 5 new guest processes/second. When operating under 100 guests, Goalkeeper increases dom0’s total system load by 9.4%.	40
Figure 3.6	As Goalkeeper is used to protect increasing numbers of guests, Goalkeeper’s per process Proportional Set Size (PSS) memory usage approaches 5.14MB, whereas CPU load grows linearly with the number of guests.	43
Figure 3.7	Once a process is labeled as unauthorized, Goalkeeper requires an average of $8.6\text{ms} \pm 0.8\text{ms}$ to terminate it.	43
Figure 3.8	The <i>MOV-to-CR3</i> round trip time overhead quantifies the time required to move a guest event through the hypervisor and into Goalkeeper’s fast path. . .	44
Figure 4.1	Two common VR roles: VR_1 provides cloud administrators with remote VPN access into the cloud management network, and T_4 serves as a firewall and NAT for a cloud tenant.	53
Figure 4.2	A VR process is monitored by Arav, which injects a breakpoint at the process’s function D. After the process encounters the breakpoint, the VR is briefly suspended while Arav records the event.	58
Figure 4.3	Arav’s capability set construction workflow.	59
Figure 4.4	A VR process’s BPs are recorded into traces, which are mapped into events and analyzed against rules.	60
Figure 4.5	Event generators reconstitute security events from the trace of Quagga OSPF’s function calls, and these security events are queried by rule analyzers for potential responses.	61
Figure 4.6	Arav’s Experiment diagram. Node A is an external host, the origin of attacks, the OSPF neighbor, and a VPN client. Node B is an internal host.	63
Figure 5.1	In-guest agents can be ported into hyperagents.	70
Figure 5.2	The three-phase methodology to re-engineer host agents into hyperagents. .	72
Figure 5.3	Adding <code>libhyperagent</code> enables agent actuators to access a guest’s kernel data structures (category 1), process memory (category 2), and file system (category 3). Using <code>libhyperagent</code> ’s VMI SyncMount component, this example GRR actuator iterates through a guest directory’s contents by temporarily mounting the guest file system into dom0.	74
Figure 5.4	The predictability of work queue behavior enables the hyperagent to divert a guest vCPU to <code>sys_sync</code>	77
Figure 5.5	The GRR architecture. From left to right: (A) a host contains a kernel and userspace processes. One of these processes is (B) the GRR C++ agent. The agent registers with (C) the GRR server across the network. (D) An investigator’s script interacts with GRR server APIs to create a <i>flow</i> that directs the agent to perform one or more of its client actions.	78
Figure 5.6	New phase 3 features take advantage of <i>locality</i> (moving work forward to the hyperagent) and <i>cross-VM visibility</i> (centralized work against all guests). . .	82

Figure 5.7	Comparing network activity between a script using the built-in GRR flows (top) and the UT performing computation forward on the hyperagent (bottom). By moving server-side analysis code forward to the hyperagent, multiple network round trips are avoided. Message counts in this figure are based on a single hypervisor ($H = 1$) running 8 guests ($G = 8$), each with 40 processes ($P = 40G = 320$).	83
Figure 6.1	Under guest G_1 , an anti-rootkit VMI tool ensures kernel integrity. The forensics tool under G_2 , G_3 , and G_4 extracts process memory images. The monitoring tool under G_4 uses breakpoint injection to trace events occurring inside guest userspace processes. Finally, an intrusion prevention tool inspects new G_6 processes. The differing guest sizes in the diagram indicate the variety of instance types and operating systems found in modern clouds.	93
Figure 6.2	In modern clouds, tenants can use cloud APIs to request a new VM ①. This request is processed by cloud services running on a controller node. These services allocate resources, perform checks, and schedule changes ②. They schedule the new VM on a compute node, which locates or downloads the VM's disk image, checks for and allocates resources, and prepares the VM's network ③. Finally, the VM is booted ④ and the tenant accesses their new VM using a remote administration tool ⑤.	94
Figure 6.3	A cloud VMI framework's objective is to provide safe, high-performance tenant access to a cloud provider hypervisor's VMI API.	95
Figure 6.4	Threat Model.	97
Figure 6.5	The Furnace architecture from the cloud provider's perspective.	98
Figure 6.6	The Furnace architecture from the cloud tenant's perspective.	99
Figure 6.7	Tenant model.	100
Figure 6.8	Furnace tenant programming model. This example tenant app and backend work together to intercept and whitelist new guest processes.	101
Figure 6.9	From bottom to top: the sandbox is assembled when a new app is started. . .	104
Figure 6.10	The syscall policy for the VMI app running in the sandbox's L_2 partition. The remaining ≈ 245 syscalls are denied.	104
Figure 6.11	This sandbox security analysis depicts the order in which security mechanisms are encountered by code running in a sandbox partition and asserts the sandbox is reasonably secure if certain criteria are met. Furnace's security evaluation will test elements of this assertion (the "shown by" column) to provide confidence that Furnace can successfully contain malicious VMI apps. <i>Notes: for brevity, only the L_0 partition is shown, but L_1 and L_2 are conceptually identical; the syscall inspector is not shown.</i>	106
Figure 6.12	In addition to sending single VMI calls ① to L_2 , a tenant VMI app can be written to batch VMI calls ② to improve performance. Batched calls in the same message can also be <i>pipelined</i> ③ where an earlier call is fed into subsequent calls. Pipelining can provide significant speedups for long chains of predictable and interrelated VMI calls, such as the enumeration of a guest's list of processes.	108

Figure 6.13	A tenant app source code snippet that demonstrates the use of Furnace's VMI call pipelining API. Here, the tenant programmer creates a batch object <code>o</code> , populates it with several hundred VMI calls, then serializes and sends it to L_2 to be processed by calling <code>batch_send()</code> . By pipelining related calls together in the same cross-call message, IPC-related overhead can be greatly reduced.	109
Figure 6.14	Process list performance measurements were taken across a variety of VMI platforms, including a native VMI tool, CloudVMI, and Furnace VMI apps. This figure's circled points correspond to the rows in Table 6.4.	113
Figure A.1	The whitelisting check.	138
Figure A.2	Grammar hierarchy.	139
Figure B.1	The following figures reference these indicated areas.	142

CHAPTER

1

INTRODUCTION

1.1 The Rise of Virtualization

Hardware virtualization enables multiple operating systems (OSs) (e.g., Windows, Linux) to run concurrently as virtual machines on the same physical host. First proposed in the 1960s, virtualization didn't achieve wide market adoption until hardware-accelerated virtualization became available for the x86 architecture in the mid-2000s. Since then, virtualization has become a foundational technology built into nearly all forms of computing. Virtualization has transformed many areas of information technology (IT), such as reducing costs by improving server utilization and management, making software development and testing easier, and increasing the speed at which businesses recover from disruptive events. Ultimately, the benefits of virtualization have given rise to entirely new sectors of business, such as the public cloud market and the resulting cloud-centric software projects, developer tools, and new cloud ecosystems which followed.

Perhaps unexpectedly, virtualization extends well beyond its initial success in the datacenter and is built into most x86 and ARM CPUs, including those in modern cell phones, tablets, laptops, desktops, and many embedded devices. As a result, virtualization can be used in a wide range of applications including small System on a Chip (SoC) boards such as the Raspberry Pi, Aircraft [KW08], Drones [Yoo17], and satellites.¹ In fact, embedded virtualization is of growing importance for mixed-

¹<https://dornerworks.com/news/space-qualified-hypervisor-supports-virtualization-satellite-payloads>

criticality workloads as health, safety, and other critical functions must co-exist on the same device.²

1.2 Virtualization as a Security Enabler

Virtualization's hardware-enforced isolation has made it attractive for security applications. Unlike traditional operating-system-based security mechanisms, these virtualization-backed security systems are both isolated from a guest³ operating system and operate at a higher privilege level, making it difficult for in-guest malicious code to disable or even detect a hypervisor-based system. As a result, virtualization has enabled researchers to safely run and analyze malicious files, detect or prevent attacks against guests, and quickly remediate large swaths of virtual machines. Additional security-related applications, such as large scale honeypots, honeynets, and honeyclients, are all made practical by virtualization. Entire virtual networks of virtual machines can be created inside a single physical host in order to present a convincing target for attackers. Finally, the commercialization of hypervisor-based security continues to improve, with major IT companies like Xen, Microsoft, VMware, and BitDefender introducing unique solutions into the mainstream market.

This dissertation centers on *Hypervisor-Based Security* (HBS), which this dissertation defines as the employment of guest-centric hypervisor-based mechanisms designed to improve the security posture of a hypervisor's guests. HBS relies upon *Virtual Machine Introspection* (VMI) techniques to inspect, interpose, and introspect [GR03] upon a virtual machine. These VMI techniques are used by an HBS system to infer and influence activity inside a guest, i.e., HBS systems apply VMI techniques.⁴

1.3 Problem Statement

Despite 15 years of research and dozens of papers on using VMI for security, HBS has yet to be adopted as a routine security practice for production environments. VMI related work, surveys, and systemization of knowledge (SoK) papers often note several contributing factors to the lack of adoption. Of these, this dissertation focuses on the following:

- *VMI struggles to scale* [Jai14]. VMI's interposition on guest execution often introduces noticeable performance overheads. Additionally, VMI often causes frequent CPU mode switches which also introduce overhead.
- *VMI lacks high-level frameworks* [Wan15a]. VMI research occurs at a low technical level. To be used in production, an HBS system must be integrated into larger systems. The degree of inte-

²<https://www.zededa.com/development/linux-foundation-announces-embedded-virtualization-platform>

³This dissertation uses *guest*, *virtual machine*, and *domain* interchangeably.

⁴Another term in literature, *VMI tool*, is defined as a program that incorporates VMI techniques. By this definition, an HBS system could be considered to be a VMI tool that is designed to perform some form of in-guest security.

gration effort required in the incorporation of VMI techniques likely makes HBS unattractive to security teams.

- *VMI is not presently available in multi-tenant cloud ecosystems* [Jai14]. VMI research is typically centered on standalone hypervisors. This results in implementations that are not compatible with how modern cloud ecosystems are designed. Although prior work has proposed frameworks to enable VMI in the cloud, none have seen successful adoption.

1.4 Thesis Statement

In this dissertation, we investigate solutions to three HBS shortfalls: scale, development and integration challenges, and lack of a safe cloud framework. First, an HBS must scale to efficiently monitor many adjacent guests. Second, building an HBS system and integrating it into a larger security enterprise must be feasible. Finally, in a cloud environment, tenants must be able to safely and inexpensively deploy their own HBS systems across their cloud VMs. From these, we form our thesis statement:

As shown through low-overhead designs, methodical integration into existing defenses, ease of use, and flexibility across virtualization platform, hypervisor-based security contributes to the integrated defense of virtualized systems.

Thesis Principles:

- *The practical application of HBS.* The proposals in this dissertation are designed and evaluated with a emphasized sensitivity to in-guest overheads and the ability to operate against large sets of guests, aspects which have limited the adoption of prior work.
- *HBS as a part of an integrated defense.* The design and implementation of prior HBS prototypes often stops short and leaves a gap between the prototype and a realistic scenario where it might be applied. This dissertation exhibits extra effort to integrate with existing defense in depth best practices (see Figure 1.1) and perform end-to-end tests that would more closely mimic real world conditions.
- *The defense of virtualized systems.* Virtual machines are not limited to data centers—the typical evaluation environment of prior work. This dissertation is inclusive and includes multiple environments, including desktop virtualization and the cloud. Additionally, the evaluations in this dissertation are designed to account for diversity in virtual machines. To demonstrate generality, evaluations often include two types of guest operating system.

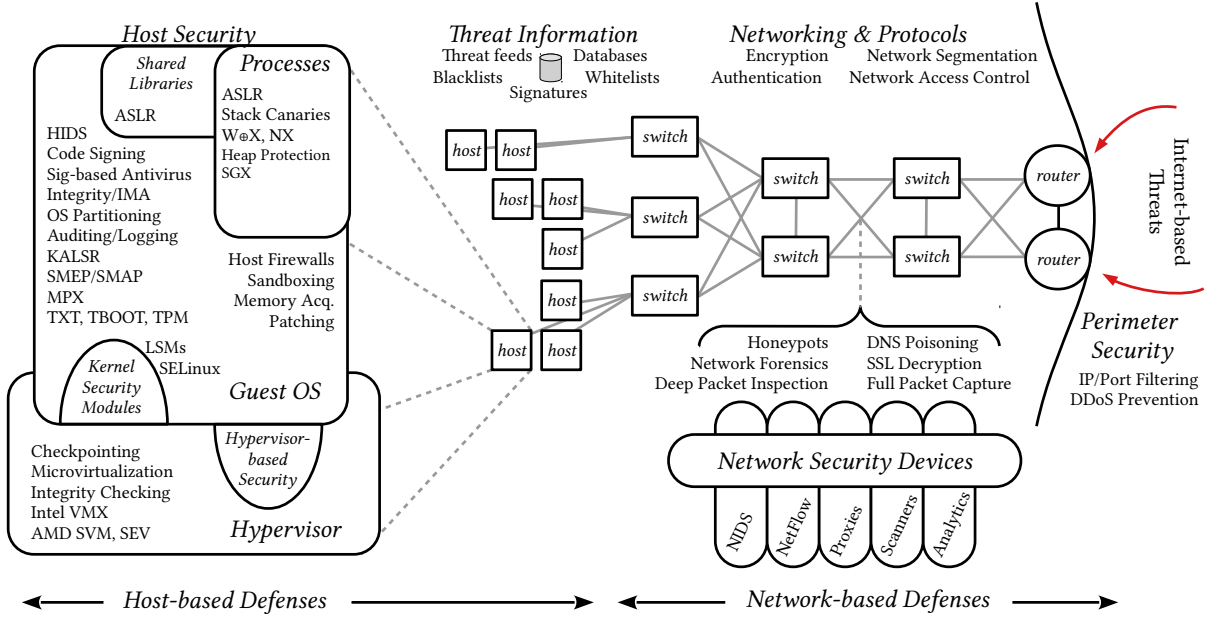


Figure 1.1 Hypervisor-based security in the context of defense in depth. HBS systems are part of an integrated approach to defense.

1.5 Dissertation Contributions

- *The scalability of Goalkeeper.* Goalkeeper [BR17] enforces guest process-level security policies and scales across tens to hundreds of guests per hypervisor by focusing on asynchronous, stateless, and lightweight Virtual Machine Introspection (VMI) techniques. Goalkeeper minimizes overhead by limiting inspections to recently-changed guest processes. In an evaluation across 100 Linux virtual desktops, Goalkeeper imposed a guest performance overhead of less than 3.1% and terminated all unauthorized processes within an average of 8.6ms. This work was published in COSE’18.

Key Findings:

1. CR3 interception, originally thought prohibitively expensive, can be managed by an optimized design.
 2. The proper use of computational parallelism allows VMI to scale linearly until CPUs become contended.
 3. Hypervisor-based security benefits from a flexible, extensible policy language.
- *The efficient monitoring of Arav.* Arav [Bus] is an HBS system that leverages a new VMI-based security monitoring design in lieu of in-guest agents. Arav inexpensively monitors guests

unsuitable for traditional security monitoring, imposing less than 0.6% overhead while monitoring two popular virtual routers. This work was published in CISR'17.

Key Findings:

1. VMI can be used to monitor and integrate systems that cannot support existing monitoring tools.
 2. Judicious use of breakpoint injection (20-30 breakpoints) imposes acceptable overheads.
- *The ease of integration using the Hyperagent methodology.* We present a methodology [BR18] to re-engineer existing security agents into hyperagents, hypervisor-based agents which gain the benefits of HBS while retaining their original in-guest capabilities. Hyperagents integrate easily with established security operations because hyperagents inherit the years of practitioner experience and best practices of the original agent. Additionally, when consolidated from multiple adjacent guests and centralized on the hypervisor, a hyperagent is more resource-efficient than its predecessor, and allows for new agent features, such as enabling server-side work to run locally on host data. These benefits are illustrated by re-engineering the GRR live forensics agent, which used 40% less memory than its in-guest counterparts and enabled a 4.9x speedup for a representative data-intensive workload. This work was published in CODASPY'18.

Key Findings:

1. The effort to re-engineer a host agent with VMI can be minimized by following a structured methodology.
 2. With the assistance of VMI libraries, adding VMI to an existing software system is reasonable and straightforward for a developer.
 3. Hypervisor-based agents can be both more resource-efficient and more powerful than their in-guest counterparts.
- *Safe, fast, and featureful cloud compatibility with Furnace.* Furnace is an open source cloud VMI framework for multi-hypervisor, multi-tenant cloud ecosystems. Furnace enables a tenant to safely run self-submitted VMI applications underneath their cloud VMs. To do so, Furnace takes a sandbox-based approach to simultaneously address two requirements: protecting the cloud provider from arbitrary and potentially-malicious tenant VMI code, but permitting tenants to run full-featured, fast, and resource-efficient VMI applications. Furnace's security properties are shown to protect against the actions of malicious tenant apps. Furnace VMI apps are shown to be resource efficient and execute up to 300x faster than with previous VMI frameworks. Finally, Furnace's flexibility and ease of use is demonstrated by porting four existing security and monitoring tools as VMI apps. This work will be published in RAID'18.

Key Findings:

1. By isolating tenant VMI applications in a sandbox on the hypervisor, the applications can run faster and are less expensive than the equivalent approach taken by prior cloud VMI proposals.
2. A tenant-centered design allows each party to focus on their strengths: the cloud provider's ability to provide a flexible, safe, framework, and the cloud tenant's detailed knowledge of tenant systems and specific VMI requirements.

1.6 Dissertation Organization

The goal of this dissertation is to investigate three related VMI shortfalls through a practitioner-focused and systems-oriented approach. This dissertation proposes solutions to these problems in order to improve VMI adoption and accelerate future VMI research. Figure 1.2 presents a chronological progression of these proposals.

In Chapter 2, virtualization and its potential security applications are introduced. VMI's primitives are discussed and framed by prior work. Guest-centric, VMI-enabled security functionality is compared to related approaches taken by other CPU modes. Finally, the advantages and disadvantages of VMI and Hypervisor-Based Security (HBS) are discussed.

Chapter 3 introduces *Goalkeeper*, a guest process supervision HBS system that exhibits good scalability properties, both in its per-guest overheads and its ability to protect many guests simultaneously.

Chapter 4 describes *Arav*, a security monitoring HBS system that can inexpensively monitor guests that cannot utilize traditional monitoring approaches.

Chapter 5 proposes a methodology to re-engineer security agents to function from the hypervisor. The methodology organizes the problem into three well-defined steps and illustrates the benefits of the approach by evaluating a successfully ported forensics agent.

Chapter 6 describes *Furnace*, a cloud VMI framework that uses a new sandboxing-based approach to balance the competing priorities of a cloud provider its cloud tenants.

Chapter 7 recaps the challenges addressed by this dissertation, summarizes current VMI-related trends, and describes future work.

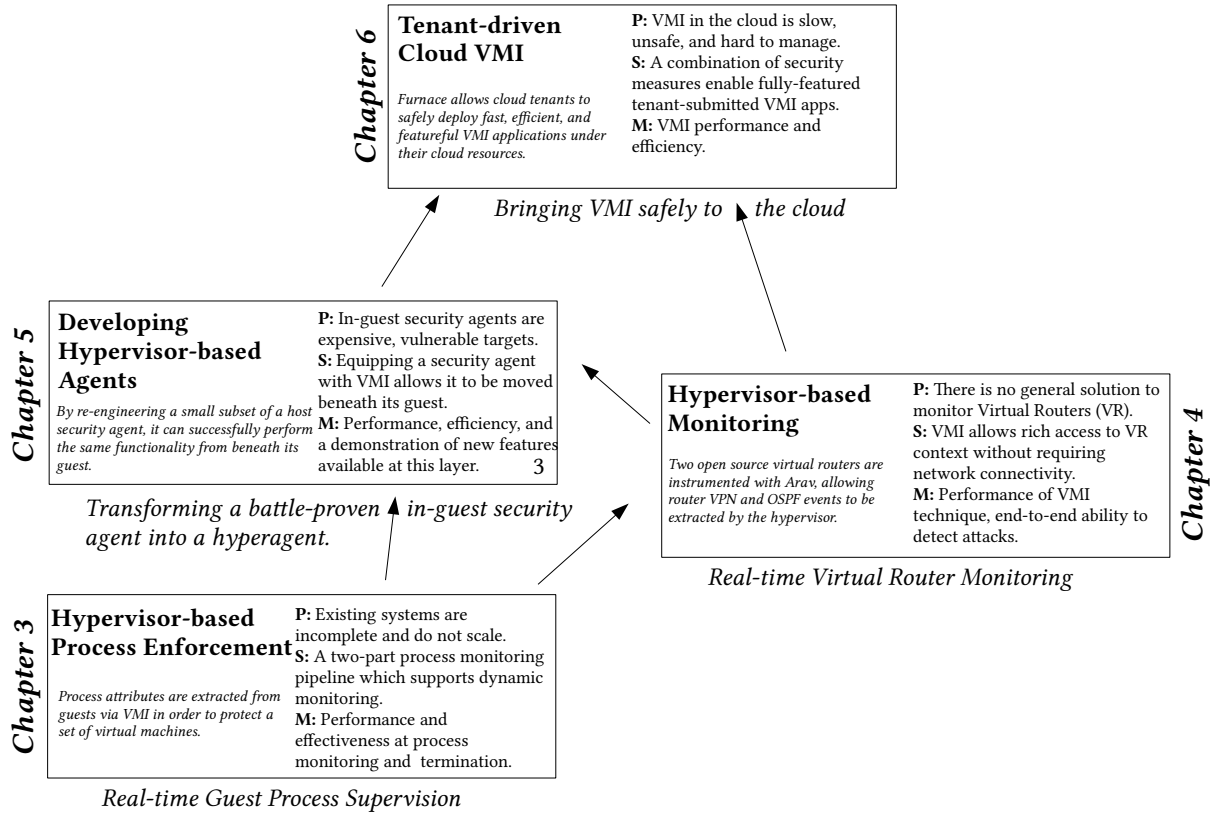


Figure 1.2 A chronological progression of dissertation-related research, including each work's Problem, Solution, and Methodology. Techniques developed in earlier chapters are incorporated in later chapters.

CHAPTER

2

FOUNDATIONS AND RELATED WORK

This section provides an introduction to this dissertation’s technical areas and their related work. First, a short history of virtualization is discussed (§2.1) before introducing Virtual Machine Inspection (VMI) (§2.2) and its four core primitives (§2.3). The two general types of hypervisors are discussed in the context of security (§2.4). Other CPU modes pertinent to security provide context to this dissertation’s hypervisor-based approach (§2.5). Finally, the motivations behind performing security at this layer are introduced (§2.6).

2.1 Virtualization

First implemented in the 1960s on IBM’s CP-40 and CP-67 mainframes [Gol73], hardware virtualization allows multiple guest operating systems to run concurrently and share the same physical host. The *Virtual Machine Monitor* (VMM), or *hypervisor*, is the “operating system for operating systems” responsible for the management of these guests, the presentation and control of resources allocated to them, and their segmentation and isolation [KF91].

Security-minded researchers quickly realized virtualization’s isolation properties also made it attractive for security applications. By necessity, virtualization inherently contains the same isolation properties that are valued in security [PG74], to the extent that early VMs were proposed to be included in strict multi-level security designs [MD73]. The rise and fall of the VAX VMM security kernel [Kar90; Kar91], besides being an early victim of the balance between security and

reality, hints at the relationship between security applications and virtualization that still largely exists today. A more complete treatment of the virtualization technologies that preceded modern hardware-accelerated processors is available [AA06].

The x86 architecture. Since the mid-1980s, CPUs on the x86 architecture have been designed to spend most of their time executing in *protected mode*.¹ Protected mode introduced four privilege levels, of which two are used—current privilege levels (CPLs) 0 and 3, more commonly referred to as ring 0 and ring 3 [Int16]. Ring 3 is designed to execute processes and user-interactive applications, such as web browsers, editors, and games. A process in ring 3 runs inside its own virtual address space, isolated and distinct from both the kernel beneath it and its adjacent processes [Gal13]. When a CPU is in ring 3, it is prohibited from executing certain privileged CPU instructions that are available to the OS in ring 0. The more privileged OS in ring 0 is responsible for the overall software system, including managing processes in ring 3, performing work on their behalf, and controlling access to the machine’s physical resources.

Early x86 virtualization. Virtualization’s introduction into mainstream computing began in the mid-1990s when researchers began experimenting with virtualization on the x86 architecture. The early days of x86 virtualization used *ring depriving* [Uhl05] to carefully coax a guest OS kernel into running in a ring other than ring 0. This depriving made room in ring 0 for the hypervisor, and was done purposefully to ensure the hypervisor retained sole control over the host’s physical resources. When guest kernels attempted to execute privileged instructions from their position in the upper rings (1, 2, or 3), these instructions could be trapped and emulated or translated into the equivalent unprivileged instructions by the hypervisor in ring 0 [Gal13; AA06]. There was a cascade of implications to early efforts to virtualize x86, but one in particular made processor manufacturers take notice: performance degradation.

Current x86 virtualization. Despite early results showing that hardware-accelerated hypervisors were actually slower than their more mature all-software counterparts [AA06], AMD and Intel began to develop and market processor features to accelerate virtualization. This culminated in 2006 with the introduction of new processor extensions, Virtual Machine eXtensions² (VMX), and a new CPU mode, VMX root mode. This mode introduces hardware-accelerated hypervisor-specific x86 instructions and remains the basis for current x86 virtualization. In VMX root mode, the hypervisor runs in VMX root ring 0, and a supporting management operating system (sometimes called a *helper guest*), usually a Linux-like OS, runs in VMX root ring 3. The hypervisor starts guest operating systems in VMX non-root mode, presents them with virtualized resources, and allows them to execute in (non-root mode) ring 0.

¹To be precise: modern 64-bit x86 processors run in *long mode* and adopt an identical concept of rings introduced by protected mode.

²This dissertation generally refers to virtualization terms by their Intel name. AMD’s hardware virtualization extensions are called Secure Virtual Machine (SVM).

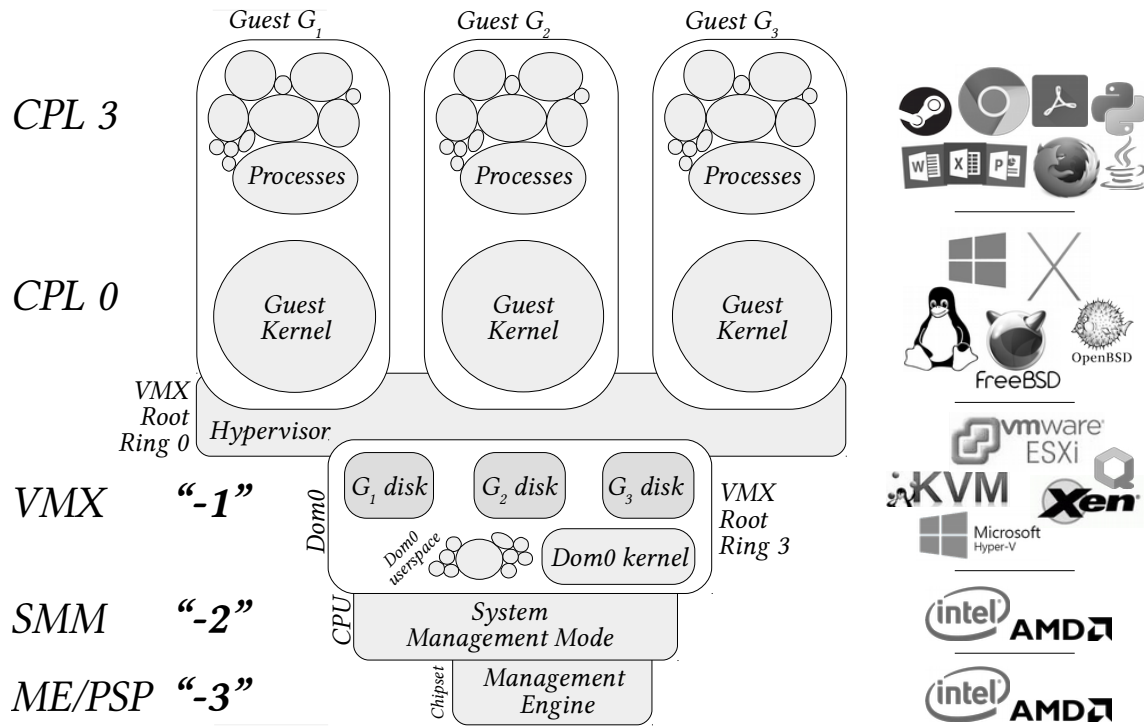


Figure 2.1 A native hypervisor diagram based on Xen. The guest OS and processes run in CPLs 0 and 3, isolated from other guests by a hypervisor using hardware-accelerated virtualization support. The hypervisor and its management domain (here called dom0) run in VMX root mode. Beneath the virtualization layer exist two other layers which can support security applications: System Management Mode and the Intel Management Engine. To the right of the diagram are currently available products found at each layer.

An example of hardware-accelerated virtualization is depicted in Figure 2.1. This figure shows a guest operating system's two privilege levels (ring 0 and 3) running above a VMX root hypervisor layer.³ Modern hypervisors typically leverage a privileged helper guest (labeled *dom0* in this figure after the name of the Xen hypervisor's privileged domain) to interface with physical devices, manage the hypervisor, and provide services to other guests.

From a security perspective, it is important to note two other entities “beneath” VMX mode. The first is another privileged CPU mode, System Management Mode (SMM), and the second is a off-CPU embedded computer called the Intel Management Engine (ME). Along with their AMD counterparts, both SMM and ME are discussed further in Section 2.5.

Virtualization abstractions. In the VMX architecture, the hypervisor retains control over the host's physical resources. Various abstractions of these resources are presented to guests.

Virtual CPUs. VMX introduces the concept of a *virtual CPU* (vCPU). When a guest is scheduled CPU time, the hypervisor maps a guest vCPU onto a physical CPU. The guest's execution then

³Although not technically accurate, VMX root is sometimes called *ring -1*.

continues where it left off. A vCPU's state and its configurable behaviors are stored in a corresponding hypervisor data structure called the *Virtual Machine Control Structure* (VMCS) [Int16].

Page Tables. Another problematic area for virtualization is *page tables*, the data structure used to translate virtual addresses into physical addresses. A fully-virtualized guest OS should generally not be allowed to know the actual physical memory addresses that a hypervisor has assigned to it [CD16]. A guest OS also assumes it is the sole manager of main memory. Therefore, hypervisors use a *shadow page table* that presents a virtual address space (addresses in this space are called *Guest Physical Addresses* (GPA)) to a guest OS such that the guest OS can act upon this address space as if it were physical memory. Guest process virtual addresses in ring 3 are therefore actually translated twice, once from the guest process's virtual address to the GPA using a guest-managed page table, and again from the GPA to the host's actual physical memory address using the shadow page table [Bha08; Ahn12]. Shortly after VMX was introduced, processor manufacturers introduced hardware acceleration for shadow page tables, generically called *Second Level Address Translation* (SLAT), but commonly referred to by its Intel name: *Extended Page Tables* (EPT).

Trappable Instructions. The hypervisor maintains exclusive control over certain x86 instructions that should not or could not be executed by the guest. Some of these guest x86 instructions unconditionally trap to the hypervisor, while others trap to the hypervisor based on certain conditions or configuration options set by the hypervisor [Int16]. As of this writing, on Intel CPUs there are currently 10 *unconditionally trapped* instructions that include VMX instructions and certain privileged operations against a CPU's translation lookaside buffers (TLBs). Furthermore, there are 33 *conditionally trapped* instructions, including trapping on writes and reads against a vCPU's control and model-specific registers (MSRs).

Aside from the special treatment given to the instructions mentioned above, the remaining x86 instructions encountered during guest execution run natively, directly on the CPU. This allows hardware-accelerated virtualization to closely approach the performance of non-virtualized systems. This arrangement is explained best by Michael Cohen, the author of the memory analysis framework Rekall:⁴

"A modern [VMX]-enabled CPU intimately manages a VM for the hypervisor, running it natively directly on the hardware, and the hypervisor is allowed to get a word in once and a while."

As will be discussed in the next section, the ability to "get a word in once and a while" is at the crux of VMI's ability to interpose upon guest execution.

Emulation hypervisors. Shortly before VMX became widely available, a machine emulator called QEMU [Bel05] was introduced. Over the next several years, QEMU and VMX took hardware virtualization down two different, oft-intersecting paths. Both *emulation* and hardware-accelerated (called *native*) hypervisors remain in use today, and choosing one over another has implications for both

⁴<http://www.rekall-forensic.com/>

hypervisor-based security and performance (discussed further in Section 2.4). This dissertation focuses on native hypervisors, and the following section describes virtual machine introspection techniques as they are applied to this platform.

2.2 Virtual Machine Introspection

Virtual Machine Introspection (VMI) is the external observation of a virtual machine's raw CPU, memory, and disk [GR03]. VMI is most often done without awareness or assistance from the guest operating system. VMI was introduced in 2003 in the seminal work by *Garfinkel et al.* [GR03], who also described the original VMI properties of isolation, inspection, interposition. There are several survey and SoK papers which provide a more exhaustive VMI background research [Bau15; Wan15a; Jai14; Sun15].

VMI runs contrary to a common assumption made by practitioners: the idea of *perfect isolation*. In the abstract sense, any savvy user of a virtualized system quickly develops an intuitive sense that there exists isolation between any two guests, and that some degree of isolation also exists between a guest and the hypervisor. It should also be equally obvious that this isolation has limits. After all, the guests are sharing the same set of hardware, including processors and RAM. However, the extent of this sharing is sometimes murky.

As might be expected, the hypervisor has inherent and automatic access at any point of a guest's execution. Furthermore, on native hypervisors (our focus), even though the guest code is running natively on the CPU, the hypervisor can use VMI techniques to “get a word in” and take control of the vCPU to take arbitrary action, including modifying the guest.

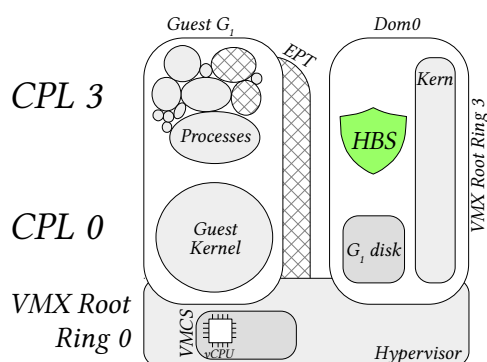


Figure 2.2 A VMI tool (shown as the green HBS shield) can access guest memory, vCPUs, and the guest's VMCS through mechanisms provided by the hypervisor.

Figure 2.2 depicts a commonly presented view of VMI on the Xen hypervisor.⁵ A guest is shown as a collection of processes in ring 3 and a guest kernel in ring 0. Independent guest process address spaces are managed by the guest kernel. The guest runs atop a hypervisor presenting the guest with virtual hardware resources. This diagram highlights the guest’s set of one or more vCPUs. Certain VMI techniques will modify vCPUs and the guest’s VMCS.

The figure also shows the hypervisor’s management domain, dom0, alongside the guest.⁶ Dom0 is typically a full Linux system, but for brevity the figure omits the variety of other programs running in its userspace. This domain is commonly used to host and execute a hypervisor’s VMI tools. The guest’s hypervisor-controlled EPT is shown behind the guest, and is used to translate guest physical memory addresses to hypervisor physical memory addresses. The EPT is crucial to certain VMI techniques.

Finally, the figure shows the guest’s virtual disk, which in the simplest case consists of one or more guest file systems residing in file-backed form (e.g., OVA, QCOW2, VMDK, or raw formats) in dom0. Depending on hypervisor implementation and configuration, the guest’s disk could also be stored on off-host storage (e.g., a storage area network) or in a dom0 disk partition.

2.3 VMI Primitives

The combination of four core VMI primitives enable a VMI tool to read from and write into a guest’s registers, physical memory, virtual memory, and disk. Each VMI tool uses one or more of these primitives to inspect or interpose on a guest.

The first primitive enables the fundamental VMI activity of reading and writing to a guest’s memory, CPU registers, and disk. This primitive can be performed on a guest at any time.

Because early works used this primitive to repetitively poll a section of guest memory for changes, they did not *interpose*, or intercept guest actions. Besides wasted cycles, this polling method left the VMI tool open to Time of Check Time of Use (TOCTOU) problems and furthermore could not detect transient events that occurred in between poll intervals [Wan15b]. Over time, security researchers developed more powerful VMI techniques, often as the result of clever re-use of processor hardware features, to precisely interrupt a guest at opportune times so that polling could be avoided.

The remaining three primitives center on techniques to perform this precise interposition on guest execution. Each has its own performance characteristics, granularity, and impact on a guest. These techniques are used to trap a guest’s execution, *stunning* [Sun15] (suspending) the guest at a precisely chosen location to allow a VMI tool to perform its purpose (e.g., to prevent an action or retrieve information from a transient guest state).

⁵Xen’s terms are used throughout this dissertation.

⁶In KVM, the management domain is the Linux host.

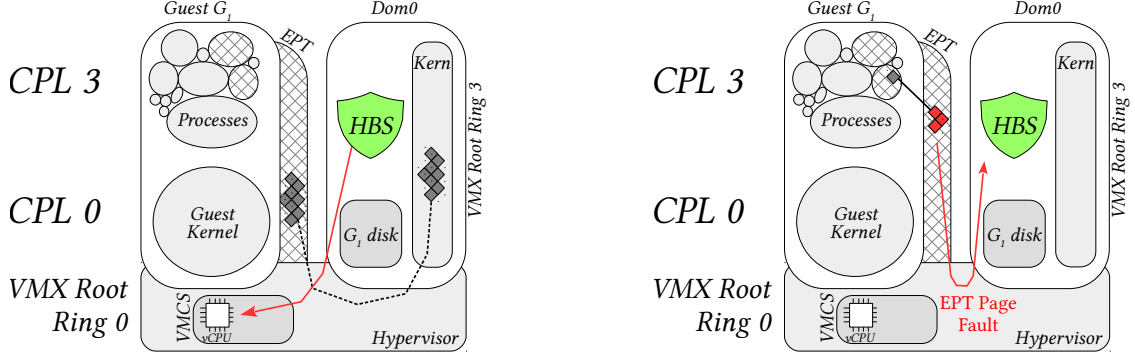


Figure 2.3 The guest Reads and Writes VMI Primitive (left) and the EPT violations VMI Primitive (right).

2.3.1 Reading and Writing

Guest memory. First used by *Livewire* [GR03], read and writing to guest memory is achieved when a hypervisor maps, shares, or copies guest memory into the domain performing VMI.⁷ Guest physical addresses can be mapped directly by the hypervisor, but guest virtual addresses must first be translated by walking the appropriate guest page table. This is typically done by the VMI tool, and certain VMI libraries have been developed to assist in this process^{8,9} [Pay07]. Recently, *Zhao et al.* [Zha17] demonstrated a new approach that avoids this software translation and instead performs it directly through the processor’s MMU, achieving a speedup that varies between 100x–1000x depending on use case.

Guest vCPU registers. To read and write to a guest’s vCPU registers, a hypervisor modifies a guest vCPU’s registers “in-place” on the corresponding VMCS.

Guest file systems. File system inspection is specific to the hypervisor implementation and its configuration, but often involves mounting a guest’s disk inside the inspecting domain. This is done without the guest’s knowledge, often while the guest is actively using the file system, and therefore the inspecting code must not modify the disk. *Atomizer* [Jav] and *VMI-HoneyMon* [Len12] both use *libguestfs*¹⁰ to launch a special-purpose *supermin appliance* (essentially an emulated Linux kernel) to access a guest’s disk. Other out-of-guest disk access methods include intercepting and interpreting a guest’s file systems and raw disk operations as was done by *vMON* [Li13], *DIONE* [MK12], open source tools such as *TSK*,¹¹ and *XenAccess* [Pay07]. A VMI tool that handles file systems directly must understand each file system format likely to be encountered. Alternatively, Chapter 5 of this dissertation, *KVMonitor* [KN14], and *V-Met* [MK17] use the *qemu-nbd* Linux kernel

⁷A discussion on the various implementations of this primitive is available [Sun15].

⁸<https://github.com/libvmi/libvmi>

⁹<https://github.com/razvan-cojocaru/libbdvmi>

¹⁰<http://libguestfs.org/>

¹¹<http://www.sleuthkit.org/>

module or dm-thin provisioning to mount the guest file system locally, transferring this burden to the inspecting domain's OS and its file system drivers.

2.3.2 EPT Violations

This primitive leverages the permission bits on the hypervisor-controlled Extended Page Table (EPT). By changing a Page Table Entry's (PTE's) permission bits on a guest's underlying EPT, a guest's attempt to access this page will result in a page fault that can be delivered to a VMI tool. Depending on the permission bits, this indicates to a VMI tool that a guest vCPU is attempting to read, write, or execute some contents on the page. Examining the vCPU in question will often reveal the exact location on the page.

HBS systems that perform guest integrity checking often leverage this primitive to invisibly mark certain vulnerable guest pages as read-only in order to prevent their potentially malicious modification.

Early applications of this primitive include performing malware analysis (*Ether* [Din08]), and enforcing in-guest memory protections (*SecVisor* [Ses07]). More recently, EPT violations were used to detect bugs in the Windows kernel [Pan17] and lock down the Linux kernel¹² [Han].

EPT violations allow a VMI tool to interpose at page-level granularity (typically 4096 bytes), which may be too coarse for certain applications. For example, if a VMI tool is only concerned with a small portion of a page's contents, the tool will still be notified of any guest activity elsewhere on the page. This was recently addressed by *Lu et al.* [Lu17], which found a technique to scope EPT violations to certain portions of a page.

Prior to the introduction of SLAT features, these type of violations were originally done on software-managed shadow page tables. SLAT improved this primitive's performance.

Recent VMX features have expanded a hypervisor's ability to maintain multiple guest EPTs. A pointer field called the Extended Page Table Pointer (EPTP) in a guest VMCS directs VMX to the current EPT a vCPU should use. By changing this pointer field, different EPTs with various permutations of permission bits can rapidly be swapped in and out underneath a guest. *Graffiti* [Cri16] used EPTPs to minimized performance impact when detecting heap sprays, and *DRAKVUF* [Len14b] used EPTPs to hide its injected breakpoints from a guest's self-inspection integrity scans such as Windows Kernel Patch Protection (i.e., PatchGuard) [Yos17].

2.3.3 Register Updates

A setting in the VMCS can be configured to instruct vCPUs to trap to the hypervisor when one or more conditional x86 instructions are executed on a guest. If one of these instructions is encountered, the processor notifies the hypervisor and the event is passed on to the VMI tool.

¹²<https://github.com/kkamagui/shadow-box-for-x86>

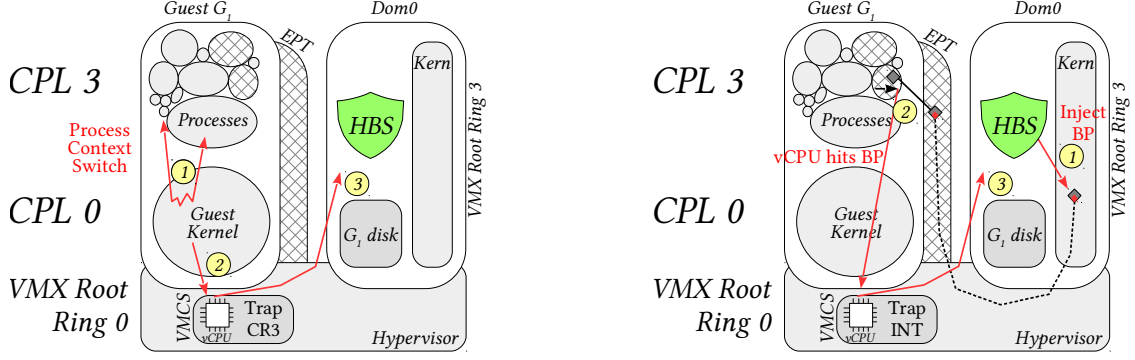


Figure 2.4 Register Events VMI Primitive (left) and Interrupt Injection VMI Primitive (right).

This capability can be used to trap writes to the CR0, CR3, and CR4 registers (among others), as well as reads to the various processor model-specific registers by trapping on the RDMSR instruction [Int16].

First explored by *Antfarm* [Jon06], and additionally used by *Patagonix* [Lit08], *HIMA* [Aza09], *Ether* [Din08], and *Panorama* [Yin07], register events are often used to capture guest process context switches (which use MOV to CR3). This primitive is used by Goalkeeper in Chapter 3.

2.3.4 Interrupt Injection

The final VMI primitive enables guest memory events at byte-level granularity, and can be used to trace the execution of individual instructions and function calls. This method is invasive in the sense it must make modifications to guest memory in order to install a software breakpoint (the INT3 instruction, 0xCC). When this breakpoint is encountered during execution, the guest's vCPU can be configured to trap to the hypervisor, allowing a VMI tool to take arbitrary action. This technique is essentially a hypervisor-based debugger.

Early works using VMI to inject breakpoints include *XenProbes* [QS07]. The primitive was refined and strengthened by *SPIDER* [Den] and *DRAKVUF* [Len14b], which inserted breakpoints at 11,011 Windows function calls, protecting the breakpoints with an EPT violation-based scheme. Debugging frameworks, such as *Stackdb* [Joh14], also use this primitive. This primitive is used by Arav in Chapter 4.

The high-level flow required to process an injected breakpoint in Xen is depicted in Figure 2.5.

2.4 VMI and Hypervisor Types

This dissertation focuses on native virtualization. However, VMI can be performed on both native hypervisors and emulation hypervisors. Choosing between these types has implications on VMI's

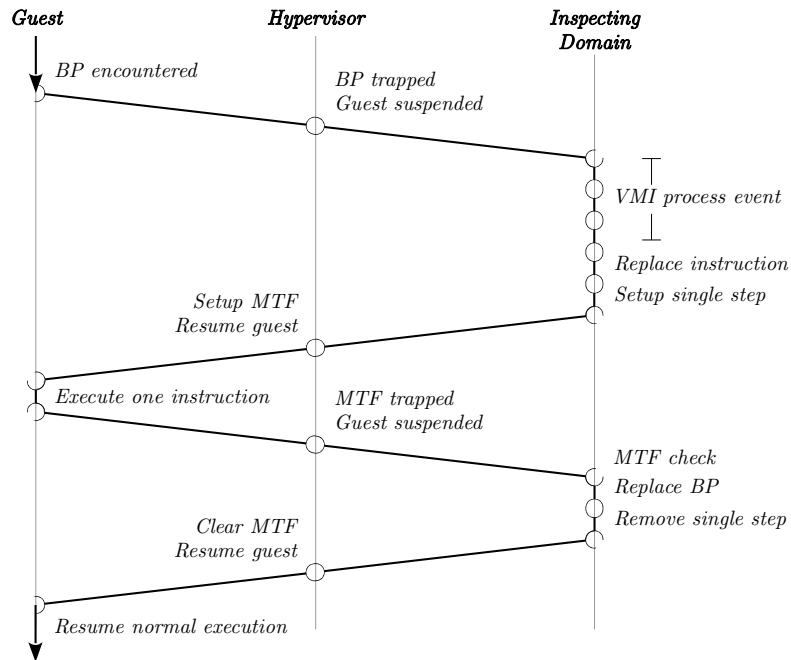


Figure 2.5 Processing an injected breakpoint. When guest execution encounters a breakpoint, the guest vCPU is suspended. The hypervisor passes this event to the inspecting domain’s VMI tool, which performs arbitrary action before removing the breakpoint and setting the vCPU to single-step mode. The guest vCPU single-steps across the breakpoint’s former location, and immediately traps back into the hypervisor so that the breakpoint can be re-injected.

level of granularity and control.

Classifying modern hypervisors into the original two hypervisor types introduced by Robert Goldberg in 1973 [Gol73] is often controversial. The traditional, often-taught “type” system does not apply cleanly to modern hypervisors, leading to confusion.

Type 1 “*The VMM runs on a bare machine*” [Gol73]. This is often interpreted as hypervisors containing a discrete VMM component such as Xen or ESXi, and is often misinterpreted to exclude KVM.¹³

Type 2 “*The VMM runs on an extended host under the host operating system*” [Gol73]. This is often misinterpreted to be limited to desktop virtualization products such as VMware Workstation, Parallels, or VirtualBox.

Instead, this dissertation adopts the hypervisor classification suggested by *Bauman et al.* [Bau15], which side-steps the original type system and focuses on the practicalities of *native* versus *emulation* hypervisors [AA06]:

¹³Much to the displeasure of KVM developers: <http://www.spinics.net/lists/kvm/msg150509.html>

Native “Native hypervisors ... directly push the guest code to execute natively on the hardware using hardware virtualization” [Bau15]. By this definition, native hypervisors include KVM [Kiv07], Xen [Bar03], ESX/ESXi [Wal02], Hyper-V [VV09], and bhyve.¹⁴

Emulation “Emulation hypervisors ... translate each guest instruction for an emulated execution using software virtualization” [Bau15]. Emulation hypervisors include QEMU [Bel05], Bochs [Law96], and Dynamips.¹⁵

Comparison. Emulation hypervisors offer a higher level of flexibility and interposition, allowing the execution of arbitrary hypervisor logic between any two guest instructions. Emulation also allows hardware invariants to be loosened. For example, since the guest CPU and MMU are emulated in software, their state can be changed arbitrarily, regardless of how these components would behave on silicon. This is in contrast to native hypervisors, which do not have this flexibility because guest code runs directly on the CPU.

As a result, emulation also imposes higher performance overheads when compared to native virtualization, and therefore is generally limited to use cases that can tolerate such limitations or specifically require emulation (e.g., running code written for another architecture). Additionally, since emulation must replicate the entire functionality of hardware in software, there are conditions in which emulated hardware behaves differently than actual hardware. As a result of the detectable differences¹⁶ during emulation, some malware samples test for these artifacts in order to prevent themselves from being analyzed and reverse engineered [SH12; Che08]. However, virtualization detection is certainly not unique to emulation hypervisors [Mir17].

Because of the arbitrary interposition allowed by emulation hypervisors, they are popular for granular malware analysis and security-related record and replay research, including the *BitBlaze* family of related research [Son08; Hen17; Yin07], *ReVirt* [Dun02], and *PANDA*¹⁷ [DG15].

Unlike emulation, native virtualization means a native hypervisor’s hardware invariants are truly immutable. Instead of arbitrary interposition, native hypervisors must use the primitives listed in the previous section to interrupt a guest. In return, native virtualization can leverage the associated performance gains of modern hardware because nearly all guest instructions run unmodified directly on the processor.

In addition to the work in this dissertation, the speed and isolation offered by native hypervisors is leveraged by *Nitro* [Pfo11], *Ether* [Din08], *DRAKVUF* [Len14b], and *Guesttrace*,¹⁸ among others.

Finally, Some papers, such as *BareCloud* [Kir14], cleverly use both virtualization types in concert to detect evasive malware.

¹⁴<https://wiki.freebsd.org/bhyve>

¹⁵<https://github.com/GNS3/dynamips/>

¹⁶<https://github.com/a0rtega/pafish>

¹⁷<https://github.com/panda-re/panda>

¹⁸<https://www.flyn.org/projects/guesttrace/>

2.5 Below the Hypervisor

This dissertation argues that the hypervisor is the ideal layer for certain types of security. For completeness, this section briefly describes System Management Mode (SMM), Intel’s Management Engine (ME), and AMD’s Platform Security Processor (PSP), three other privileged alternative “layers” present in modern x86-64 systems.

System Management Mode. SMM was designed to enable hardware-specific management features, such as motherboard fan speed and power management [Duf06]. A CPU enters SMM when encountering a System Management Interrupt (SMI). While in SMM mode, the CPU cannot be interrupted (even by non-maskable interrupts) and SMM’s section of main memory, called *SMRAM*, is inaccessible to other modes.

Because of its isolation, stealthiness, and privilege, *Souza et al.* [ST15] argues that SMM is in a good position for certain security features, such as verifying the integrity of the hypervisor layer. *HyperCheck* [Zha14] and *HyperSentry* [Aza10] make similar proposals.

Other work, *MalT* [Zha15] and *Spectre* [Zha13], go further to leverage SMM to address ring 0 and ring 3 security issues such as heap spraying and rootkit detection.

However, others argue that SMM may be inappropriate for certain types of security. Intel recommends that time spent in SMM should be limited to 150 microseconds [DK14], yet several SMM security proposals typically remain in SMM on the order of milliseconds [Zha14; Aza10; Zha13]. Additionally, every core on a system must enter and exit SMM together, causing serious performance overheads for software running in ring 0 and ring 3 on multicore systems [DK14]. As a result, some SMM-based systems operate at random intervals to trade overhead for probabilistic coverage [Lea17].

Intel Management Engine. The Intel ME is an embedded computer system built into modern Intel chipsets [Rua14]. The ME supports a variety of applications ranging from Digital Rights Management (DRM) to IT management. The ME allows access to a desktop, laptop, or server even if the system’s operating system is non-functional and has its own processor, RAM, and firmware. It can access the machine’s physical memory, and communicate on the host’s physical NIC without assistance from the host operating system. The ME also remains powered on in certain sleep modes via a direct connection to the power supply [CD16]. The Intel ME’s fundamental access to a system makes it a good candidate for security applications.

The ME is essential to various aspects of system security (and is fundamentally involved in bootstrapping the system’s main CPUs [CD16]). However, the ME itself consists of a low-power processor with a relatively small amount of RAM, making it a poor choice for the more frequent interposition described in this dissertation [Rua14]. The ME is also designed to only accept Intel-

signed firmware (rumored to be powered by *MINIX*¹⁹) making it out of reach for most use cases.

AMD Platform Security Processor. The PSP is built into all modern AMD processors and is responsible for processor-level security functionality. The PSP consists of an ARM core and cryptographic coprocessor, and, much like the Intel ME, has its own memory and access to system resources such as main memory and the network.²⁰ Distinct from the Intel ME, a unique AMD virtual machine hardening feature called Secure Encrypted Virtualization (SEV) uses the PSP to encrypt the main memory of virtual machines to protect against “benign but vulnerable” hypervisors [HB17; Sou16].²¹ This feature requires high-speed cryptographic hardware in the PSP. The PSP otherwise generally resembles the capabilities and processing power of the Intel ME and is unlikely to be suitable for the same type of security applications found in this dissertation.

2.6 Why Hypervisor-based Security?

This last section introduces the motivations for performing guest-centric security at the hypervisor, many of which have been echoed by prior work [CN01; GR05; Fu16].

2.6.1 The Isolation Security Argument

The isolation security argument for HBS consists of three related security generalisms: 1) smaller interfaces are generally safer [Jai14], 2) less code is generally safer [Jai14], and 3) attacks generally enter a system at ring 3 [Sol16]. Taken in combination, these generalisms arrive at the conclusion that an HBS system is more isolated from attacks than code running in ring 3 and ring 0. Because of this isolation, they are resistant to attempts to deactivate or subvert them, and are therefore better candidates for security applications.

Smaller interfaces. Smaller well-defined interfaces are often associated with higher security. In virtualization, the two relevant interfaces are the system call interface (allowing communication between ring 3 and ring 0), and the hypercall interface (allowing communication between a guest and the hypervisor). Depending on the operating system and hypervisor in use, there is an order of magnitude difference (or more) in the number of system calls versus hypercalls [Jai14]. Therefore, the smaller interface between a guest OS and the hypervisor makes it theoretically easier to defend. Finally, since not all systems have virtualization enabled, this interface may be given less attention (but still some attention [Wil16; Gor15]) by threat actors because it is less ubiquitous.

Smaller code size. Compared to an operating system kernel, a hypervisor’s smaller code base is also associated with higher security, based on the argument that less code means less bugs. This

¹⁹<https://www.cs.vu.nl/~ast/intel/>

²⁰https://support.amd.com/TechDocs/52740_16h_Models_30h-3Fh_BKDG.pdf

²¹<https://lkml.org/lkml/2016/8/22/960>

argument stands up well in studies which compare source code size with number of vulnerabilities found [Jai14]. Additionally, the many thousands of third-party device drivers are also a major OS security concern [Yos17], as buggy drivers are a major source of vulnerabilities in an otherwise reasonably secure OS. In essence, a modern monolithic operating system kernel can only be as trusted as its least trustworthy driver.

Direction of attack. Vulnerability statistics and industry studies generally support the fact that most attacks target software running in ring 3 [Sol16]. This argument combines observations of attack statistics, ring 3 and ring 0 code sizes, and the combination of vulnerabilities and their exposure in order to conclude that ring 3 is where most attackers initially land on a host. As compared to the relative order, conservative programming, and thoughtful consideration given to operating system implementations (and even they get it wrong occasionally²²), there is both more code running in userspace and code with less quality controls applied to it.²³ So when a vulnerability in a userspace process—whether it be a web browser, document reader, or music player—inevitably gets exploited, an attacker gains a foothold on a system in the form of user-level privileges. Third-party applications are not the only processes running in ring 3. Operating system daemons, commonly found in Windows and Linux, also run in ring 3. These services are valuable targets because they are ubiquitous (i.e., always running on that flavor of OS) and have higher OS-level privileges (i.e., the jump to ring 0 is easier).

Discussion. For an attacker to get from ring 3 into the operating system kernel in ring 0, they typically need a second exploit that escalates them to administrative privileges or exploits code on or behind the system call interface.

Finally, for an attacker to break out of a virtual machine and onto the hypervisor in VMX root mode, they need a third exploit to leverage a vulnerability on or behind the hypercall interface or through one of the virtual devices that depends on it [Wil16]. These are rare and especially valuable considering they could be used by an attacker to seize control of a public cloud.²⁴

While these three generalisms are generally accepted as true, it must be simultaneously accepted that all sorts of other things can go wrong at each one of these levels.

First, bugs in SMM or ME, such as injecting code into SMM through implementation flaws,²⁵ or logic bugs used to bypass ME authorization²⁶ mean that attackers can avoid fighting downward through the rings and simply *directly* enter a privileged mode. Unlike coming in from ring 3, gaining access at these levels grants the attacker full access to the layers above it.

Second, interfaces themselves are weak in certain areas. For example, the `ioctl` system call is

²²<https://outflux.net/blog/archives/2016/10/18/security-bug-lifetime/>

²³<https://web.archive.org/web/20140223013701/http://blog.james.rcpt.to/2012/02/13/debian-wheezy-us19-billion-your-price-free/>

²⁴<http://venom.crowdstrike.com/>

²⁵<http://theinvisiblethings.blogspot.com/2008/08/intel-patches-q35-bug.html>

²⁶<https://nvd.nist.gov/vuln/detail/CVE-2017-5689>

often misused^{27,28} [Xue08] by security vendors seeking to enable communication between their anti-virus user interface in ring 3 and their kernel mode driver in ring 0. It is ironic that a security vendor's poorly implemented input validation can actually make it *easier* for an attacker to escalate to root.

In conclusion, the HBS isolation security argument concludes that HBS is well suited to defend against attacks coming in from ring 3 and ring 0, allowing that these are only a subset of every possible attack a system might face.

2.6.2 Other Arguments

In addition to the isolation security argument, there are other practical reasons for implementing guest-centric security at the hypervisor layer. In industry, these properties have led to HBS-like system being called the *Goldilocks zone* for data center security.²⁹

Privilege. The virtualization layer is the first layer more privileged than the guest kernel. Using reference monitor concepts [And72] (a reference monitor must be *always invoked*, and *tamper-proof*, and *verifiable*), it can be argued that due to the isolation between a guest and a hypervisor, a hypervisor-based reference monitor would have better security properties than one based in the guest kernel. This benefit is commonly mentioned throughout VMI literature and is leveraged throughout this dissertation.

Visibility. The virtualization layer is the first layer that has access to every guest's memory, allowing it to *look across* two or more guests at the same time. This can allow HBS systems to perform work more quickly and effectively and is discussed in depth in Chapter 5.

Relative performance. The virtualization layer is the lowest level in a system where reasonably high performance remains achievable. While security functionality running in this layer must still interrupt the guest kernel, the performance hit is not as great as it is for the SMM layer [DK14]. Additionally, while the Intel ME may be suited for certain types of security, it has a relatively slow processor ("hundreds" of MHz, compared to several GHz for main CPUs [Rua14; CD16]). Chapters 3 and 4 discuss this benefit further.

Consolidation. If duplicate security functionality can be removed from adjacent guests and consolidated at the hypervisor in the form of an HBS system, efficiency and centralization benefits can be realized. For example, a single hypervisor-based agent can use less RAM than the combined usage of its in-guest counterparts. Additionally, a single centralized agent can be more easily administered than an agent in each guest. These benefits are explored further in Chapter 5.

²⁷<http://securitytracker.com/id/1020196>

²⁸<http://www.securityfocus.com/bid/20360/info>

²⁹"Not too hot, and not too cold." <https://www.datacenterknowledge.com/archives/2014/04/15/virtualization-security-goldilocks-zone>

traverse object pointers on the kernel heap (much like the kernel does) until reaching the desired kernel object.

Crossing VMI's semantic gap has become its own area of research. A recent survey paper [Bau15] organized gap-crossing techniques into five categories: Manual, Debugger-assisted, Compiler-assisted, Binary analysis, and Guest-assisted. Of these, manual is used most frequently. All the VMI techniques in this dissertation use manually derived and debugger-assisted techniques.

Because forensics practitioners routinely cross this semantic gap during the analysis of forensic samples such as memory and file system dumps, forensic tools also represent a source of techniques to solve this problem. VMI's integration with *Volatility*³⁰ and *Rekall*³¹ [DG11a] combined these two research areas, allowing forensic tools to operate against live virtual machines through VMI.

In 2014, *Jain et al.* [Jai14] laid bare several issues with how modern VMI tools (mis)handle the semantic gap. In essence, to cross the gap, modern tools have implicitly trusted the guest operating system to faithfully observe the properties it had when the semantic mapping was generated. In the case of leveraging debugging symbols to cross the semantic gap, VMI work (including ours) assumes that the data structure layouts described in the symbols (generated at guest kernel compile time) are identical to the data structures actually in use by the guest OS. Prior work has shown this assumed trust to be exploitable [Bah10; Len14a]. Attackers seeking to exploit VMI tools in this manner anticipate (or guess) the kernel objects a VMI tool is interested in, and then deliberately manipulate the object such that its fields present misleading or benign information. *Jain et al.* concludes that the semantic gap actually consists of two parts, a solved engineering problem called the *soft semantic gap*, and an open security problem called the *strong semantic gap* [Jai14].

The *strong semantic gap* is not directly addressed by this dissertation. This work uses mitigation techniques to reduce the threat, including careful input validation on raw guest memory, sandboxing, and behavioral whitelisting.

OS-specificity. To cross the semantic gap, a guest's abstractions must be reconstituted by the VMI tool. Because these abstractions are often OS-implementation-specific, any semantic mappings used by the VMI tool are fragile and may not apply to other operating systems. In Chapter 3, we briefly investigate the degree of specificity of Goalkeeper's semantic maps. Ultimately, the changes from one kernel version to the next (we termed this semantic map *kernel drift*) are dependent on circumstances, as certain areas of the Linux kernel are very stable and are not expected to change often from version to version.

Performance. VMI is often slower than analogous in-guest code. VMI must cross the semantic gap, which introduces an additional computational step. This overhead exhibits itself in various ways, from the slowdown in software-based guest virtual address translation to the extra VMI machinery that processes the semantic mapping. In the case of interposition, a guest vCPU is often suspended

³⁰<http://www.volatilityfoundation.org/>

³¹<http://www.rekall-forensic.com/>

while VMI mediates the event, reducing the vCPU's productive work. Finally, overhead is introduced by CPU mode switches in and out of VMX mode, which occur more frequently when a VMI tool is using an introspection primitive.

CHAPTER

3

HYPERVISOR-BASED PROCESS ENFORCEMENT

In the next two chapters, two scalable, efficient HBS systems are described. Each solves an important security problem using designs that scale horizontally to many guests in parallel and minimize in-guest performance penalties. First, this chapter introduces *Goalkeeper*.

Goalkeeper’s security objective focuses on in-guest process whitelisting. Controlling when and how a process runs is essential to the security of a system. In virtualized environments, an out-of-guest approach to process control is attractive because it allows fine-grained in-guest inspection and enforcement from the relative safety of the hypervisor, which makes in-guest misconfiguration by users or deliberate interference by malware more difficult. However, prior work in this area is incomplete, either lacking policy enforcement, missing certain types of malicious code due to insufficient coverage, or being unable to scale to many simultaneous guests. Goalkeeper is a hypervisor-based security system that focuses on asynchronous, stateless, and lightweight Virtual Machine Introspection (VMI) techniques to enforce comprehensive guest process security policies *at scale* across tens to hundreds of guests per hypervisor. Running beneath each guest, Goalkeeper uses policy rules to ensure only whitelisted guest processes are allowed to execute, and terminates policy violators using a customizable set of VMI-based process termination techniques. In an evaluation across a population of 100 Linux virtual desktops, Goalkeeper is shown to catch malicious code that is missed by prior work while imposing a comparable performance overhead.

3.1 Introduction

The ability to control which processes run, and how they run, is a fundamental aspect of system security. In practice, controlling processes often means vetting them at startup (e.g., code signing, whitelisting, and blacklisting) and testing them during runtime (e.g., host intrusion detection). Although many implementations of these approaches exist, nearly all depend on the integrity of a modern operating system’s monolithic kernel, where any vulnerability in the kernel can result in complete compromise and subsequent bypassing of these security features.

In response, prior work has moved security components to the hypervisor, including traditional application whitelisting [HC14], page-level binary code signing [Lit08], heap allocation monitoring [Jav; Len14b], and interactive system call redirection [FL13; Wu14]. However, these implementations are limited in terms of when and how they inspect in-guest processes, as well as the granularity and flexibility of the policy language used to specify a process’s allowed behavior. Additionally, performance overheads often limit prior work to use cases for which high overheads are acceptable (e.g., dynamic malware analysis).

Migrating security components to the hypervisor provides security, efficiency, and manageability benefits. The hardware-enforced boundary between a hypervisor and its guest grants a hypervisor-based security system independence from the guest OS as well as a degree of concealment from potential guest interference. Furthermore, from a manageability perspective, installing, configuring, and managing a single security component spanning all guests is more attractive than a security component inside each guest.

This chapter describes a more complete, more scalable Xen-hypervisor-based process supervision system called *Goalkeeper*. Goalkeeper passively identifies in-guest processes using a policy language specification based on an extensive set of process attributes, ranging from high-level administratively-friendly attributes (e.g., process name, username, and executable file path) to strong hardware-invariant-based attributes (e.g., code hashes and page table state). To exert control on guest processes and prevent or terminate their execution, Goalkeeper resorts to an escalating sequence of VMI techniques that gracefully shuts down a process in conjunction with the guest OS, or, alternatively, unilaterally destroys the process without guest cooperation.

Goalkeeper was evaluated against userspace malware and a return oriented programming attack, and its performance was tested on a hypervisor concurrently hosting 100 Linux guests. Using automatically-generated policy rules, Goalkeeper terminated all unauthorized processes without any false positives, and terminated processes infected by malware that would be undetected (i.e., allowed to execute) by previous hypervisor-based approaches. Goalkeeper’s in-guest performance overhead at scale on 100 busy guests (3.2 new processes/sec) is comparable with (single guest) prior work, and its resource costs on the hypervisor are acceptable.

Goalkeeper’s scalability and flexibility demonstrate that hypervisor-based security controls are

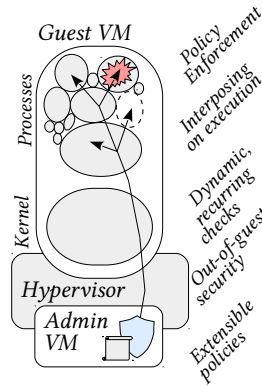


Figure 3.1 Guest Process Supervision

a valuable complement to guest operating system security.

This chapter makes the following contributions:

1. *The design and evaluation of Goalkeeper, a scalable, hypervisor-based guest process supervision system.* Goalkeeper interposes on guest process context switches, and is shown to perform well in hypervisors with high guest densities.
2. *The introduction of a class of simple, asynchronous guest memory manipulation techniques used to terminate guest processes.* These techniques alter guest behavior by making small modifications to guest memory.
3. *A flexible policy specification based on process attributes.* These policies allow Goalkeeper to protect against a wider set of malicious behavior than prior work, and provide a high degree of extensibility for whitelisting.

3.2 Motivation

This chapter pairs efficient VMI-based process inspection and termination techniques with a comprehensive process-centric security policy¹ on in-guest processes, termed *guest process supervision*. Guest process supervision inspects in-guest processes to ensure that these processes are both allowed to run *and* are in a known-good state during runtime (the general process is depicted in Figure 3.1). The objective is to prevent unauthorized code execution initiated via an unintentional user action or remote attacker. The remainder of this section and Table 3.1 compare relevant approaches in this area and highlight the requirements for a more complete approach.

An early example of detecting modified programs was Tripwire (succeeded by AIDE²), which compared a hash of each program to a hash taken in the past. If the hashes did not match, the

¹A set of rules describing constraints on process attributes.

²<http://aide.sourceforge.net>

Table 3.1 Process Supervision Comparison

		Out-of-guest	Interposing	Dynamic	Enforcing	Extensible	Scalable
	In-guest FIM	○	○	○	○	●	●
	<i>AppLocker</i> , <i>Gatekeeper</i> , <i>SELinux</i> [Los98]	○	●	○	●	●	●
	<i>Livewire</i> [GR03]	●	○	●	○	○	?
	Syscall Redirection [FL13; Wu14]	◐	○	◐	●	○	○
	<i>Patagonix</i> [Lit08]	●	●	○	◐	○	?
	<i>CLAW</i> [HC13]	●	●	○	◐	○	?
	<i>Goalkeeper</i>	●	●	●	●	●	●
Filled circles are better. Partial fill indicates marginal applicability. "?" indicates a system lacking a scalability evaluation.							

program had been changed between measurements, prompting further investigation. However, while Tripwire and other File Integrity Management (FIM) tools remain a security best practice, they are limited to detecting malicious code stored on a file system.

Concurrently, operating systems began to include process-centric security controls such as *Windows AppLocker*, *MacOS Gatekeeper*, and *SELinux* [Los98]. By virtue of their direct integration into kernel pathways that manage processes, these approaches have the benefit of *interposing* on process execution, allowing them to measure and compare system state against a policy before allowing an action to continue. These approaches are also *extensible* because they support complex policy rules and are built to integrate into a larger security architecture. Unfortunately, as part of a larger monolithic kernel, these approaches can also be deactivated since they are within reach of malicious code that has exploited a kernel vulnerability.

For this reason, providing protection from outside the OS has been proposed, namely by use of virtualization. Performing security functions at the hypervisor, sometimes called an *out-of-guest* approach, provides several benefits. The hardware-enforced isolation and smaller exposed interface between a hypervisor and its guests is more difficult to subvert than the the syscall interface between a user process and its kernel [GR03; Pay08]. Therefore an out-of-guest security component is less likely to be tampered with or disabled by the user or malware.

The earliest work on out-of-guest security was *Livewire* [GR03]. *Livewire* ran in the hypervisor and used a modular VMI-driven framework that inspects guests for signs of compromise. *Livewire*'s implementation was *dynamic*, meaning it was designed to monitor behavior periodically during execution. However, *Livewire* did not prevent or disrupt execution if problems were found, nor did all of its policy modules interpose upon guest execution, instead relying on polling.

In contrast, *Patagonix* [Lit08], a hypervisor-based system that compared in-guest executable code to a known-good database, could interpose upon guest execution, allowing it to detect malicious

code about to execute. Cloud Application Whitelisting, or *CLAW* [HC13] was a similar hypervisor-based executable code-hashing system. Both systems included rudimentary enforcement that could prevent execution of non-whitelisted processes. However, like OS-based whitelisting implementations, *CLAW* limited its checks to certain events such as process startup and dynamic library loads. Furthermore, like Patagonix, *CLAW*'s whitelisting ruleset was essentially a list of binary hashes, limiting *CLAW*'s ability to detect attacks such as return-oriented programming.

Syscall redirection research, such as *VMST* [FL13] and *Shadow Context* [Wu14], while not originally envisioned for process supervision, contains the primitives required to inspect and enforce in-guest processes. In this approach, system calls are redirected from a trusted VM into an untrusted VM, allowing access to in-guest context from an out-of-guest approach. However, syscall redirection needs both an additional degree of guest cooperation via a small in-guest agent and a companion guest with similar characteristics running alongside the guest under inspection.

Each of these systems include some degree of policy specification to define allowed behavior. These specifications can include hard-coded integrity checks [GR03], a simple list of hashes [HC13], or a complete policy language [Los98]. Intuitively, an expressive and descriptive policy specification allows for more granular rules and ultimately reduces the chance for false positives (benign processes labeled as malicious) and false negatives (malicious processes labeled as benign). Of the out-of-guest approaches, none support an extensible policy language.

Finally, prior work is often not evaluated thoroughly with respect to scalability. If a system is designed with scalability in mind, it can be shown to efficiently support many simultaneous guests in terms of performance overheads, response time, and CPU and memory resource costs. This is important because modern hardware allows hypervisors to support tens-to-hundreds of guests. Scalability in particular has been identified as an under-explored issue in VMI, and non-scalable solutions have hampered adoption [Jai14].

In summary, this chapter will present a higher-quality out-of-guest approach with the following attributes:

- *Out-of-guest*: Inherently resists guest evasion attempts.
- *Interposing*: Intercepts processes before they execute.
- *Dynamic*: Re-inspects a changing guest process.
- *Enforcing*: Terminates a process for violating its policy.
- *Extensible*: Uses a flexible, integrated policy language.
- *Scalable*: Runs cheaply at high guest densities.

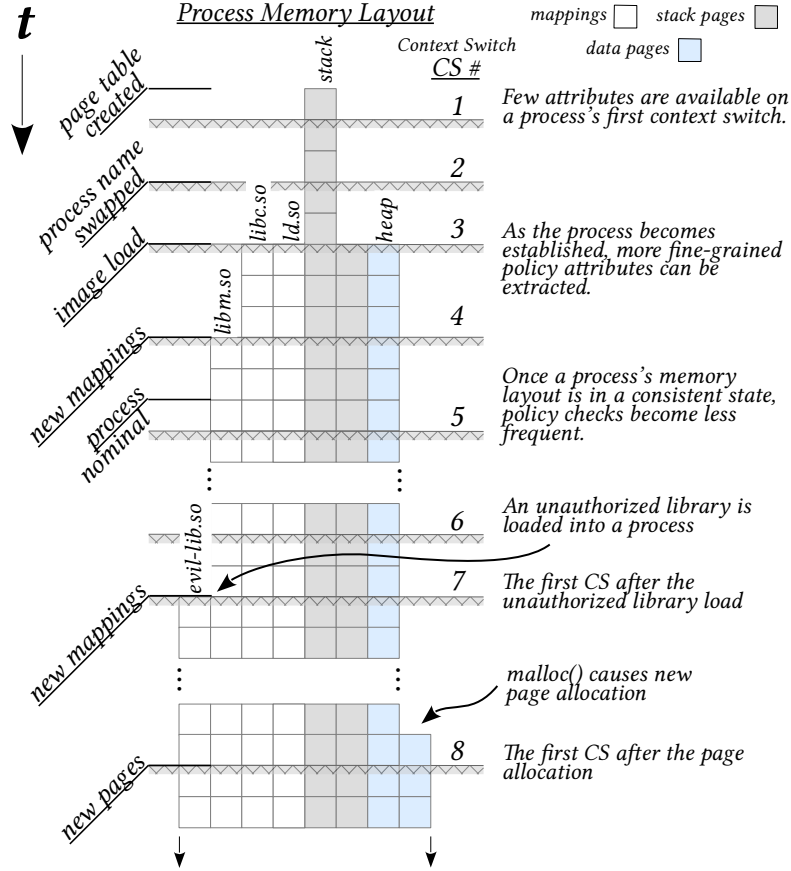


Figure 3.2 Process attributes are manifested as a new process is created and initialized. Over time (from top to bottom), the process's memory layout reflects new dynamically loaded libraries (CS 4, 7) and new page allocations (CS 8).

3.3 Background

This section introduces the hypervisor-based primitive used in this chapter, *MOV-to-CR3* interception, and describes how this method observes the initialization of new guest processes.

There are several hypervisor-based techniques [Wan15a] that leverage hardware virtualization extensions to observe and interpose upon a guest's activities, including the observation of guest vCPU register changes (like the CR3 register) [Aza09; Din08; Lit08; Yin07], injecting breakpoints into guest memory [Len14b], or manipulating a guest's extended page table (EPT) permissions [Din08].

First explored for VMI by *Antfarm* [Jon06], the x86 CR3 register contains a pointer to the guest vCPU's current page table. Each non-kernel process has a distinct virtual address space, so the page table pointed at by the CR3 register can be used to uniquely identify a process. A guest kernel writes to this register, called a *MOV-to-CR3* operation, when changing page tables. *MOV-to-CR3* events are

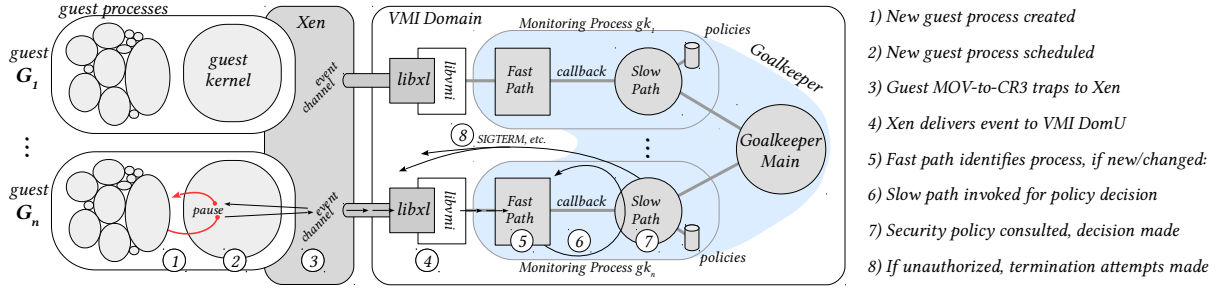


Figure 3.3 Goalkeeper spawns a monitoring process for each guest virtual machine to be monitored. Each monitoring process consists of a fast path and slow path, and both are driven by its corresponding guest’s process context switches. The guest process being “switched to” is subject to Goalkeeper enumeration, inspection, and potential termination if it fails its policy check.

simple, commonly used, occur at comparatively moderate rates, and are resilient to false negatives (i.e., failing to notice new guest processes).

New processes are brought to life over a few context switches, and intercepting *MOV-to-CR3* operations allows a hypervisor-based security system to observe and intervene on process initialization. In addition to the guest OS updating kernel objects to account for the new process, process initialization also occurs inside the new process’s virtual memory space. These changes include swapping out the parent’s executable for the child’s, loading dynamic libraries, and allocating memory areas such as the heap and stack. Figure 3.2 shows a more detailed view of this procedure for Linux.

For the typical `fork()` and `execve()` families of system calls, the child initially inherits a subset of its parent’s attributes (see Figure 3.2’s CS 1). Here, the new process generally resembles its parent and has a single private page (its stack) in the child’s new address space. At this stage, the only information available pertains to the process’s parent, but this is enough to begin making and enforcing policy decisions, especially if the parent is not allowed to have children.

Subsequent trips to the CPU allow the inspection of additional process attributes. As time goes on, the hypervisor can build an increasingly well-informed picture about the new process, from its new name (CS 2) to its executable file path and shared libraries (CS 3). Even before a process executes its first bit of useful code, a hypervisor-based system has had two or more “looks” (intercepted *MOV-to-CR3* operations) to ensure it is allowed to run.

The same technique allows ongoing, dynamic inspections on stable, running processes. Arbitrary library loads (CS 7) and heap allocations (CS 8) that have occurred since the last context switch can be detected, measured, and used in policy decisions. Therefore, if a process is compromised at runtime, this technique can also be used to detect changes resulting from the compromise.

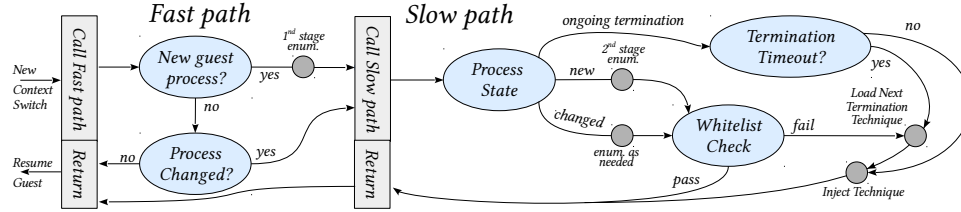


Figure 3.4 Goalkeeper's high-level state machine. Existing, stable processes are filtered out and quickly return execution to the guest. New or changed processes are enumerated and checked against their corresponding policy.

3.3.1 Threat Model

This chapter considers two types of security-related scenarios. First, enterprises in many cases wish to restrict the software that can be run on computers that are part of the organization. Employees and clients sometimes attempt to install and execute unauthorized software, in contradiction to this policy. Enforcement of the enterprise's policy, by means of a process security policy, is a solution.

Second, in most environments, including virtualized guests, hosts are constantly subjected to attacks. The goal of these attacks is to exploit vulnerabilities in a host's processes or operating system in order to establish adversarial control and persistence. Defending against these attacks requires a characterization of normal process execution, and runtime monitoring of process behaviors.

3.3.2 Assumptions

This chapter trusts that the hardware and hypervisor are reliable, and not subject to control or corruption by an adversary (a commonly adopted assumption [Jai14; Lit08; HC13; GR03]). The initial installation of VMs is further assumed to be uncorrupted.

To fully function, this approach assumes certain guest kernel data structures are properly built and maintained during runtime. While sanity checks are performed to ensure these structures are well-formed and reasonable, this chapter is focused on guest userspace processes and does not claim to protect the guest kernel.

3.4 Design

This section introduces Goalkeeper, a hypervisor-based system that intercepts the execution of guest processes running in a virtualized environment and evaluates them against a multifaceted security policy. The first area of focus is flexibility; Goalkeeper makes decisions by comparing a guest process's attributes against a security specification that allows a variety of process attributes to be examined and used for whitelisting. If Goalkeeper decides a process is unauthorized, its policy language allows a flexible specification of the steps to be taken.

Goalkeeper is also designed to do this across many guests simultaneously with modest performance overhead for the guests and hypervisor. This effort is centered on a tight, hierarchical design which inspects a guest process only when warranted, and terminates unauthorized processes by making near-stateless, asynchronous changes to guest memory.

This section begins with Goalkeeper's concept and overall state machine (§3.4.1). Goalkeeper's identification and evaluation of guest processes is then discussed (§3.4.2) along with the methods to prevent and terminate guest processes that fail the policy check (§3.4.3). The section concludes with implementation details (§3.4.4).

3.4.1 Goalkeeper's Concept and State Machine

Goalkeeper's overall design is shown in Figure 3.3. Goalkeeper operates from a privileged Xen guest (dom0 or VMI domU) and spawns one child monitoring process per protected guest. Each monitoring process interacts with its assigned guest through Xen hypervisor APIs. This monitoring process loads the policies appropriate for its guest, and registers itself with the hypervisor to receive notifications when this guest schedules processes (via *MOV-to-CR3* interception). When a guest process is scheduled (steps 1-2), the event is trapped by the hypervisor (steps 3-4) and relayed to Goalkeeper's fast path (step 5). Goalkeeper's hierarchical design is meant to minimize overhead, so Goalkeeper will only inspect a guest process if it is new or has changed its memory layout since the last time it was scheduled to execute. If so, Goalkeeper's slow path is called, which extracts a full set of guest process attributes and compares them against the process's security policy (steps 6-7). If the policy check fails, the guest process is prevented from further execution via a series of efficient termination attempts (step 8).

Goalkeeper's state machine (Figure 3.4) shows additional detail on actions performed by each component. The fast path identifies and discards from further consideration context switches that occur for existing, stable processes. A guest process is considered stable if its total number of pages and mappings has not changed since the last time it was inspected by Goalkeeper. In general, process memory layouts will change when dynamic libraries are loaded during runtime, new memory is allocated on the heap, or shared memory regions are added or removed.

3.4.2 Process Identification and Vetting

Guest processes for which memory pages and mapping have changed since the last inspection will be further examined by the slow path. The slow path compares extracted guest process attributes against a security specification.

Goalkeeper's policy language allows administrators to specify per-process whitelisting rules. Each rule consists of a set of whitelisting conditions that test process attributes (supported attributes are listed in Table 3.2). Each condition must be satisfied for a process to satisfy the whitelisting rule.

Table 3.2 Policy Rule Components

<i>Extracted by Fast Path</i>		<i>VMI Reads Re-quired</i>
1.	Process's name	1
2.	Process's executable name with path	$2+2w$
3.	Process owner's user ID	2
4.	Process's parent name	3
<i>Extracted by Slow Path</i>		
5.	Process arguments	6
6.	Shell/env. variables	6
7.	Process's priority	4
8.	Process owner's username	6
9.	Executable hash	$6+2y$
10.	Libraries w. code pages & hashes	$6+(4x[+2y])z$
11.	Number and size of anon. mmap areas	(same as 10.)
12.	Number and type of context switches	2

w = Absolute file path depth (e.g., `/usr/bin/zsh` = 3)
 x = $2+2w$
 y = Size of text area in pages, [if policy contains hashing]
 z = Number of loaded SOs

Prior work has often used only memory hashing [HC13; Lit08] for whitelisting purposes. Goalkeeper can make use of a much more extensive set of process attributes. This offers advantages in terms of precision, speed, and security. For instance, Goalkeeper is able to detect security policy violations that do not modify executable code.

The information on which whitelisting rules are based can be collected *statically* from executable files, *dynamically* during process execution, *manually* by security administrators, or a combination of these.

Generating quality security policies is a hard problem, and has been an area of research for several decades [SS94; May06]. The approach taken in this chapter to generate the security policy was effective and straightforward. Every process that was to be whitelisted was inspected in order to extract process entry point information, hash values, and the executable's file path. To capture attribute information from running processes in several different states, several VMI-based process attribute extractions were performed, and these scans were combined to form the remainder of each process's attributes, including arguments, environment variables, its owner, and loaded shared libraries. In all, Goalkeeper's policy set consisted of static information about every executable, and enhanced runtime information for the processes running at the time of dynamic measurement.

Some process attributes are more useful than others for whitelisting purposes, and the work required to obtain process attribute values varies as well. For example, obtaining a process's name (stored as a character array in guest memory) is trivial, but easy to manipulate or forge. In contrast, hashing the contents of executable areas of memory is more expensive, but much harder to manipulate or forge an incorrect result. Table 3.2 compares the costs (number of VMI reads) for collecting

Table 3.3 Memory Manipulation Techniques

<i>Technique</i>	<i>Ignorable</i>	<i>Graceful</i>	<i>Permanent</i>	<i>VMI Reads Req'd</i>	<i>VMI Writes Req'd</i>
Priority Manipulation	no	-	no	1	1
Handler Manipulation	no	-	no	$1+2a$	a
SIGTERM Injection	yes	yes	no	$2+2b$	b
SIGKILL Injection	no	no	no	$2+2b$	b
Stack Corruption	no	no	no	$3+3c+d$	d
Page Corruption	no	no	yes	$7+2f$	e

a = Length of signal handler array (64 at time of writing).

b = Number of process's existing pending signals (usually 0).

c = Stack VMA location (usually 8-10).

d = Size of stack in pages (usually 34).

e = Size of exe in pages.

f = Absolute file path depth (e.g., `/usr/bin/zsh` = 3)

the value of each attribute used in this study.

Goalkeeper's policy language also describes actions to take when a process fails to satisfy a whitelisting rule, such as alert logging, external notifications, how the process should be terminated, or the modification of adjacent rules.

The following is an example of a whitelisting rule that restricts the `top` program to be executed only by one of three users, to be spawned by either the `bash` or `zsh` shells, and only with a restricted set of command line arguments. The rule specifies which shared libraries may be loaded, and executable page hashes that are enforced once the process is stable.

```
[/usr/bin/top]
arguments      = "top", r"top -d [0-9]+"
comm           = top
username       = bob, alice, charlie
parent         = /usr/bin/bash, /usr/bin/zsh
children       = 0
entry-point-hash = 9b94143f1085c973c11ef6df94424e8ad2fa...
page-hash      = {
    0: eca47b3bb7ad4afe6687f84224a8df127f1f596a6cf3579e3...,
    1: 81aa74eec166e7f647073ab059b4c5667f4f02dedc00564ba...
}
shared-libs     =
    /usr/lib64/libsystemd.so,
    /lib64/libc.so.6: { 33: 9627e679dfb03975126a04c0d84... },
    /lib64/libdl.so.2
exec-pages      = 1600-1650
stack-pages     = 34
options         = {
```

```

policy-violation-sequence = sigterm, corrupt-stack
log                       = sendToSIEM
}

```

This example was generated by measuring a running top process inside a guest. As such, the policy captures a snapshot in time and does not necessarily represent every potential state during normal execution. Providing a range of values for certain components (e.g., *exec-pages*, *stack-pages*, etc.), and including those values in the policy specification, may reduce false positives during whitelisting.

Appendix A fully describes the security policy grammar and how policies are applied.

3.4.3 Process Termination

If a guest process's attributes do not match the specified values, Goalkeeper by default initiates a sequence of actions to stop process execution by manipulating the guest kernel into degrading or terminating the unauthorized process. These memory manipulation techniques are listed in Table 3.3.

These techniques leverage well-known kernel and process components. However, prior work has proposed a single termination technique [HC13; Lit08] or simply ignored the issue of termination [GR03]. Having multiple techniques is important for several reasons.

First, some techniques allow a process to gracefully exit, whereas others forcefully terminate it. The ability to specify the termination sequence in the security policy allows the policy writer to pair a guest process with its most appropriate termination sequence. To avoid abruptly terminating a critically important process and risk data loss, a policy writer may specify a termination sequence that is limited to graceful techniques. In contrast, for programs that are more likely to be malicious, a policy writer may instead specify a more abrupt termination.

In addition, having multiple termination techniques makes evading and detecting termination attempts progressively more difficult for an attacker.

Goalkeeper's default sequence first sends a SIGTERM signal to an unauthorized process in order to allow it to gracefully terminate. If this fails, Goalkeeper sends the stronger SIGKILL signal, which the target process cannot catch, mask, or ignore. If this too fails (e.g., because the process is currently suspended), Goalkeeper deliberately crashes the process by overwriting its stack.

Following are the guest kernel memory manipulation techniques used by Goalkeeper.

3.4.3.1 Injecting Signals

A guest operating system can be manipulated into issuing a signal to one of its processes. By overwriting a guest process's sigpending struct (part of task_struct) with the desired signal, the kernel will deliver this signal to the target process at the next opportunity, typically at the conclusion

of a system call. If the injected signal allows the process to gracefully exit, such as SIGINT, the policy is enforced while reducing the risk of accidental data loss. Note: *νMon* [Li13] achieves a similar result via a different technique.

3.4.3.2 Manipulating Signal Handlers

Before injecting signals, it can be important for Goalkeeper to take additional steps to reduce the impact of certain signal handlers. A process will register signal handlers with the OS in order to perform a certain action if the handler's corresponding signal is received. These signal handlers can perform a custom shutdown procedure, report attempted tampering, or help the user file a bug report with the software developer. Signals can also be configured to be ignored or blocked altogether. To prevent a target process's signal handlers from activating, Goalkeeper erases its signal handlers (stored in `task_struct`'s `sighand_struct`), causing the guest operating system's default handlers to be used.

3.4.3.3 Corrupting the Process Stack

A process can be forced to terminate by purposely corrupting its stack, causing the process to crash. This method does not allow the process to gracefully terminate, nor provide any protections against data loss. However, since forcefulness is desired, stack corruption has the advantage of being both difficult to delay and difficult to prevent. The consequences of a stack corruption are typically immediate. With its stack erased, the process will soon perform an illegal operation, such as attempting to dereference a (now invalid) pointer stored on the stack, which yields a segmentation fault. When combined with the signal handler manipulation technique, the targeted process is unable to catch and handle the signal and is forcefully terminated by the guest operating system.

3.4.3.4 Corrupting the Kernel Page Cache

The preceding techniques terminate a guest process, but do not prevent a program from being restarted. While the new process will still be subject to Goalkeeper inspection and enforcement, it may be desirable to instead prevent the execution of *any* future processes spawned with the same executable code. To do so, Goalkeeper overwrites a target process's executable pages with a combination of `nop` instructions and calls to the `sys_exit` system call. Not only does this cause the current process to exit on its own, but this behavior persists to all future processes because the corrupted pages remain in the guest kernel's page cache after the process exits, and will get reloaded into any future processes as well. Goalkeeper remembers which processes it has affected in this manner, so it can return to other termination techniques when the corrupted pages are eventually evicted from the page cache.

3.4.3.5 Manipulating Process Priority

By manipulating a process's priority values (stored in `task_struct`), the scheduler will re-schedule the process at the desired priority level at the conclusion of the process's *next* CPU time slice. By minimizing a process's priority, Goalkeeper can significantly reduce the fraction of the CPU available for that process's execution.

3.4.4 Implementation Details

To find, parse, and traverse data structures resident in guest memory, Goalkeeper leverages the LibVMI library³ and knowledge of the guest's kernel version. Goalkeeper uses a C-based fast path and a Python-based slow path. As a part of this chapter, LibVMI's Python extension, PyVMI, was extended to support interposing on guest events.

Goalkeeper works with the Xen hypervisor and is designed to be installed inside dom0 or a domU guest with hypervisor privileges allowing VMI access.

3.5 Evaluation and Results

The evaluation consisted of several empirical studies of Goalkeeper's *effectiveness* at supervising guest processes as well as Goalkeeper's *performance* costs. Effectiveness was measured by successful identification and termination of unauthorized processes, including variants of Linux userspace malware. Performance was measured by the additional resource demands placed on the hypervisor, and the resulting overhead experienced by its guests.

The evaluation consisted of four experiments:

1. *100 Guest Goalkeeper Experiment*: A four part, 100-guest experiment measured Goalkeeper's impact on the hypervisor and its guests, as well as its effectiveness in catching unauthorized processes.
2. *MOV-to-CR3 Profiling*: A performance evaluation specifically designed to isolate *MOV-to-CR3* interception overhead across several Intel processor generations.
3. *Goalkeeper Profiling*: A detailed profile of the overheads induced by Goalkeeper's major code paths, including those associated with process identification and termination.
4. *Malware Validation*: A verification that Goalkeeper could find and terminate processes affected by a variety of malicious activity, including a return-oriented programming (ROP) attack and Linux userspace malware.

³<https://github.com/libvmi/libvmi>

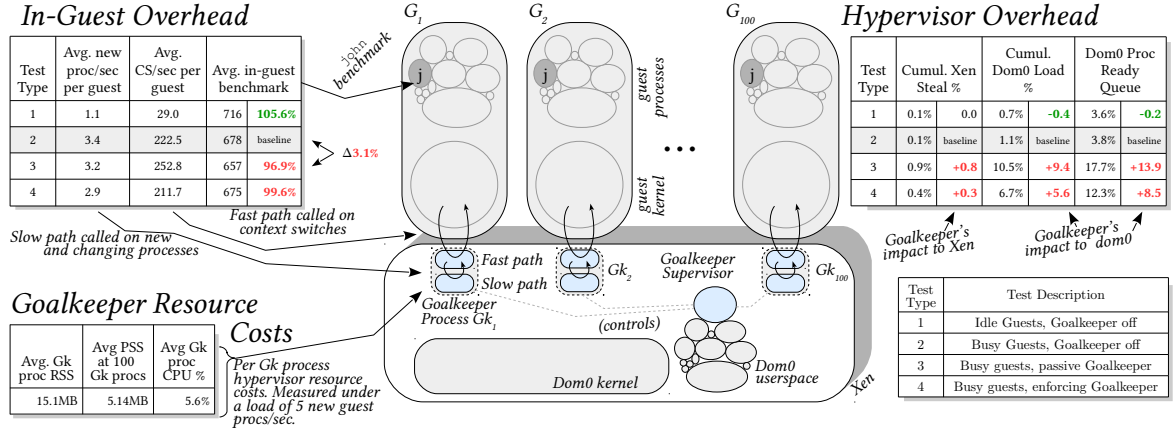


Figure 3.5 A summary of Goalkeeper's performance overheads taken during the 100 guest experiment. Goalkeeper imposes a 3.1% in-guest performance overhead on guests it protects. Each per-guest Goalkeeper process requires an average of 15.1MB dom0 memory (RSS) and 5.6% dom0 CPU while inspecting 5 new guest processes/second. When operating under 100 guests, Goalkeeper increases dom0's total system load by 9.4%.

3.5.1 Evaluation Environment

The evaluation environment consisted of a single Dell R920 hypervisor running Xen version 4.7.1 with a Fedora 25 domain 0. This hypervisor was furnished with four 2.3GHz Intel Xeon E7-4870 processors and 256GB RAM.

Each fully virtualized guest virtual machine was allocated a single vCPU, 1024MB RAM, and 10GB of storage. The Gnome window manager and popular desktop applications like LibreOffice, Firefox, and Evolution were installed on each guest using the SaltStack configuration management software.⁴

Goalkeeper policy ruleset contained 7,270 rules in this experiment, one rule for each of the executables found on a fully-configured guest. The ruleset was initially created automatically by a script which crawls a guest's filesystem and characterizes each executable. The ruleset was expanded by a second script which extracted runtime information, including memory layouts, from a guest's running processes.

3.5.2 100 Guest Experiment

In this experiment, new guest processes were invoked in random guests to exercise the *frequency* and *quality* of Goalkeeper's decisions. Goalkeeper's *scalability* was also evaluated by monitoring many guests simultaneously. Four test types were used to isolate certain aspects of Goalkeeper under heavy load. In the experiment, Goalkeeper ran under 100 concurrently executing guests,

⁴<https://github.com/saltstack>

50 running Fedora 23, and the other 50 running Fedora 25 (released in November 2015 and 2016 respectively).

In-guest overheads were calculated by measuring a guest's performance difference with and without Goalkeeper's protection. These deltas were quantified by the built-in benchmark test provided by brute force password guesser John the Ripper.⁵ John's benchmark test calculated the average number of bcrypt hashes per second executed by a guest. Since Goalkeeper briefly pauses a guest during process context switches, this lost time reduced John's time on the guest vCPU and resulted in a smaller benchmark score.

During each trial, unauthorized processes were spawned at the rate of three every two seconds, meaning they made up roughly 1% of the new processes. Goalkeeper was expected to find and terminate these unauthorized processes from within the total number of running processes across all the guests.

The results of this experiment are shown in Figure 3.5 and discussed below.

In-guest overhead. A set of four test types isolated Goalkeeper's in-guest performance overhead. The type 1 *idle control* test measured the CPU benchmark's upper bound on a hypervisor running otherwise idle guests. This served as a reference point for the remaining tests. Goalkeeper was not running during this test.

The type 2 *busy control* test introduced the spawning of test processes, which in turn measured a drop in John's benchmark performance when compared with type 1 results. This was due to the addition of the thousands of additional processes (both authorized and unauthorized) from both the in-guest test framework and the test processes themselves. Goalkeeper was not running during this test.

The type 3 *passive Goalkeeper* test was identical to the type 2 test, except with the addition of Goalkeeper running under each guest in passive mode. In this mode policy checks occur and are logged, but no termination attempts are made. This test had the worst CPU benchmarks, combining the influx of processes from the type 2 test with the per-process-switch overhead of Goalkeeper performing frequent (roughly 3.2/second/guest) policy checks. When compared to the prior test, Goalkeeper's overhead is calculated as approximately 3.1%.

The type 4 *enforcing Goalkeeper* test showed an improvement in the benchmark scores. This improvement was attributed to the savings in CPU time which would have otherwise been utilized by the unauthorized processes. These unauthorized processes, allowed to run to completion in the type 3 test, were instead terminated by Goalkeeper in the type 4 test. With these unauthorized processes absent, the CPU time could instead be used by the remainder of the system, including the benchmark processes. The quantity of guest context switches also indicates this effect. The unauthorized, long-running processes in the type 3 test caused Goalkeeper to observe many more context switches (and to a lesser degree, handle more policy checks).

⁵<https://github.com/magnumripper/JohnTheRipper>

Summary: Goalkeeper efficiently protects large sets of guests and imposes in-guest overheads comparable with related work (CLAW: $\approx 4\%$, Patagonix: $\approx 3\%$).

Goalkeeper resource costs. For 100 guest VMs running on a single host machine, the cumulative sum of Goalkeeper’s main process and 100 children required approximately 514MB of dom0 memory, which averages to 5.14MB per guest. This amount of memory overhead is easily sustainable by today’s virtualization standards; for example, on a guest allocated 2GB of memory (a common rule of thumb), this represents less than 0.3% of available memory. The number of processes awaiting CPU time also rose by an average of 13.9 processes.

With the memory efficiency of shared libraries, the incremental cost of Goalkeeper processes running in dom0 is considerably less than each individual process’s in-memory Resident Set Size (RSS). Figure 3.6 shows the declining per-Goalkeeper-process Proportional Set Size (PSS) for increasing numbers of Goalkeeper processes. This PSS is defined as the (unique) memory specific to that process plus its “share” of the memory used by its shared libraries. Goalkeeper’s per-process PSS approaches 5.14MB.

Summary: Goalkeeper’s parallelized design is economical, especially for increasing numbers of guests.

Hypervisor overhead. Goalkeeper’s dom0 overhead was measured by analyzing the difference between the results of the type 2 and type 3 tests. Figure 3.5 shows that Goalkeeper increased dom0’s system load from 1.1% to 10.5%. Additionally, Goalkeeper’s load on the underlying Xen hypervisor increased the percentage of time the hypervisor stole from dom0 from 0.1% to 0.9%.

Summary: Goalkeeper’s hypervisor overhead further demonstrates its scalability properties.

Effectiveness. Goalkeeper’s successfully found all unauthorized processes and terminated them within an average of $8.6\text{ms} \pm 0.8\text{ms}$ of process CPU time. Due to the granularity of the security policy rules, no false positives (mistaken terminations) or false negatives (undetected unauthorized processes) were recorded during the trials. Figure 3.7 shows a histogram of the time required to terminate a process using the default termination sequence (SIGTERM, SIGKILL, Stack Corruption).

Summary: Goalkeeper’s automatically generated whitelisting policies effectively identify unauthorized processes, and Goalkeeper’s termination techniques prevent their further execution.

3.5.3 MOV-to-CR3 Profiling

The second experiment quantified the overhead involved in capturing and delivering *MOV-to-CR3* events from a guest, through the hypervisor, and to Goalkeeper. This experiment is important because it separates Goalkeeper-specific overheads from the unavoidable overheads inherent with CR3 interception.

This “minimum round trip time” represented the highest event throughput performance currently possible within this experimental setup, and the shortest potential time a guest could expect

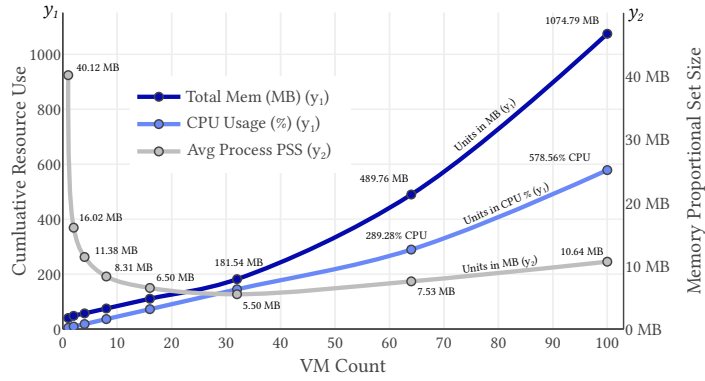


Figure 3.6 As Goalkeeper is used to protect increasing numbers of guests, Goalkeeper’s per process Proportional Set Size (PSS) memory usage approaches 5.14MB, whereas CPU load grows linearly with the number of guests.

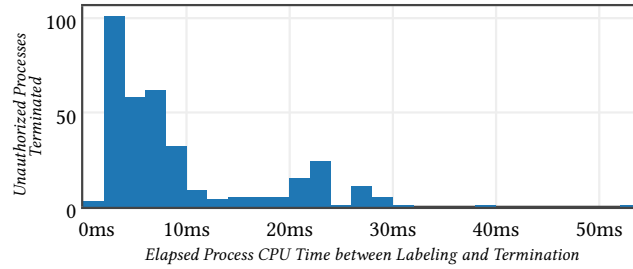


Figure 3.7 Once a process is labeled as unauthorized, Goalkeeper requires an average of $8.6\text{ms} \pm 0.8\text{ms}$ to terminate it.

to be paused. For this experiment, Goalkeeper’s inspection functionality was disabled, and its slow path was set to immediately return without performing inspections. This isolates the time required to capture an event from the guest, relay it through the hypervisor and the VMI domU’s software stack, and into Goalkeeper’s fast path.

This overhead was measured across various new process spawn rates on several Intel processors, including a Sandy Bridge i7, Skylake i7, and Xeon E7. In addition to the john benchmark, the SysBench⁶ CPU benchmark was also used in this evaluation as calibration.

Figure 3.8 shows the resulting overheads. In the case of 32 new processes per second, capturing and delivering *MOV-to-CR3* events created an in-guest performance overhead ranging between 2.5 - 3.5%. When compared to the overheads in the preceding scalability experiment, it can be concluded that the *MOV-to-CR3* component of Goalkeeper constitutes one sixth, or approximately 0.5%, of the in-guest overhead.

Summary: Goalkeeper’s underlying interposition technique is inexpensive.

⁶<https://github.com/akopytov/sysbench>

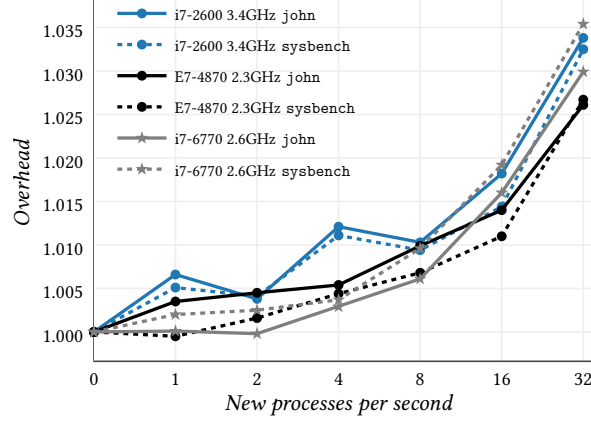


Figure 3.8 The *MOV-to-CR3* round trip time overhead quantifies the time required to move a guest event through the hypervisor and into Goalkeeper’s fast path.

Table 3.4 Goalkeeper Profiling

Name	Avg. μ sec	Ratio
<i>Policy Checking Performance</i>		
Fast path only	15.7 ± 1.0	88.4
Slow path name only	488.2 ± 89.9	0.6
Slow path full enum.	$1,420.8 \pm 738.3$	9.5
Slow path partial enum.	175.5 ± 54.0	1.5
<i>Termination Performance</i>		
Signal Injection	$3,145.1 \pm 196.7$	-
Stack Corruption	$5,087.6 \pm 236.2$	-
Page Corruption	337.9 ± 35.4	-

Results are averaged and shown with 0.95 confidence interval.

3.5.4 Goalkeeper Profiling

This experiment measured overhead associated with Goalkeeper’s common code paths. In addition to Goalkeeper’s fast path and slow path, its termination attempts and partial enumerations were also measured. Partial enumerations are important because, from Goalkeeper’s perspective, new guest processes are “built up” over their first few context switches. Goalkeeper performed these partial enumerations based on the subset of information available during each context switch. Table 3.4 shows the resulting cost and frequency of Goalkeeper’s policy checking modes.

These results show that on average Goalkeeper’s fast path called into its slow path in only 11.6% of guest context switches. Thus, Goalkeeper imposes its minimal 15μ s guest overhead in the large majority of cases (88.4%).

Summary: Goalkeeper is efficient because its fast path prevents unnecessary inspections.

3.5.5 Malware Validation

In this experiment, three malware samples (two Jynx^{7,8} variants and Azazel⁹), were individually installed and executed on guests. Unlike other Linux userspace malware, which establish persistence via cron or init scripts and can be readily marked as unauthorized using Goalkeeper's combination of CR3 interception and process security policies, these samples establish persistence by manipulating the victim's dynamic linker into preloading a malicious shared library into each new process. This method allows the malicious shared library to hook standard `libc` functions to conceal its effects on network connections, `procfs`, and the file system. These samples were selected for this unique persistence method, as this method was initially undetectable by the popular rootkit detection tools `rkhunter` and `chrootkit`.

Goalkeeper detected the presence of all three samples of userspace malware as affected processes started execution. When inspecting a guest process compromised by a malware sample, Goalkeeper detected a discrepancy in the shared libraries between the guest process and its security policy, and correctly labeled the victim process as unauthorized.

Goalkeeper was also evaluated for its ability to detect the consequences of a canonical Return-Oriented Programming (ROP) attack [Sha]. In such an attack, a vulnerability allows an attacker to overwrite a process's stack with a carefully crafted sequence of pointers (*gadgets*) to executable code elsewhere in memory. Each gadget points to a code snippet that, when combined with adjacent gadgets and executed in series, performs malicious logic. In the canonical ROP attack, this logic spawns a child process that calls back across the network to the attacker, allowing access into the guest.

For this test, a ROP-vulnerable program was written and installed on a guest. A corresponding security policy was automatically generated. The program was started, and a buffer overrun vulnerability allowed unsanitized input to spill onto the stack, overwriting existing data with the ROP gadgets which spawn the child listener.

Goalkeeper labeled both the victim process and its ROP-induced child process as unauthorized. The child process created by the ROP attack was inspected by Goalkeeper. Since the child's parent did not match the parent specified by the Goalkeeper rule (in this case, a shell), the policy check failed (and vice versa for the parent).

Patagonix and *CLAW* both focus on hashing and verifying a guest process's executable code. Since ROP re-uses existing executable code, these approaches would be unable to detect or prevent this attack.

Both portions of this experiment indicate that Goalkeeper is excellent at detecting changes to a process's memory layout and anomalous process attributes. In the first case, a policy detected

⁷<https://github.com/chokepoint/Jynx2>

⁸<https://github.com/chokepoint/jynxkit>

⁹<https://github.com/chokepoint/azazel>

changes in process memory space, such as when malware abused LD_PRELOAD to load (or remove) shared libraries. In the second case, Goalkeeper found violations of child-parent relationships, such as seen with a ROP-vulnerable program and its child listener. Goalkeeper can also catch inconsistencies related to file path, arguments, environment variables, and user information. Taken as a whole, Goalkeeper raises the level of difficulty required to attack a system.

Summary: Goalkeeper can catch malicious code missed by prior work. To generalize, an attack would only be able to induce behaviors that are allowed by policy anyway.

3.6 Discussion

3.6.1 Goalkeeper's Policy Language

Language utility. It can be asserted that the overall utility of a firewall, intrusion detection system, or other policy-based security system is directly related to the flexibility and extensibility of its policy language. When compared to prior VMI work in this area, Goalkeeper's policy language uniquely provides a degree of flexibility and extensibility that has not previously been proposed. This expressiveness and flexibility is made practical by Goalkeeper's scalability to many simultaneous guests.

Since various process attributes are also examined by both VMI-based [HC13; Lit08; GR03] and in-guest (e.g., AppLocker, Gatekeeper) approaches, a policy language can be applied to them as well. For example, CLAW, Patagonix, and Livewire's user program integrity detector module all work by comparing a known-good hash with the comparable hash of a guest process's executable in memory. In Goalkeeper's policy language, these are easily expressed as:

```
[/usr/sbin/sshd]
page-hash      = {
  0:17655518bf67e6a17be2a44000e082d560ff6ff4a1e6e56a6...,
  1:9f0f82dde765254ec6a4e0f59b601d0f171399846bc7fa514...,
  ... }
```

The adoption of a policy language would support user-extensibility across all tools in this class, and would be particularly useful if standardized.

Language additions. Other process attributes might be useful additions to the policy language, including time (e.g., a process's start time, CPU-time behavior, and duration), order and sequence (e.g., processes which start or exit together in episodes), or memory access patterns.

Adding a new process attribute to Goalkeeper would involve developing a VMI procedure to extract the attribute from the guest, integrating this attribute into the policy language grammar, and updating relevant portions of the Goalkeeper slow path code. The attributes discussed in this

chapter were selected based on how their ability to discriminate malware from benign code, and the overhead of collecting and examining them.

3.6.2 Resistance to DKOM

Since Goalkeeper’s security policies leverage information under the control of the guest OS, Goalkeeper belongs to the class of VMI tools which are open to potential manipulation by either a malicious guest kernel or via Direct Kernel Object Manipulation (DKOM) attacks [Bah10; Jai14]. To defeat Goalkeeper’s process identification scheme in this way, an attacker must forge a fake process with attributes matching the corresponding security policy being enforced by Goalkeeper. If done correctly, this fake process will then adequately resemble an authorized process and pass Goalkeeper’s policy checks.

While this manipulation is possible for policies completely sourced from OS-based invariants (e.g., file path or process name), given this chapter’s assumptions and threat model there is no known way for in-guest attackers to subvert hardware-based invariants (e.g., code hashing) to mount a similar attack. There is also no known way for a guest to conceal process execution by preventing CR3 interception.

3.6.3 Resistance to Anti-Virtualization

Goalkeeper does not prevent a guest from determining that it is virtualized. A guest could discover this fact by detecting virtual devices which are only present in virtualized environments, measuring time variations, testing certain processor instructions, or refusing to run on a single-vCPU (in the eyes of attackers: potentially sandboxed) machine [SH12; Che08].

Ultimately, these techniques are leveraged by malicious code to abort execution when inside a VM [Din08], which although hampering malware analysis, is consistent with the purpose of Goalkeeper.

3.6.4 Guest Kernel Sensitivity

Although VMI-based process supervision applies to all operating systems, Goalkeeper is currently implemented to work under a single OS (consistent with related work [GR03; HC13; Wu14]). Linux was selected because of the availability of its source code and debugging symbols. Goalkeeper’s VMI techniques rely on common operating system concepts, such as signals and scheduler structures.

Upgrades to Goalkeeper may be required when new OS versions are released. To investigate this level of effort, Goalkeeper was tested against several other Linux distributions with kernels in the 4.0 series. Of particular concern was the kernel-version sensitivity of Goalkeeper’s termination techniques. Kernel drift in the 4.0 series did not affect the particular structures and behaviors that Goalkeeper uses to target a process. The two versions of Fedora used in the evaluation, Fedora 23

and 25, differed by a year’s worth of kernel updates (kernel versions 4.2.3 and 4.8.6, respectively), but Goalkeeper process inspection and termination worked flawlessly in both cases.

3.6.5 Limitations

Policy generation. Goalkeeper produces a conservative policy that is an under-approximation of the number of potential process states. Practically, this means that programs with unpredictable memory loads can not take advantage of process attributes that limit memory usage to certain ranges. It is also difficult to detect malicious activity that is not directly reflected in process attributes, including browser-based or interpreter-based applications. As future work, Goalkeeper could be extended to increase its heap-based and anonymous-memory-based attributes, since these areas are where web-based content and interpreted code is stored during execution.

Ultimately, the scope of attributes, policy grammar, and policy generation were adequate for Goalkeeper, but further work in this direction could yield additional attributes and more concise, effective policies.

MOV-to-CR3 granularity. *MOV-to-CR3* interception interposes on process context switches. While suitable for supervising guest processes, it does have limitations.

First, as compared to other VMI techniques, intercepting writes to the CR3 register cannot be used to directly detect process-generating system calls (e.g., `fork`) or monitor kernel code that actually builds the new process.

Second, although CR3 interception allows Goalkeeper several “looks” at processes during their initialization, processes after initialization may be inspected infrequently. As an example, the context switch interval for a running process, which triggers re-inspection by Goalkeeper, is tens of milliseconds for Linux’s default scheduler. Activities or changes that occur purely within one context switch interval may therefore be overlooked. This does not include activities that execute a blocking system call, such as writing or reading data to disk, sending data across the network, or performing inter-process communication. Such blocking calls trigger a context switch, allowing Goalkeeper to inspect the process when it is resumed.

3.6.6 Related Work

VMI’s technical integration with *Volatility*¹⁰ and *Rekall* [DG11a] reinvigorated VMI-based research, including novel work related to high-interaction honeypots [Len12], and real-time guest kernel data structure monitoring [HC14]. Although *Volatility* and *Rekall* are often the tools of choice for memory analysis, they are not designed to write into guest memory and therefore Goalkeeper handles a guest’s data structures directly.

¹⁰<http://www.volatilityfoundation.org/>

By focusing on hardware-based invariants, *Patagonix* [Lit08] enforced strong security guarantees based on characteristics of a process’s page tables. This strengthened Patagonix’s security model and reduced the effort required to support additional guest operating systems. However, with Patagonix there is still the cost of maintaining its oracles and lookup databases, and it is more difficult for users to create practical, high-level security policies given Patagonix’s avoidance of OS-based attributes. Additionally, Patagonix was not tested against guests in parallel. In contrast, Goalkeeper examines processes and enforces security in a scalable way. Goalkeeper exploits both OS-maintained process attributes and hardware-based invariants, and its security policy language is powerful and easily understood.

Goalkeeper’s escalating termination sequence is conceptually similar to *VICI* [Fra08], which performed an escalating sequence of attempts to remove rootkits. If VICI’s initial inexpensive techniques were unsuccessful, increasingly expensive methods were attempted until finally the system was rolled back to a known-good state.

Hypervisor-based systems such as Goalkeeper are conceptually similar to a host operating system’s kernel modules. Security systems can be implemented at either location, and similar proposals implemented via kernel modules include *PinUP* [Enc08], *PRIMA* [Jae06], *bin-locking* [Wur10], and *configd* [Wur10]. Other memory acquisition or system monitoring techniques do not rely on VMI. *LO-PHI* [Spe16] used hardware instrumentation in the stealthy access of memory on bare metal machines. Additional low-level memory access techniques via System Management Mode were explored by *MalT* [Zha15] and *HyperCheck* [Zha14]. For each technical approach (in-guest, VMI, hardware instrumentation, and special-purpose CPU modes), there is a cost-benefit argument to be made across performance, security, and applicability. For more discussion, see [Wan15a; Bau15].

3.7 Summary

This chapter showed that guest processes can be flexibly inspected and controlled *en masse* on hypervisors with high concentrations of guests. Goalkeeper’s access to a variety of process attributes allow multifaceted security policies, which are shown to catch malicious code that slips by the more limited policies of prior work. Using guest kernel memory manipulation, Goalkeeper can induce a guest kernel to slow down, gracefully terminate, or force guest processes to exit. Goalkeeper’s performance was measured on a population of virtual Linux workstations, and the results determined that it performs and scales well. Goalkeeper also provides a rich framework for future work, including the further study of kernel manipulation sites and more sophisticated process monitoring analysis. Ultimately, this chapter strengthens the idea that a well-designed hypervisor-based security system is a viable complement to traditional OS-based security.

CHAPTER

4

HYPERVERSOR-BASED MONITORING

While the previous chapter focused on evaluating performance and scale through a large, multi-guest evaluation, this chapter introduces *Arav*, a hypervisor-based security system that infers guest activity by tracing a small set of guest functions. Across two guest operating systems and two guest programs, Arav shows that various security events can be successfully recovered from guest memory. In many cases, tracing just a single function can recover any log message normally emitted by a process during runtime—an insight that allows Arav to be remarkably efficient. This chapter applies Arav to Virtual Routers (VRs), which stand to benefit most from its approach.

VRs are increasingly common in cloud environments, and are used to route traffic between network segments and support network services. Routers, including VRs, have been the target of several recent high-profile attacks, emphasizing the need for more security measures, including security monitoring. However, existing agent-based monitoring systems are incompatible with a VR's temporary nature, stripped-down operating system, and placement in the cloud. As a result, VRs are often not monitored, leading to undetected security incidents.

Arav scrutinizes VRs by novel application of Virtual Machine Introspection (VMI) breakpoint injection. Arav monitored and addressed security-related events in two common VRs, *pfSense* and *VyOS*, and detected four attacks against two popular VR services, *Quagga* and *OpenVPN*. Arav's performance overhead is negligible, less than 0.63%, demonstrating VMI's utility in monitoring virtual machines unsuitable for traditional security monitoring.

4.1 Introduction

Modern cloud infrastructures, including OpenStack, VMware NSX, and Amazon Web Services (AWS), rely on Network Function Virtualization (NFV) to provide flexible software-based virtual networks for tenant network segmentation. Virtual Routers¹ (VRs) are used to connect these network segments and can also provide common network services such as NAT, VPN, dynamic routing, and firewalling. VRs are also positioned in non-tenant-facing roles, such as assisting in the switching and internal management of segmented tenant traffic.

Despite their wide use, the routine security monitoring of VRs is particularly inelegant in cloud environments, leading to situations where VRs are not monitored and any resulting security incidents are undetected. Centralized VR monitoring is difficult because VRs are either too ephemeral to be worth the effort, require runtime configuration, or lack connectivity to a centralized management system. VRs are also often stripped of software components required for agent-based monitoring.

To address the practical monitoring challenges of a cloud's VRs, this chapter proposes *Arav*², a Virtual Machine Introspection-based (VMI) system used to monitor VRs with the visibility of an agent but without the associated drawbacks. Arav operates from the cloud's hypervisors to monitor VRs and extract their security-related events during runtime. Arav uses VMI to install instruction-level breakpoints into a VR's active memory, allowing Arav to interpose and analyze a VR's process-level function calls, reconstitute security events from these calls, and take the appropriate mitigating action to minimize negative impact. Arav is designed to be an internal cloud environment security feature, and could also be offered to tenants as a managed security service.

To illustrate and evaluate this approach, Arav was used to defend two common VRs, *pfSense* and *VyOS*. Arav monitors two network services in each of these routers, their OSPF daemon, *Quagga*, and their VPN concentrator, *OpenVPN*. Arav is evaluated by measuring the time required to detect and mitigate four attacks against these services. Arav's breakpoint injection mechanism is also measured for its impact on network throughput.

Arav detected each attack in near real time, costing less than 0.63% reduction in packet throughput. Each breakpoint injected by Arav required the VR to be suspended for an average of $161\mu s$. We conclude that use of VMI on isolated virtual infrastructure is an efficient, feasible, and effective monitoring method.

This chapter makes the following contributions:

- Arav, a general purpose VMI framework which reconstitutes security events from guest process function calls.
- The novel application of Arav to monitor VRs. VRs are often difficult to monitor by other means.

¹A virtualized router OS running as a guest on a hypervisor.

²Hebrew, to lie in wait

- A breakpoint injection evaluation demonstrating VMI's feasibility in production systems.

4.2 Virtual Router Security

This section introduces VRs, their applications and security issues (§4.2.1), and the difficulties in monitoring them effectively (§4.2.2). A comparison of available monitoring designs suggests that a VMI-based design may be best (§4.2.3). This new design should be VR-role-agnostic, simple, and have access to rich context. The benefits and challenges of an agentless approach are discussed (§4.2.4), and the section concludes by introducing the threat model (§4.2.5).

4.2.1 Background

A virtual router is a virtual machine dedicated to performing common router functions across two or more networks. VRs perform many of the same functions as business and consumer-grade routers.

4.2.1.1 Common VR Roles

Modern cloud ecosystems rely on VRs for a wide variety of tasks. Figure 4.1 shows two VRs in different roles, where VR_1 is used to support internal cloud functions, and T_4 is provisioned by a tenant for their own use. Each of the following cloud environments use VRs in different ways, which highlight VR utility and widespread adoption.

VMware NSX environments ship with two VR components. The first, NSX's Distributed Logical Router (DLR) system, pairs each hypervisor with a control VM (the DLR VR), which functions as the routing control plane, issuing forwarding updates to the data plane running as a hypervisor kernel module. The second, NSX's Edge Gateway VR, provides VPN and NAT services.

OpenStack takes a different approach. OpenStack's reference implementation provides conceptually similar segmented logical networks through its hypervisors' Linux IP stacks and iptables configurations. However, OpenStack API abstractions allow a variety of products, including this chapter's definition of VRs, to actually implement the API features under the hood.

Amazon Web Services (AWS) use VRs to support product offerings such as Virtual Private Cloud (VPC), which offers tenants the ability to establish a site-to-site VPN between AWS resources, such as a tenant's virtual machines, and the tenant's on-premises data center.

Lastly, many cloud tenants choose to deploy their own ecosystem-independent VRs, either by uploading a custom VR image to their cloud provider, or by selecting a VR from a cloud's storefront. At the time of this writing, AWS Marketplace offered over a hundred VMs associated with routing, firewalling, and VPNs.

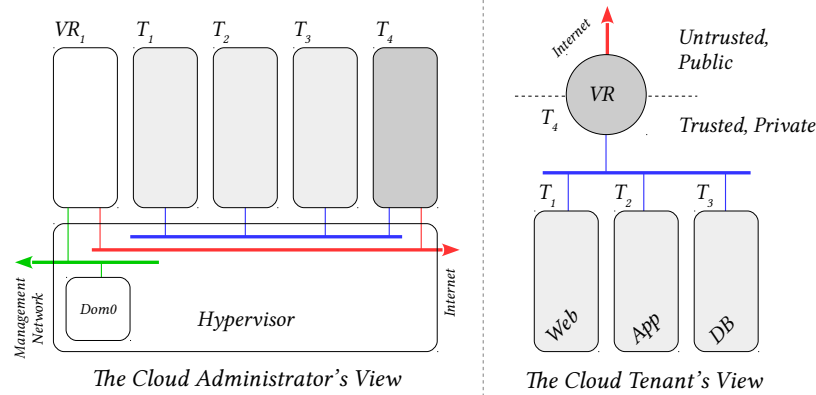


Figure 4.1 Two common VR roles: VR_1 provides cloud administrators with remote VPN access into the cloud management network, and T_4 serves as a firewall and NAT for a cloud tenant.

4.2.1.2 VRs and Security

Routers, including VRs, are valuable targets. They have access to protected networks and can manipulate packets routing between these networks. Routers often host network services which can be attacked and manipulated. A router's network throughput is also valued by botnet administrators. Finally, attacking a router and taking it offline can cause a denial of service for other hosts dependent on its services.

Recent attacks against routers include the SYNful Knock attack³ against Cisco devices, two high-profile Juniper CVEs related to hidden backdoor access and weakened VPN encryption,^{4,5} and various critical vulnerabilities against consumer-grade routers from D-Link, ASUS, and others.^{6,7} Recent research found vulnerabilities in hundreds of consumer-grade firmware images [Che16]. At least 25% of these images were routers.

Although high-profile VR attacks are less common, VRs perform identical roles as hardware and consumer-grade routers and remain valuable targets. The absence of high-profile VR attacks does not mean VRs are not targeted or compromised, and VR-specific vulnerabilities⁸ and shared vulnerabilities⁹ often occur. Since VRs are based on commodity operating systems like Linux (e.g., Cisco CSR1000V, VyOS, Brocade vRouter, NSX Edge) or FreeBSD (e.g., pfSense), VRs must still be monitored and patched regularly.

Lastly, VRs can also contain quirks, limitations, and weaknesses not found in their upstream

³ https://www.fireeye.com/blog/threat-research/2015/09/synful_knock_-_acis.html

⁴ <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-7755/>

⁵ <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-7756/>

⁶ <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-2718/>

⁷ <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-2052/>

⁸ <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6385/>

⁹ <https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-2-0150325-iosxe/>

operating systems. Some commercial VR developers introduce anti-tamper techniques to hamper analysis of the VR's filesystem. Additionally, since many VRs consist of specialized forks of open source projects, their software components often are not updated and patched as frequently, and may fall months or years behind.

4.2.2 VR Monitoring Challenges

VRs are integral components of the cloud and NFV, and require security monitoring. However, monitoring VRs is often more complex than a typical VM.

First, in some cases it is impractical to monitor a VR via a centralized network management system (NMS). Some VR lifecycles are tied to the tenant they support, meaning the VR is ephemeral enough to make integration into a NMS administratively unattractive. Other VR roles (e.g., VMware NSX DLR) are isolated without any network connectivity.

Second, VRs are often stripped of many software components required to install and operate agent-based monitoring (e.g., compilers, debuggers, header files, and shared libraries). While software components could always be manually added back into most VRs, it is unappealing to update a VR each time a new version is released. Agents and their software dependencies are also not uniformly compatible across different VRs. A cloud provider desiring a uniform solution across all its VRs would likely have to retrofit a VR, either before or after provisioning. Providers are focused on operations, and would be hesitant to invest in directly retrofitting and supporting customized VRs.

Lastly, some VRs operate in situations where they are provisioned automatically by the cloud to serve a single tenant. The tenant indirectly configures the VR via cloud APIs, which creates a situation where both parties have input into the VR's security configuration.

4.2.3 VR Monitoring Requirements

Given the real-world constraints of a cloud provider and the unique challenges associated with monitoring VRs, some traditional monitoring approaches may not be feasible. Table 4.1 lists several designs, and compares them against properties valuable for monitoring VRs.

VR role agnostic. Every VR type should be monitored in a similar way. For example, if a monitoring solution depends on a VR's network connectivity to a NMS, it likely cannot be universally applied, because certain VR roles will lack this connectivity.

Simple. The monitoring system should be easy to administer. Increasing monitoring complexity places additional load on administration and troubleshooting. Most monitoring solutions require some degree of VR configuration, such as the addition of a destination IP for syslog messages or the routine maintenance of in-guest detection signatures.

Context. The design of a monitoring solution impacts the quality and fidelity of the host information it can collect. Remote logging may be simple and maintainable, but log messages may be insufficient

to diagnose a potential security incident. Network-based intrusion detection, which inspects a VR's traffic somewhere along a VR's network path, can have limited monitoring utility due to packet encryption.

4.2.4 Replacing Agents with VMI

As discussed in §4.2.2, agent-based monitoring adds complexity and cannot be applied in certain VR roles. However, agents offer the required security context. A desirable solution would provide equivalent context without the additional complexity.

An agent runs within a VR as an internal observer, monitoring and reporting on the VR's runtime information. Agents have access to rich context, and can query the VR's operating system directly about process state, logs, and kernel objects. By virtue of running inside the VR, agents have inherent access to enough information to detect security events and patterns in VR behavior.

In contrast, a Virtual Machine Introspection (VMI) application runs on the underlying hypervisor and inspects a VR as an external observer. VMI applications have fundamental access to a VR's active memory and registers at a much lower level than most agents. However, this access comes without knowledge on guest data structures and their locations (VMI's *semantic gap* problem [CN01]), so VMI applications must take additional steps to properly interpret the contents of guest memory. Additionally, unlike an agent, a VMI application cannot directly interact with in-guest APIs.

If a monitoring system with equivalent context could be built around VMI instead of an in-guest agent, such a system could avoid an agent's disadvantages.

First, unlike agents, using VMI to monitor a VR from its underlying hypervisor does not require network connectivity. A hypervisor can observe its guests irrespective of the guest's network connection. Also, since hypervisors are already typically monitored by an NMS, this arrangement could be reused for VR monitoring data. Second, a VMI monitoring system resides entirely outside of the VR. VMI is inherently agentless, and does not require code or configurations to be added into VRs. VRs could be monitored without internal customization. Finally, VMI minimizes the need for additional cloud complexity associated with monitoring-related network connectivity, configuration, and maintenance. Hypervisors are already in a natural position to monitor VRs.

4.2.5 Threat Model

This chapter adopts a remote attacker threat model. Remote attackers originate from the Internet or from within the tenant's private network. The attacker probes and discovers a VR's exposed services, then attacks these services in an attempt to achieve effects against the VR's execution. Networking devices like VRs can also be intermediate targets; a compromised VR allows an attacker to achieve additional effects against the cloud or downstream tenant VMs.

Attacks against VRs can include protocol manipulation, application-layer, and denial-of-service

Table 4.1 Security Monitoring Classes

Class	VR Role Agnostic	Operational Simplicity	Access to Context	Tool Examples
Active Agent	○	○	●	OSSEC, fail2ban
Monitor Agent	○	○	●	Zabbix, Nagios
Remote Logging	○	◐	◐	Syslog, Filebeat
SNMP	○	◐	◐	net-snmp
NIDS/NIPS	●	●	○	Snort, Bro
VMI	●	◐	●	Spider, Drakvuf

Filled circles are better. Partial fill indicates marginal or incomplete applicability.

attacks. These could take the form of forged routing packets, brute-force credential guessing, or a flood of new connection attempts. Also considered is a general class of attacks which result in the crash of a network service. This threat model assumes attacks do not originate from underlying cloud components, which are considered the trusted computing base. If the cloud control plane is compromised, any of the monitoring systems in Table 4.1 can be deactivated or subverted by the attacker. The VR is considered pristine at provision-time and will not attempt to subvert security monitoring functions. Subverting security monitoring is discussed further in §4.6.1.

4.3 Design and Architecture

The objective is to design a new hypervisor-based security monitoring system that leverages VMI instead of in-guest agents. The implementation of this design, *Arav*, scrutinizes VRs by tracing their process-level function calls using breakpoint (BP) injection, reconstituting security events from these traces, reasoning about these events, and enacting responses.

This section describes Arav’s major components, including its VMI technique (§4.3.1), the one-time analyst workflow to prepare a VR version for Arav (§4.3.2), and Arav’s ability to extract and respond to VR security events (§4.3.3).

Arav was designed to inspect any Linux or FreeBSD-based VR, and the two common VRs used in Arav’s evaluation, pfSense and VyOS, are introduced along with two of their network services, Quagga and OpenVPN (§4.3.4). The section concludes with a discussion on the attacks used for Arav’s evaluation (§4.3.5) and Arav’s implementation details (§4.3.6).

4.3.1 Gaining Equivalent Context with VMI

Arav's principle challenge is extracting a real time event stream from a running VR without the use of in-guest agents or network connectivity. VMI [Wan15a] presents a clear answer to this challenge, and VMI's breakpoint (BP) injection technique offers the most potential, allowing a VMI application to trace a VR's function calls [Den].

To trace a VR's function calls, Arav's BP injection implementation inserts a breakpoint (the INT3 instruction, 0xCC) into a VR process's executable code. Each BP is inserted at the entry point of a function. When that function is called, its BP is encountered and trapped by the hypervisor, allowing further action, such as extracting the function's arguments or inspecting the VR process's stack. Trapping breakpoints to the hypervisor is a feature of modern Intel processors, enabled by modifying the exception bitmap in the VM-execution control group of a guest's Virtual Machine Control Structure (VMCS) [Int16].

Figure 4.2 depicts an Arav BP injection scenario. Shown are three contiguous pages of a VR process's virtual address space containing its executable code. When the VR schedules this process to run, execution begins at $t = 1$ and follows the process's normal control flow, including the invocation of function calls (labeled with letters). The entry points for these functions are potential targets for injection, and the figure shows one BP injected at the beginning of function *D*. As the process's execution continues to $t = 5$, it encounters this BP. The VR's vCPU, configured to VMEXIT when encountering a BP, suspends the VR and performs a world switch to the hypervisor, which relays the event to Arav. Arav can then take a variety of actions (discussed in §4.3.3).

4.3.2 Analyst Preparation

Before Arav can monitor a VR, an analyst (e.g., a VR developer or security researcher) must tell Arav how to monitor a VR by constructing a VR *capability set*, which consists of three related components: 1) a list of the desired VR function names to trace, 2) a set of event generators to query the function call trace and produce security events, and 3) a set of rule analyzers to query security events and perform response actions.

The analyst begins by creating security monitoring goals, such as those pertaining to the VR's intended purpose, any anticipated attack vectors, or recently patched VR vulnerabilities. For example, if a VR is intended to terminate VPNs, the analyst would focus on the VR's VPN daemon, where one security monitoring goal might be to monitor new connection attempts.

The analyst then iteratively inspects the target program's source code and selects functions that are correlated with each goal. Continuing the VPN example, if a pair of functions are invoked together when a user attempts to establish a VPN connection, the analyst could build an event generator to produce a security event when these functions occur together in a trace.

There are potentially thousands of functions to consider, but shortcuts exist. For example,

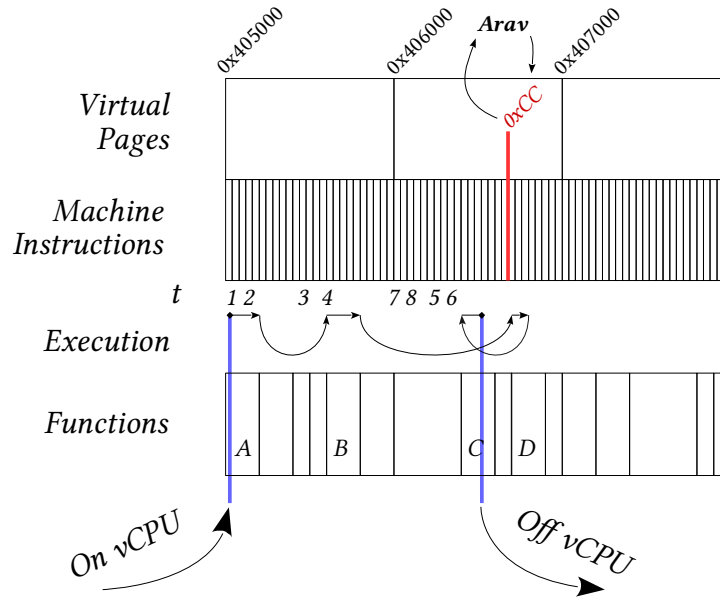


Figure 4.2 A VR process is monitored by Arav, which injects a breakpoint at the process's function D. After the process encounters the breakpoint, the VR is briefly suspended while Arav records the event.

programs often contain a global logging function, and instrumenting *just* this function brings Arav into parity with syslog-based monitoring solutions. As discussed later in §4.3.4, adequate security event coverage was achieved by instrumenting 20-30 function calls for the scenarios tested.

The list of target function names should be refined until reaching the desired accuracy. VRs can be placed in a sandbox while memory analysis tools (e.g., Valgrind) record function call traces. Arav's effectiveness is dependent on this refinement step.

This process is described in Figure 4.3, which depicts a VR's binaries being analyzed and selected function names being refined and tested, culminating with a final VR capability set. The set is then placed on the hypervisor to be used by Arav at runtime.

4.3.3 Inferring Security Events from Traces

While monitoring a VR, Arav records each encountered breakpoint by logging the current time, the guest process function called, and, optionally, that function's arguments.

The sequence of function calls made by an in-guest service may be sufficient to reconstitute a security event. For example, in the case of OpenVPN, a common SSL VPN service, if a trace contains calls to `close_instance()` and `init_instance()` within close proximity, it can be inferred that an OpenVPN tunnel was reset.

However, a series of function calls is often by itself insufficient, because, as in this example, it may be a security goal to know *which* tunnel was reset. This additional context is almost always

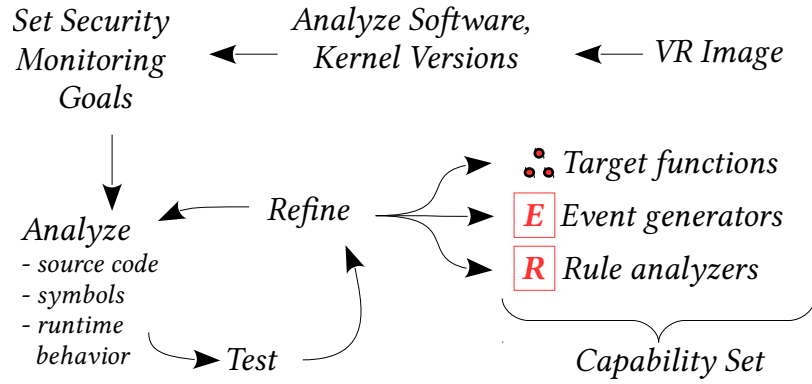


Figure 4.3 Arav's capability set construction workflow.

present in function arguments, which Arav can optionally extract from the guest's vCPU registers. In the case of the x86_64 architecture, a function's first six integer and pointer arguments are passed in registers [Lin]. Arav reads these arguments from registers, dereferences them if needed, and adds them to the trace.

As seen in Figures 4.4 and 4.5, once new function call trace data arrives from a VR, it is made available to an arbitrary number of analyst-designed event generators. The purpose of each generator is to query the trace for a certain set of function call or argument patterns. If the generator finds matches in the trace, it generates a security event.

The resulting stream of security events is made available to rule analyzers. These analyzers perform a similar function, querying the event stream for patterns and generating alerts based on rule-based thresholds. These alerts are the basis for a response action.

A VR capability set can be conceptually related with a Snort ruleset. However, unlike Snort, a Arav VR capability set is not bound by the limitations of rule or signature-based matching. Arav's Python-based generators and analyzers could alternatively be written to use anomaly detection, to query external resources, or to take the event streams of nearby VRs into account.

4.3.4 VR and Network Service Selection

Two open source VRs, pfSense¹⁰ and VyOS,¹¹ were selected to evaluate Arav's overhead and demonstrate its effectiveness. These VRs were chosen because they are commonly used in the cloud and based on different operating system families (Linux and FreeBSD). Both are also available on the AWS Marketplace. pfSense's website reports over 300k pfSense installations, and VyOS is related to several other routers, including Brocade's vRouter and Ubiquiti's EdgeOS. Table 4.2 shows the specific VR versions used in this chapter. Arav was designed to be able to monitor any Linux or

¹⁰<https://pfsense.org/>

¹¹<https://vyos.io/>

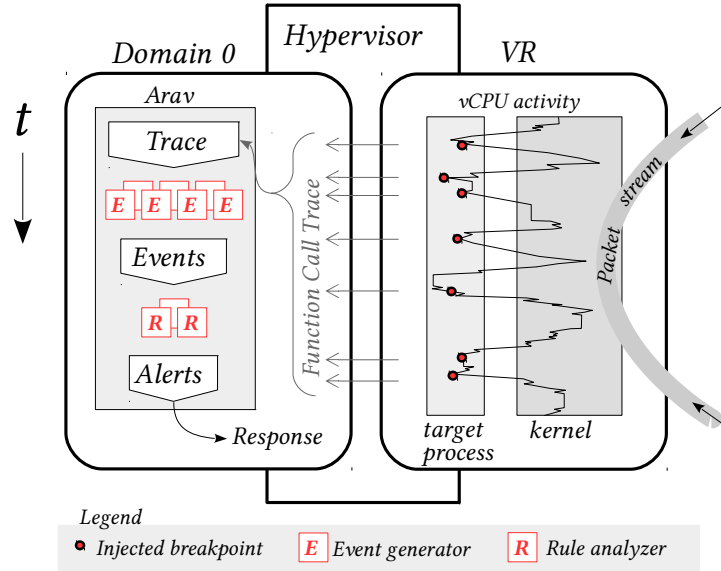


Figure 4.4 A VR process's BPs are recorded into traces, which are mapped into events and analyzed against rules.

Table 4.2 Virtual Routers (VRs)

VR	Version	Released	Base OS	Kernel
pfSense	2.2.3	10-2016	FreeBSD 10.3	10.3p9
VyOS	1.1.7	02-2016	Debian Squeeze	3.13.11

FreeBSD-based VR with minor modifications.

Each VR supports a set of common router capabilities, including support for dynamic routing protocols, VPNs, and network services. Two services were selected for Arav's development and evaluation, OpenVPN,¹² and Quagga's OSPF daemon.¹³

OpenVPN and Quagga were selected for their architectural differences. Quagga is a family of routing daemons, including RIP, BGP, and OSPF, and as a result, Quagga developers placed the bulk of its code into shared libraries, to the point where Quagga's `ospfd` executable only contains 12 functions. In contrast, OpenVPN, a popular SSL VPN daemon, has its core functionality placed in its main executable. Practically, this means Arav's implementation must handle inserting breakpoints in shared libraries as well as process code. Table 4.3 shows a comparison between the two services.

Arav's OpenVPN capability set. Arav inserts breakpoints at 21 OpenVPN functions (2% of total) in order to reconstitute 10 types of security events, including new connections, authentication attempts, broken tunnels, and connection time-outs. These security events are analyzed by 3 rules, which

¹²<https://openvpn.net/>

¹³<http://www.nongnu.org/quagga/>

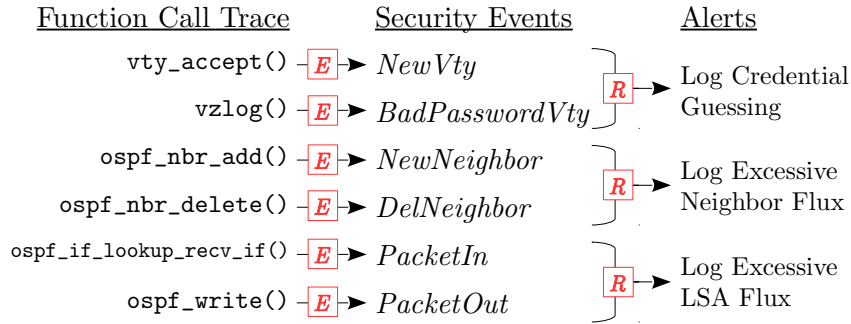


Figure 4.5 Event generators reconstitute security events from the trace of Quagga OSPF’s function calls, and these security events are queried by rule analyzers for potential responses.

Table 4.3 Target VR Services

Service VR	OpenVPN		Quagga OSPF	
	pfSense	VyOS	pfSense	VyOS
Version	2.3.11	2.1.3	20160315	0.99.20.1
Released	05-2016	05-2013	03-2016	03-2012
Disk (kB)	906	524	1,275*	892*
Mem (pgs)	220	132	298*	198*
Functions	1,015	966	2,217*	1,709*
BP Injected		21		26
Evt. Gen.		10		12
Rule Anal.		3		5

**aggregate of ospfd, libospf.so, and libzebra.so*

detect brute force connection attempts and process crashes. Arav extracts usernames, passwords (optional), X.509 certificate information, and remote IP and port information from OpenVPN’s function arguments.

Arav’s Quagga capability set. Arav inserts breakpoints at a total of 26 Quagga functions (1.5% of total) in order to reconstitute 12 types of security events, which are further analyzed by 5 rules that alert based on flooding attacks, administrative brute force attempts, or crashes. Arav extracts neighbor information, including router IDs and source IPs, and administrative usernames and passwords (optional) from Quagga’s function arguments.

4.3.5 Attacks and Responses

Both capability sets are tested against their ability to detect and respond to several types of attacks which cause VR resource exhaustion and denial of service.

The first OpenVPN attack consisted of a rapid sequence of new site-to-site connection requests.

The second OpenVPN attack was made up of multiple malicious OpenVPN clients attempting to brute force guess the authentication credentials of OpenVPN remote access users. In both cases, Arav is expected to detect the increase in activity, extract source IPs and guessed credentials from function call arguments, and determine the attacker source IP.

Two application-layer attacks were used against Quagga. The first attack consisted of a flood of fake OSPF neighbors, and the second attack was composed of hundreds of forged Link State Advertisements (LSAs). Arav should detect the in-rush of new information, sort new neighbors from existing, and log an alert.

When a capability set's rule analyzers issue an alert, they also specify the type and duration of response actions to take. Arav's initial set of response actions are to 1) log the alert, 2) install an IP block between the VR and the attacker, or 3) reboot the VR.

Logging an alert is appropriate for Quagga's application-layer attacks. Since these attacks lack a simple network-layer mitigating action, manual intervention is required, such as adding OSPF authentication to prevent future attacks.

Arav's IP block response installs a precise firewall block on the underlying hypervisor's appropriate network interface. This block specifically prevents traffic between the VR's targeted IP and the attacker, which squelches both OpenVPN attacks.

Arav should also respond when either service crashes. In this case, Arav should detect the absence of the service's original process, conduct a search to determine if the service was restarted, and if not, reboot the VR. In this way, rebooting the VR is a coarse method to recover from software runtime failures.

4.3.6 Implementation Details

Arav is designed to run as a privileged process in the Xen hypervisor's privileged domain, domain 0. Alternatively, Xen's XSM permissions allow Arav to run in a normal domU guest with a subset of hypervisor privileges.

Arav interfaces with Xen and injects breakpoints via the LibVMI library¹⁴ [Pay07]. As a part of this chapter, LibVMI was extended to inspect FreeBSD guests.

By leveraging Python's Numpy and Pandas libraries, a capability set's event generators and rule analyzers are typically each less than 20 source lines of code.

4.4 Evaluation

Arav is evaluated for its *performance*, which is quantified as the percentage reduction in a VR's aggregate packet throughput when Arav is active. A microbenchmark is also used to quantify the

¹⁴<http://libvmi.com/>

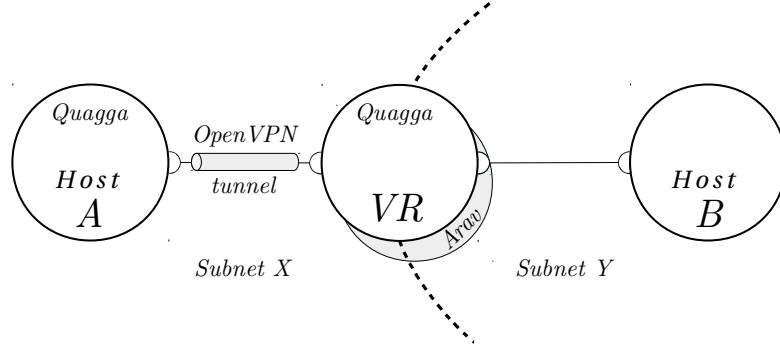


Figure 4.6 Arav's Experiment diagram. Node A is an external host, the origin of attacks, the OSPF neighbor, and a VPN client. Node B is an internal host.

impact of a single injected breakpoint.

Arav's *effectiveness* is evaluated for its recall ($\frac{TP}{TP+FN}$) in correctly identifying attacks. Arav's time-to-detect is measured as the difference between when an attack begins until the first response action is begun.

4.4.1 Experiment Environment

Figure 4.6 shows the experiment network diagram. The external *Host A* acted as a tunnel endpoint for OpenVPN and an OSPF neighbor for Quagga. This host was also responsible for attacking the VR protected by Arav. During the performance evaluation, an internal *Host B* received a file transfer from the external host.

The experiment environment consisted of a single Dell R920 hypervisor with four 2.3GHz Intel Xeon E7-4870 processors and 256GB RAM. The hypervisor ran Xen version 4.6.3 and was paired with a Fedora 24 domain 0 host.

Each VR under test was fully virtualized (HVM) and allocated 1 vCPU, 1GB RAM, 40GB disk, and two e1000 virtual NICs. The VRs were installed with default settings without VR-specific optimizations. VyOS ships with Xen's paravirtualized drivers installed and enabled by default, but pfSense does not.

4.4.2 Experiments

Throughput overhead. In this experiment, a VR's network throughput and latency are measured during a 1GB file transfer from the external *Host A* traveling through the OpenVPN tunnel, across the VR, and to the internal *Host B*. To ensure that Arav's effect on bandwidth would be emphasized in the results, the CPU and RAM capacities of hosts A and B were increased until the VR's CPU utilization was maximized during the file transfer.

Table 4.4 File Transfer Performance Tests

		Arav?	MB/s	$\Delta\%$
pfSense	1	no	14.78 ± 0.07	
	2	yes	14.69 ± 0.08	-0.57%
VyOS	3	no	21.00 ± 0.11	
	4	yes	20.87 ± 0.10	-0.63%

*1GB file transfer from Host A $\rightarrow$ B though OpenVPN tunnel.
Transfer rate shown as average with 0.95 confidence interval*

Performance. Since performance overhead is a function of the number of breakpoints a VR encounters, Arav is also measured against the microbenchmark first used by *SPIDER* [Den] to quantify the overhead of a single injected breakpoint. This benchmark recorded the number of CPU cycles required to execute a loop containing a function F which iterated a variable 1000 times. The number of loop iterations was varied from 10^4 to 10^6 in increments of 10^4 . A baseline was first established to determine the average number of cycles required to execute this loop in the absence of Arav. Arav was then used to inject a breakpoint at F 's entry point, and, with Arav running, the average number of cycles was measured again. The average difference in these two cycle counts quantified the overhead of a single breakpoint.

Effectiveness. Each attack in §4.3.5 was executed against the pfSense VR, measuring Arav's 1) detection recall and 2) the time difference between the start of the attack and the corresponding Arav alert.

4.5 Results

Throughput overhead. When Arav was active, the impact to VR network throughput was less than 0.63%. There was no measurable impact to latency. Since OpenVPN is responsible for the tunnel and the encryption and decryption of its packets, any interruption should impact throughput, and this is reflected in Table 4.4. During these trials, OpenVPN encountered one of Arav's 21 injected breakpoints at an average rate of 1/sec.

Performance. The microbenchmark experiment showed that guests were suspended for an average of $161\mu s$ while Arav processes a single breakpoint, which is comparable with prior work [Len14b; Den].

Effectiveness. Table 4.5 shows OpenVPN's time-to-detect for each attack. During initial experiments, Arav's recall statistic was influenced by false negatives, which was attributed to the way two event generators were written. The generators were refined, and Arav's final recall measurement was 1.0,

Table 4.5 pfSense Effectiveness Tests

	OpenVPN	$T_{Alert} \text{ sec}$
1	New Connection DoS	20.20 ± 21.49
2	Credential Guessing	2.31 ± 0.40
3	Process Failure	0.75 ± 0.02
	Quagga OSPF	$T_{Alert} \text{ sec}$
4	Neighbor Flood	1.02 ± 0.04
5	LSA DoS	2.27 ± 0.40
6	Process Failure	7.44 ± 1.28

Results are averaged and shown with 0.95 confidence interval.

but this highlighted the possibility that the system will miss events if generators and analyzers are not built and tested carefully. Arav did not encounter false positives, but like these false negatives, poorly written generators and analyzers could produce them.

4.6 Discussion

4.6.1 VMI Breakpoint Protection

Breakpoints injected by Arav modify VR memory, replacing one byte of existing executable code with a breakpoint. These breakpoints can be detected by the VR, both by anti-debugging methods [Che08] and code integrity checkers such as kernel patch protection schemes. Prior VMI work was focused on dynamic malware analysis and took effort to prevent a guest and its malware from detecting injected breakpoints [Len14b; Den]. These protections concealed or temporarily removed the breakpoint while the guest was reading through its memory. Although these protections could be adopted in a future Arav version, Arav does not currently protect its breakpoints in order to avoid any associated performance penalty.

4.6.2 Analyst Effort with New VR Releases

The entire analytic process described in §4.3.3 does not need to be repeated when a new VR version is released. Arav only needs access to the new VR's debugging symbols.

Generally, function names could potentially drift between versions. Functions could be refactored, renamed, or removed. Despite this possibility, it was not encountered in this chapter. pfSense's and VyOS's versions of OpenVPN and Quagga are approximately 4 years apart, but the same capability sets could be used on both, indicating that the functions correlated to security events are not changed often.

4.6.3 Limitations

4.6.3.1 Mapping Memory Addresses to Functions

The mapping from a function name to its future location in memory allows breakpoint injection events to carry meaning. One straightforward, compiler-agnostic method to extract this mapping is by obtaining VR binaries with debugging symbols. These symbols associate each function name in a program with its offset within the binary. Many open source VRs make their debugging symbols available, or these symbols can be re-created. However, for proprietary VRs, or VRs in which re-creating binaries with debugging symbols is not possible, other analytic methods must be considered [Shi15; HM].

4.6.3.2 Mapping Functions to Events

The process by which Arav’s breakpoint locations are identified, tested, and validated involves manual analyst effort. This process can be time-consuming, especially if a monitored service’s source code is difficult to understand. During development, OpenVPN’s capability set was developed before Quagga’s and took an estimated time of 6 hours. Once the process became better practiced, Quagga’s capability set was developed in half that time. The automatic production of Arav’s capability sets is left for future work.

4.6.4 Related Work

Virtual Routers are often associated with Network Function Virtualization (NFV), which Bays *et al.* [Bay15] segmented into three types: protocol-based, machine virtualization-based (this chapter’s assumed environment), and programmable networks. Detecting problems in VRs can be done in a variety of ways, including work to protect the overall system from any misbehaving VRs [FD11; Egi]. Recovering a VR via reboot is relatively primitive. More sophisticated recovery methods include checkpoint-restart mechanisms for individual VMs [Cui15] and for groups [Cui16].

VMI techniques generally consist of a core set of primitives, allowing a VMI application to read from and write into a guest’s registers, physical memory, and virtual memory. For example, *Near Field Monitoring* [Sun14] aims to achieve similar monitoring goals using only memory reads, and in this way does not *interpose*, or intercept guest actions, instead relying on polling guest memory for changes. Over time, security researchers have uncovered more powerful VMI techniques, often as the result of clever re-use of processor hardware features. Ultimately, since VMI was first explored by *Livewire* [GR03], improvements in processor hardware have also improved VMI’s ability to interpose on guest execution. For example, *Ether* [Din08] used EPT violations to receive notifications when a guest accessed certain memory pages. In comparison, breakpoint injection allows even greater granularity. Early works using VMI to inject breakpoints include *XenProbes* [QS07]. The technique

was refined and strengthened by *SPIDER* [Den] and *DRAKVUF*, which inserted breakpoints at 11,011 Windows function calls, protecting the breakpoints with an EPT violation-based scheme. Breakpoints are typically associated with debugging. Debugging frameworks, such as *Stackdb* [Joh14], also use this technique.

Effort has been made to integrate VMI with cloud environments [Bae14], including *NvCloudIDS* [Mis16], which leveraged breakpoint injection to combine a server’s system call traces with network packet captures to detect malicious binaries. In contrast, by monitoring process-level function calls, Arav observes richer application-specific context unavailable in system call traces.

4.7 Summary

This chapter emphasized the importance of virtual routers in today’s cloud ecosystems. Despite the vital role VRs play, they are often difficult to monitor. This chapter introduced Arav, an elegant VMI-based approach which satisfies desirable characteristics for a VR security monitoring system. Arav traces a VR’s function calls using breakpoint injection, and reasons about these traces using analyst-provided querying and filtering algorithms. Through performance and effectiveness evaluations, Arav showed an acceptable overhead in several use cases involving two common VRs running two common network services. Arav detected all four attacks used to evaluate it, demonstrating the feasibility, effectiveness, and potential of this approach.

CHAPTER

5

DEVELOPING HYPERVISOR-BASED AGENTS

The previous two chapters documented efficient designs for hypervisor-based security systems. Using these chapters as a foundation, this chapter introduces a straightforward, approachable methodology to integrate hypervisor-based security into existing security software. The ease at which HBS can be integrated is important for the broader adoption of hypervisor-based security. This chapter's methodology describes and evaluates one such integration approach that focuses on equipping third-party software daemons with VMI capabilities.

These third-party software daemons, called host agents, are increasingly responsible for a modern host's security, automation, and monitoring tasks. Because of their location within the host, these agents are at risk of manipulation by malware and users. Additionally, in virtualized environments where multiple adjacent guests each run their own set of agents, the cumulative resources that agents consume adds up rapidly. Consolidating agents onto the hypervisor can address these problems, but places a technical burden on agent developers.

Through this development methodology, agent developers can predictably and reliably re-engineer a host agent in to a *hyperagent*, an out-of-guest agent that gains unique hypervisor-based advantages while retaining its original in-guest capabilities. This three-phase methodology makes integrating Virtual Machine Introspection (VMI) functionality in to existing code easier and more accessible, minimizing an agent developer's re-engineering effort. The benefits of hyperagents are

illustrated by porting the GRR live forensics agent, which in return for a modest growth of 11% of its original codebase, used 40% less memory than its in-guest counterparts and enabled a 4.9x speedup for a representative data-intensive workload. This chapter shows that a conventional off-the-shelf host agent can be feasibly transformed into a hyperagent and provide a powerful, efficient tool for defending virtualized systems.

5.1 Introduction

Third-party host agents are pervasive in modern computing, where it is common practice to install multiple adjacent agents on production systems to monitor performance [Ace13], conform to regulations and industry standards [Gik10], and scale system administration [Del10]. These agents perform their duties locally on a host on the behalf of a centralized management server. Agents specializing in security-related areas such as intrusion detection, policy enforcement, forensics, and incident response are employed at a system's highest privilege level, where they have unrestricted access to the host, can intercept and mediate undesirable activities, and resist subversion.

However, even when agents are running as kernel modules or privileged processes, they can still be subverted through vulnerabilities found elsewhere in the kernel [Jai14]. Although more privileged hardware-enforced modes are available (e.g., virtualization and processor management modes), their use is uncommon in production due to framework immaturity, performance, perceptions of complexity, and uneven hypervisor support [Jai14]. Integrating an existing agent with one of the few security frameworks that does take advantage of these hardware-enforced modes is complicated by the requirement to re-write agent code against unique APIs [Joh14; Len14b; Vas13]. Therefore, despite the potential benefits of running in the hypervisor, host agents have continued to be implemented on the “guest side” of the virtualization layer.

This chapter presents a methodology to assist agent developers in making a simple, well-defined set of changes to port an existing security agent to the hypervisor. Once ported, the agent is completely removed from the guest, and instead performs the same functionality as it did “in guest” from a layer below. This methodology relies on the observation that the core functionality of a host agent does not rely upon which side of the virtualization layer it executes in. As a result, porting only requires that the agent code that interacts with host APIs (this chapter terms these *actuators*) be replaced with hypervisor-based Virtual Machine Introspection (VMI) equivalents.

The porting methodology has three software development phases:

Phase 1: An agent's source code is examined to identify code sections that interact with the host. These sections are refactored to make use of `libhyperagent`, a library we specifically designed to simplify the integration of VMI functionality into an agent.

Phase 2: Agent code is further modified to allow multiple agents, for different guests, to execute under the control of a single process (the hyperagent) in the hypervisor.

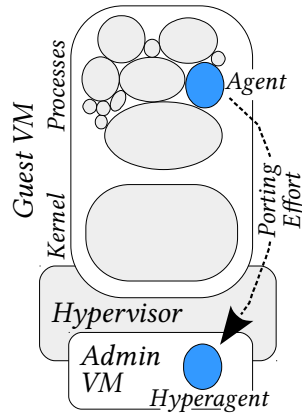


Figure 5.1 In-guest agents can be ported into hyperagents.

Phase 3: New features are built into the hyperagent to take advantage of its environment. These features take advantage of the broad cross-VM view of a hyperagent and new opportunities to move server-side work forward to the hyperagent.

The proposed methodology is illustrated by porting the GRR Rapid Response (GRR) [Grr] live forensics C++ agent into a hyperagent. This GRR hyperagent is compared with its in-guest counterpart in terms of the developer effort required to port the agent's code, resource usage, execution time, and performance. The evaluation is conducted on two platforms: the Xen hypervisor and Qubes. This chapter also ported the ClamAV virus scanner and discusses how three other agents could be re-engineered.

This chapter makes the following contributions:

1. A development methodology to convert host agents into hyperagents that execute in the hypervisor's administrative domain.
2. A VMI library of useful hyperagent functions, including a new VMI technique to sync a guest's cached disk writes.
3. The implementation and evaluation of the GRR hyperagent.

5.2 Motivation

This chapter presents a methodology that assists agent developers with the technical hurdles associated with the re-engineering, or *porting*, of a third-party host agent, ensuring the agent can do the same job executing under control of the hypervisor as it could under control of a guest OS (see Figure 5.1). There are several compelling benefits to re-engineering a host agent in this manner [GR05], as described below.

Better isolation. It is more difficult for malicious code in a guest to both identify and resist a hypervisor-based security agent. The hardware-enforced isolation and smaller exposed interface between a hypervisor and its guests is also more difficult to subvert than the interface between a user process and its kernel [GR03; Jai14]. An agent running beneath the guest is both isolated (and to a degree [Din08], concealed) from the guest and operates at a higher privilege level than the guest kernel, so the agent is less likely to be tampered with or disabled by users or malware.

Enhanced agent capability. An agent running on the hypervisor combines the best architectural strengths of both host agents and network security devices. A hypervisor-based agent has the host-level advantages of fine-grained visibility into a host and the ability to interpose on guest execution (e.g., hook and mediate host actions), but also the network-level advantages of broad visibility across hosts, independence from host influence, and centralized administration.

From its position beneath guests, a hyperagent takes advantage of two concepts unavailable to in-guest agents. The first is *locality*, where a more trusted hyperagent performs analysis that is normally kept server-side. This delegation to the hyperagent cuts out network round trips and speeds up analysis by co-locating the analysis with VMI-acquired guest data. The second is *cross-VM visibility*, where work is consolidated and executed centrally at the hyperagent instead of once-per-guest as is the case with in-guest agents.

Resource efficiency. Collapsing multiple per-guest copies of a normal host agent into a single hyperagent reduces the system's overall memory footprint.

Centralized administration. In a guest-dense environment with tens-to-hundreds of guests per hypervisor, administrative effort is reduced if a single hyperagent replaces many guest agents.

Additionally, a hyperagent can be managed independently of the virtual machines it inspects [GR05]. The agent can be activated, reconfigured, or deactivated without regard to the guest's state [Fu16].

5.3 Background

This section introduces the concepts upon which this chapter is based, including agents, VMI, and live forensics, then concludes with assumptions and threat model.

5.3.1 Host Agents

A *host agent* is a third-party software daemon or kernel module that acts on behalf of an external agent *controller* (e.g., server) to access or control local host resources. Using agents, the agent controller can gather or change local system state, update or back up files, start or stop processes, and control policy enforcement. An *agent-based system* includes the agents, the controller (i.e., agent server), and the protocols by which they communicate.

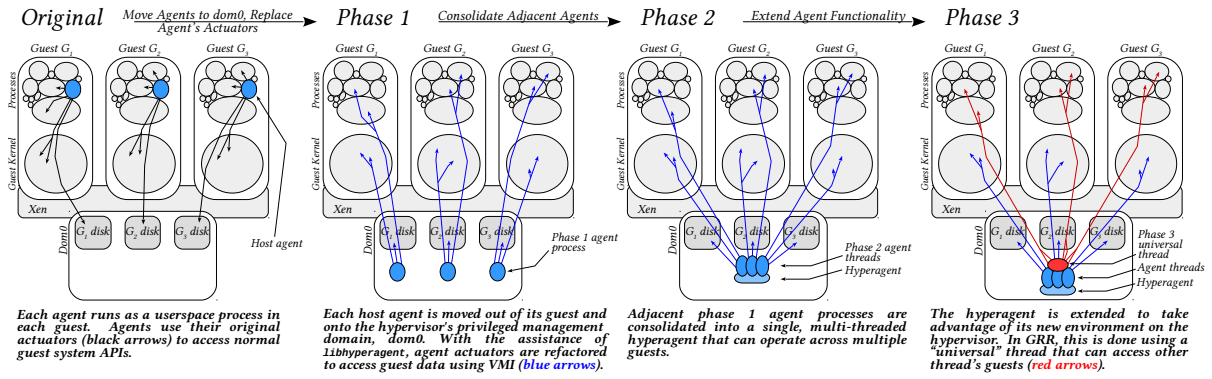


Figure 5.2 The three-phase methodology to re-engineer host agents into hyperagents.

Table 5.1 Example agents.

Category	Examples
Security	OSSEC, ClamAV, and centrally-managed anti-virus systems
Forensics	GRR, Mozilla MIG, Access Data agent, Encase Enterprise servlet
Monitoring	Nagios NRPE, Zabbix, Ganglia
Cloud Mon.	AWS CloudWatch, GCE Stackdriver, DataDog
Cfg. Mgmt.	SaltStack, Puppet, Chef
Hypervisor	QEMU guest agent, VMware tools, Hyper-V LIS, Xen/Qubes util.

Security-focused host agents in particular are designed to extend or supplant the host operating system's security functionality. These agents often consist of components which proactively hook into the operating system in order to intercept and mediate actions, such as preventing the execution of malicious code.

System administrators are often compelled to run multiple agents on the same host. For example, compliance requirements often mandate that a security agent (such as an anti-virus product) protect each host, a host configuration management agent may be needed to allow remote, centralized administration, and a monitoring agent may be needed to ensure service level agreements are met. Table 5.1 lists several categories of agents that may be required, with examples of each.

The impacts of these agents on resource utilization can quickly add up. In a virtualized environment, that cost is multiplied by the number of virtual machines that are executing in parallel.

5.3.2 Hypervisor-based Security Agents

Hypervisor-based security agents often use *Virtual Machine Introspection* (VMI) to inspect and influence guests. VMI is generally defined as the direct external observation of a guest's internal state [GR03]. This *out-of-guest* (sometimes called *agentless*) approach enables security functionality at a privilege level higher than the guest kernel. VMI and clever re-use of processor virtualization extensions has been recently leveraged for such purposes as malware analysis [Len14b], foren-

sics [Mar10], debugging [Joh14], and browser security [Cri16].

In the context of Xen, the hypervisor used in this chapter, running “on the hypervisor” means executing from Xen’s privileged management domain, dom0.

An agent developer seeking to leverage VMI for hypervisor-based security would survey the available frameworks and likely conclude that the porting effort either must require extensive knowledge of low-level guest OS details or require significant refactoring to conform their agent into a VMI framework’s API such as those provided by *DRAKVUF* [Len14b], *XMHF* [Vas13], and others [Joh14; Sun14].

5.3.3 Live Forensics

The methodology introduced in this chapter is validated by applying it to the GRR live forensics agent [Grr]. Live forensics is the ability to access volatile forensic artifacts in a potentially compromised machine while it is still running. These artifacts include its network connections, opened files, and userspace processes [Lig14; Coh11]. Live forensics can be contrasted with dead box forensics, which focuses on analyzing data on persistent storage.

Software-based live forensics is accomplished using one of two general approaches. Either privileged forensics software is preemptively installed on the host (e.g., GRR), or the host is *hyperjacked* [Rut06], transplanting its OS in-place into a virtual machine on top of a specially-built forensics hypervisor (an on-the-fly approach introduced by *HyperSleuth* [Mar10]).

VMI is often used by forensics tools [Lig14]. Examples include combinations of LibVMI and Volatility by *Dolan-Gavitt et al.* [DG11a] and *MiniSecHV* [Lut16].

5.3.4 Assumptions and Threat Model

This chapter considers an adversary who seeks to gain control of a host by exploiting a weakness in userspace software and escalating to root via a kernel vulnerability. The adversary establishes persistence on the host using a rootkit with anti-forensic and anti-debugging capabilities and has awareness of how security agents are designed.

Our approach assumes the integrity of the hardware, the hypervisor, and the agent system.

It is advisable for security reasons that the ported agent code remain outside of the hypervisor’s trusted computing base (TCB). Our work is based upon the Xen hypervisor, where isolating a hyperagent would be accomplished using XSM [Spe99] and others [Tau16] to encase the hyperagent in a privileged VM. This has been shown to exert acceptable overheads ($\leq 3.0\%$ [Tau16]). Hyperagent code could also be sandboxed (e.g., chroot, seccomp-bpf, and Linux namespaces) or otherwise constrained by the dom0 OS.

Some of the hypervisor-based techniques used to replace an agent’s actuators depend on guest OS invariants [Wan15a] that implicitly assume that the guest kernel remains in a state where the

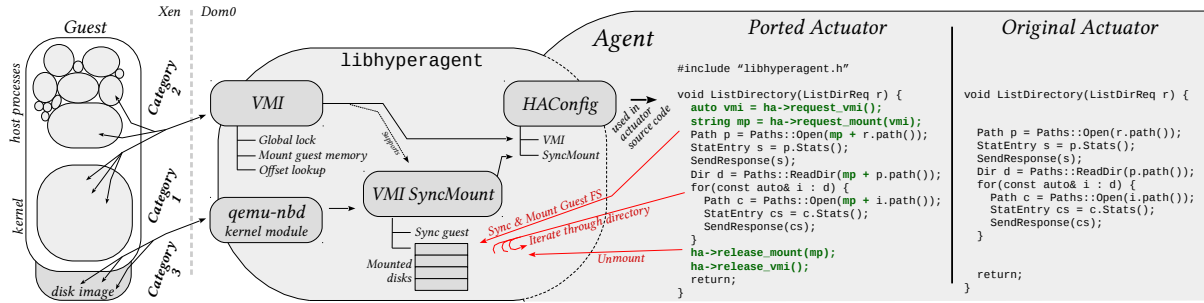


Figure 5.3 Adding libhyperagent enables agent actuators to access a guest’s kernel data structures (category 1), process memory (category 2), and file system (category 3). Using libhyperagent’s VMI SyncMount component, this example GRR actuator iterates through a guest directory’s contents by temporarily mounting the guest file system into dom0.

invariants hold [Bah10; Len14a]. Guaranteeing this assumption is equivalent to solving the strong semantic gap problem [Jai14], an open problem out of the scope of this chapter. We assume, as do others, that only a small fraction of malware samples are capable of identifying and deliberately manipulating the specific OS invariants relied on in this chapter.

5.4 Design and Methodology

This section introduces a three-phase methodology for porting host agents across the virtualization barrier:

Phase 1. The agent is moved out of the guest and into the hypervisor’s administrative domain (dom0). A LibVMI-based [Pay07] VMI library, called libhyperagent, is used to selectively replace agent components with their hypervisor-based equivalents. This grants an agent with the **higher privilege, isolation, and concealment** security benefits associated with an out-of-guest approach.

Phase 2. The multiple adjacent phase 1 agent processes (one per guest) now running in dom0 are collapsed into threads inside a single multithreaded hyperagent process. This consolidation makes **more efficient use of memory**, and facilitates the next phase.

Phase 3. New features are added to the hyperagent to take advantage of functionality offered by its new hypervisor environment. By exploiting locality and cross-VM visibility, the hyperagent is able to **outperform in-guest agents**.

Figure 5.2 depicts these phases. The term *hyperagent* is distinguished from an agent at the conclusion of phase 2, and is taken to mean a single process running and controlling multiple agent threads within the hypervisor.

5.4.1 Methodology Goals

The proposed methodology has three goals. The first goal is to faithfully preserve the original agent's functionality. Any action the agent was able to take inside the guest as a standalone process should also be possible by the hyperagent. The second goal is to minimize the amount of source code changes required during re-engineering, especially changes that require specialized VMI expertise. The final goal is to measurably improve the agent's effectiveness and efficiency.

5.4.2 Phase 1: Porting Agent Actuators

In phase 1, portions of an agent's source code are refactored to use VMI so that the agent can be moved to the hypervisor and inspect and influence a guest virtual machine. With Xen, dom0 is typically Linux. Once a few necessities are resolved (e.g., library dependencies and network connectivity) a host agent will run equally as well in the hypervisor as it would in any Linux-based guest.

Once on dom0, the agent's actuators must be refactored to use VMI techniques. An agent's actuators are where its purpose is realized. For example, if an agent exists to read a local file and send its contents to the agent controller, the block of agent code that reads the file would be considered an actuator. Examples of actuators include accessing file systems, making system calls, and using `ptrace` to access data in other processes.

A developer must identify these actuator sites in the agent's source code. Our experience has shown that code modularity and common software design patterns makes actuators straightforward to identify. For example, actuators may occur in all derived classes of a certain class, or actuators might only occur in source code files containing unique imported libraries (e.g., an actuator that uses `ptrace` imports the `ptrace` header file).

This chapter introduces `libhyperagent`, a LibVMI-based [Pay07] memory and guest file system introspection library that presents an API for hypervisor-based agents to use. Through `libhyperagent`, actuators can interact with guest kernel data structures, access a guest process's memory, and read from the guest file system. Internally, `libhyperagent` enables each of these categories of capabilities using a combination of VMI techniques as described below.

Category 1: Kernel data structures. Normal host agents typically access kernel data structures by interacting with `/proc` or `/sys`, two pseudo file systems that essentially represent useful kernel data structures as files and directories that are easily accessible by userspace programs.

In the case of VMI, kernel information is instead accessed directly in guest memory, based on location information gleaned from the guest's kernel debugging symbols. For objects stored common kernel data structures such as linked lists and binary trees, the typical VMI approach is to find the exported kernel symbol address for the "anchoring" statically-addressed kernel object, then to traverse object pointers on the kernel's heap (much like the kernel does) until reaching the

desired kernel object instance [Pay07; Len14b].

Category 2: Process memory. Kernel memory objects can be predictably accessed via debugging symbols made available by the OS distribution, but the analogous symbols for userspace processes may not be available [Bus]. The approach normally taken by forensics software is to carve up process memory into its memory areas (as defined in kernel data structures). These areas are then accessed by the agent without further VMI-specific semantic interpretation.

Category 3: File system operations. Host agents routinely interact with a host's file systems. This may be necessary both to fulfill the agent's specific purpose or for tasks such as loading the agent's configuration file and writing log files. When porting the agent on to dom0, it is challenging to provide equivalent access to the guest's file system while minimizing the work required to refactor the agent's actuators. In the worse case, agent actuators might have to be completely re-written against a disk introspection API (e.g., libguestfs¹).

The approach taken by libhyperagent is to shadow mount the guest file system locally on dom0, essentially causing the guest file system to be mounted a second time (the file system has already been mounted once by the guest OS).² As a result, actuator code that performs file system operations needs only to be directed to the guest file system's local mount point in dom0. Figure 5.3 shows the minor changes required to refactor an actuator that lists the contents of a directory.

Since the guest has already mounted (and is actively using) the same file system, this mounting process must be transparent to the guest and should be read-only.

Solving the problem of delayed writes. Ensuring an up-to-date and consistent view of the guest file system is complicated by the layers of caching performed by modern operating systems. Guest writes are often delayed before being committed to disk, creating a temporary inconsistency between in-guest and out-of-guest views. To flush writes to disk and avoid this problem, a hyperagent needs the ability to perform the equivalent of `sys_sync`, the Linux kernel's `sync` system call, immediately prior to an actuator's work in the guest file system. Since synchronizing to disk must be blockable, any VMI technique seeking to induce `sys_sync` must avoid interacting with the guest kernel at an undesirable time, such as when interrupts are disabled.

To flush outstanding writes to disk, we propose a new VMI technique for Linux,³ called *Work Queue Interception (WQI)*. Inspired by *SYRINGE* [Car12], WQI is useful because it does not introduce new code into the guest, is simpler than stack-based code-injection [Len14b] or syscall redirection [FL13], and can be reliably triggered by the hyperagent. While WQI is specifically used here to call `sys_sync`, it is general enough to be used to call other functions elsewhere in existing kernel code.

¹<http://libguestfs.org/>

²For raw and QCOW2 file-based image formats, libhyperagent uses `qemu-nbd`, a Linux kernel module, to mount guest file systems in dom0.

³Although this implementation is Linux-specific, Windows has equivalent *work items*.

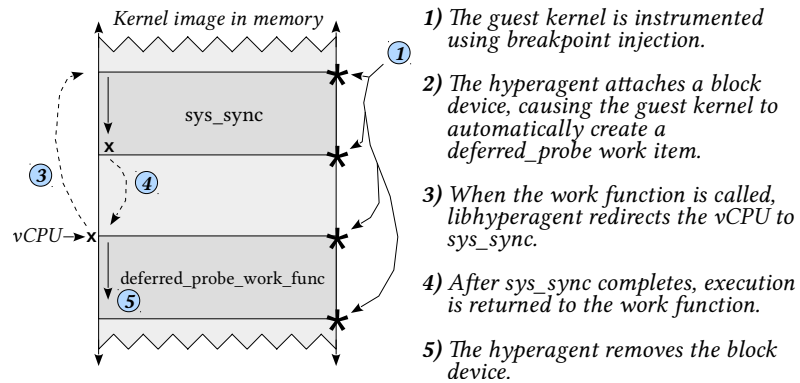


Figure 5.4 The predictability of work queue behavior enables the hyperagent to divert a guest vCPU to `sys_sync`.

Work queues are usually used by kernel interrupts to perform low-priority work outside of interrupt context [Lov10]. They are interesting with regard to guest file system introspection because they allow arbitrary work to be performed in kernel process context, in which `sys_sync` can be safely executed. During normal use, when work is to be added to a work queue, a *work struct* is allocated and populated with the kernel function pointer to be called. Later, a kernel *work process* will run down its work queue and dutifully call each work struct function pointer.

An examination of the Linux kernel identified a work item that can be reliably triggered by the hypervisor. It happens that the `deferred_probe` work item is generated by kernel device hotplug code when a new block device is attached to a guest (this same code runs when a USB device is inserted into a physical machine). `libhyperagent` uses VMI to interrupt the guest when this work item is processed in order to redirect the guest vCPU to `sys_sync`. The procedure is described below in detail and illustrated in Figure 5.4:

1. `libhyperagent` configures the guest vCPU to trap to the hypervisor when an injected breakpoint is encountered at the beginning and end of each `sys_sync` and `deferred_probe` function call.
2. `libhyperagent` attaches a dummy block device to the guest, inducing the guest kernel into enqueueing a `deferred_probe` work item.
3. A guest kernel work process calls `deferred_probe`. At the first instruction of this function, the guest vCPU encounters the injected breakpoint and traps to the hypervisor. `libhyperagent` saves the vCPU's register state, then directly alters the guest vCPU registers to jump to `sys_sync`.
4. As `sys_sync`'s breakpoints are encountered, `libhyperagent` receives confirmation the guest disks are synchronized. `libhyperagent` returns control to the calling agent actuator so that it can perform work on the guest file system. Afterwards, `libhyperagent` restores the original vCPU state and allows `deferred_probe` to execute normally.

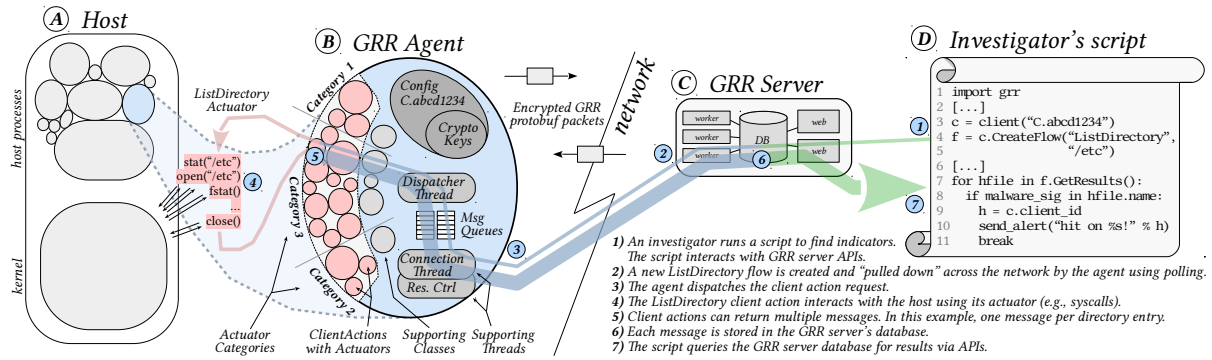


Figure 5.5 The GRR architecture. From left to right: (A) a host contains a kernel and userspace processes. One of these processes is (B) the GRR C++ agent. The agent registers with (C) the GRR server across the network. (D) An investigator's script interacts with GRR server APIs to create a *flow* that directs the agent to perform one or more of its client actions.

5. Finally, `libhyperagent` detaches the block device, returning the guest to its original state.

Phase 1 summary. At the conclusion of the first phase, the agent's actuators have been replaced with their out-of-guest equivalents made available by the `libhyperagent` library. The agent can then run inside the hypervisor, and connect to the (unchanged) agent controller as before. With the exception of the VMI-enabled actuators, the body of the agent behaves precisely as it did inside the guest.

The primary benefit of this phase is security. The agent is now isolated and concealed from the guest. As an additional benefit, because each guest's phase 1 agent now runs inside `dom0`, agent management is centralized within the hypervisor.

5.4.3 Phase 2: Consolidating Adjacent Agents

Modern hypervisors often host many guests, which after phase 1 would mean that multiple agent processes run inside the hypervisor. It is intuitively appealing to eliminate redundancy between multiple agent processes by consolidating them into a single process.

A simple solution is to encapsulate the original agent's functionality in a thread, running as part of a new multi-threaded process, termed the hyperagent. Each time an agent is to be ran for a guest, this process creates a new agent thread. The translation from process to thread requires a management mechanism to spawn and supervise agent threads. This approach eliminates process-level overheads and allows agent threads to share the expensive VMI-related library objects.

Depending on how the agent is ported, synchronization mechanisms may be needed to mediate each agent thread's access to shared objects. The scope and effort required to add these synchronization mechanisms is dependent on how the agent was originally designed. `libhyperagent`

itself is thread safe and handles synchronization internally.

Phase 2 summary. The immediate benefit of phase 2 is memory savings, achieved by replacing independent agent processes by threads.

5.4.4 Phase 3: New Agent Functionality

In addition to the described security and efficiency benefits, the hyperagent can support powerful new features that take advantage of its new position beneath its guests.

Compared to an in-guest agent, a hyperagent is dramatically less visible to in-guest malware, and is difficult to affect even if detected. This allows the hyperagent to safely assume functionality that is normally executed by the agent controller. Additionally (from phase 2), the hyperagent has the ability to monitor and control the agent threads for all guests, further motivating a migration of functions from the server to the hyperagent.

While the specific features introduced in phase 3 largely depend on the agent's purpose, in general, these features take advantage of a combination of two related ideas:

Locality refers to the benefits of migrating work from the server to the hyperagent. This delegation of work can help eliminate processing bottlenecks at the server. In addition, communication with the server is reduced, saving network bandwidth and eliminating some round-trip latencies. The combination of these makes hyperagents more attractive for applications requiring quick response times.

Cross-VM visibility refers to benefits achieved when a single hyperagent can interact with multiple guest VMs. This direct, cross-VM access can benefit any application that aggregates agent information. The hyperagent can compute and communicate to the server only aggregated results, rather than raw data collected from the individual VMs.

Due to its position on dom0, a hyperagent also has the potential to support VMI applications described in prior VMI work [Bau15] and link into the hypervisor APIs responsible for controlling guests (e.g., to roll back a guest when a security problem is detected).

New functionality introduced in phase 3 should be a well-behaved extension of the original agent system. Migrating server functionality to the hyperagent requires understanding the server's purpose and organization, and requires changes to the server as well. The challenge becomes introducing these new features in a way that preserves the norms and internal rules of the agent system's design.

Phase 3 summary. Phase 3 introduces new agent features which leverage the greater access and scope provided by the hyperagent environment, including synchronous local access to each guest and a suitable platform for moving analysis to the hyperagent.

Table 5.2 A matrix of GRR agent actuators (columns) matched against the libhyperagent components (rows) used to function on a guest.

<u>Actuator</u>	EnumerateFilesystems EnumerateInterfaces ListProcesses GetPlatformInfo DumpProcessMemory SysDiff* ListDirectory StatFile GetInstallDate EnumerateUsers Find FingerprintFile GetConfiguration GetClientInformation DeleteGRRPVersions Grep TransferBuffer																	
#	1	2	3	4	5	-	6	7	8	9	10	11	12	13	14	15	16	17
Kernel	■	■	■	■	■	■												
Process					■	■												
Disk	■					■	■	■	■	■	■	■						
Minor													■	■	■	■	■	■

*SysDiff is a new GRR flow enabled by phase 3.

Minor: Actuator required few changes.

5.5 Re-Engineering GRR

The previous section described a methodology for porting agent system functionality into the hypervisor. In this section, the methodology is illustrated by re-engineering the GRR live forensics C++ agent in to a hyperagent.

Figure 5.5 shows GRR’s client-server model where GRR agents are installed across an enterprise on workstations, servers, and virtual machines. These agents run as daemons on each host and poll the GRR server for work. Forensics investigators use the GRR server to query the agent “fleet” during an investigation.

Each agent can perform a set of primitive tasks (called *client actions* [Coh11]) to collect evidence, triage, and respond to an incident. These client actions are triggered remotely by *flows*, pre-built GRR server procedures that are invoked by an investigator’s analysis scripts. An example GRR flow, ListDirectory, is shown in Figure 5.5.

GRR supports two types of host agents. The first, written in Python, is the default agent used in nearly all cases, and is compatible with Windows, Linux, and OS X. The second, an experimental C++ variant built for a minimal footprint, is limited to Linux. To demonstrate the methodology, the C++ agent was ported into a hyperagent.

5.5.1 Phase 1

In this phase, each of the GRR agent’s actuators were identified and modified to perform the same in-guest function using VMI. The rest of the agent remained as before, and the GRR agent configuration file was unchanged.

The GRR C++ agent's actuators all inherit from a single class that defines the agent's client actions. In the source files for these child classes, `libhyperagent` was included. The actuator was either replaced with equivalent VMI code (such as when accessing guest userspace or kernel memory) or wrapped in `libhyperagent` function calls that `SyncMount` a guest file system. Figure 5.3 shows how an actuator (the `ListDirectory` client action) was wrapped in a call to `libhyperagent` to mount a guest's file system. Table 5.2 shows the results of phase 1 porting of the GRR C++ agent's 17 client actions.

5.5.2 Phase 2

The objective of the second phase is to replace agent processes with threads running inside a single consolidated hyperagent process.

The GRR C++ agent's main class, called directly by the agent's main function (`int main()`) in the original agent, was reimplemented to instead be started as a thread. Each agent's main class was self-contained (i.e., did not use global variables or shared objects) so did not require additional locking mechanisms. A new management process (the hyperagent) was implemented to spawn these agent threads according to a configuration file. This configuration file identifies the guests to be monitored, their disk path, and their kernel symbols (all required for VMI).

5.5.3 Phase 3

This phase integrates a new GRR workflow that takes advantage of the hyperagent's new position on the hypervisor to dramatically increase the efficiency of a common incident response task.

New feature integration. Before adding any new features, the GRR hyperagent requires a workable framework for adding new code into the established GRR system. This was accomplished by defining an additional agent thread called the universal thread (UT). This thread accommodates new phase 3 features as its client actions.

The UT is an entity designed to work within the GRR system to house new hyperagent functionality, and appears to the GRR server as an agent running alongside other normal agent threads. When an investigator wishes to use a new feature introduced in this phase, the investigator includes a call to the UT and its desired client action in their GRR server script.

Any server-side analysis code migrated to the hyperagent gains *locality*, and can now access guest data directly. To further enable the concept of *cross-VM visibility*, the UT is granted access to the data structures of every other agent thread in the hyperagent, allowing it to “borrow” the VMI and guest file system introspection handles of adjacent (normal) agent threads in order to work with their assigned guests.

The functions of the GRR server that will gain the most from migration to the hypervisor are those that involve substantial data transfer from multiple guests. With slight modifications the

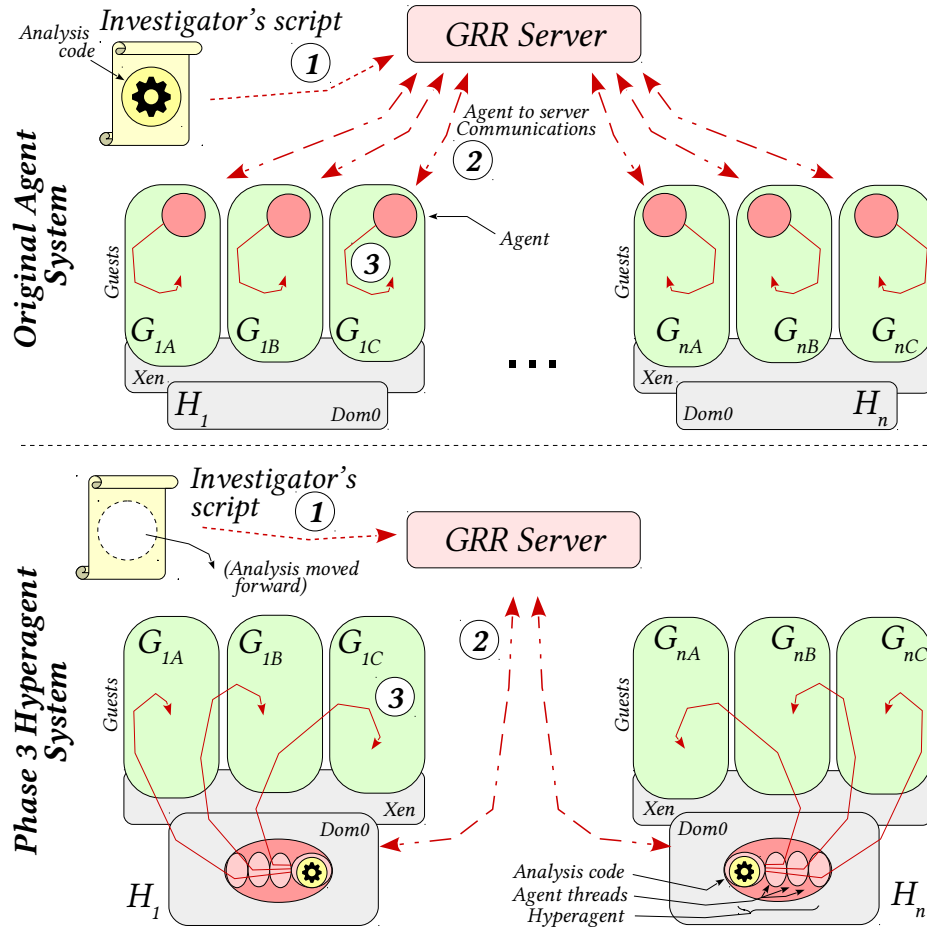


Figure 5.6 New phase 3 features take advantage of *locality* (moving work forward to the hyperagent) and *cross-VM visibility* (centralized work against all guests).

analysis portions of these scripts can be executed on the hyperagent instead.⁴ GRR investigator scripts were examined for such cases.

Figure 5.6 shows evidence of this migration of work. ① In the original agent system in the top pane, an investigator's script performs high-level analysis by directing the GRR server to issue a series of flows to each agent. In the bottom pane, this high-level analysis will instead be performed directly by the hyperagent using a single GRR flow. ② The GRR server signals agents to initiate each flow's corresponding client actions. In the bottom pane, because there are fewer hyperagents (1 per hypervisor) responding to a single flow, network traffic is greatly reduced. ③ The agent's actuators interact with the guest and perform the desired task.

The SysDiff flow. To illustrate phase 3, we introduce the SysDiff flow, a hyperagent-unique GRR flow

⁴To avoid re-writing the script in C++, the UT can invoke it through an embedded Python interpreter linked into the hyperagent for this purpose.

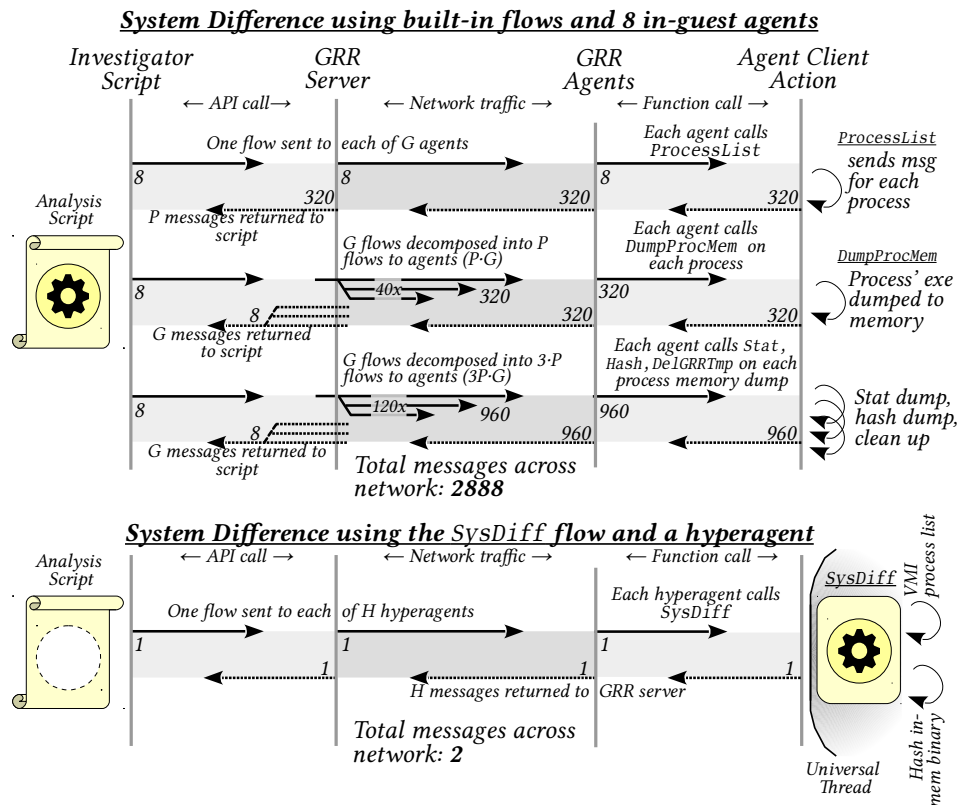


Figure 5.7 Comparing network activity between a script using the built-in GRR flows (top) and the UT performing computation forward on the hyperagent (bottom). By moving server-side analysis code forward to the hyperagent, multiple network round trips are avoided. Message counts in this figure are based on a single hypervisor ($H = 1$) running 8 guests ($G = 8$), each with 40 processes ($P = 40G = 320$).

that detects anomalous processes across a set of guests. The SysDiff flow implements the common forensics task of identifying hosts running unique (i.e., potentially anomalous) processes. For this purpose, the in-memory executable sections of each guest’s running processes are collected, and a hash for each page is computed. These hashes can then be compared to detect differences between guest processes; two processes are considered identical if their intersecting set of resident page hashes match.

For example, if three guests were running the same process binary, but a fourth guest ran an altered version, SysDiff might produce the following output:

Exe SHA256:	Exe Path:	Exe running in:
a9b8c7d6e5...	C:/Windows/System32/notepad.exe	vm-1, vm-2, vm-3
e5d4c3b2a1...	C:/Windows/System32/notepad.exe	vm-4

The UT SysDiff client action uses VMI to retrieve the in-memory executable sections of each guest’s running processes, computes the hash of each resident page, and compares the hashes. The

hyperagent then returns just the final result (resembling the SysDiff output above) to the GRR server.

Figure 5.7 contrasts the hyperagent’s approach with an analogous number of standard GRR agents performing the equivalent of SysDiff using a scripted combination of built-in client actions. This script requires three flows: the first collects the list of active processes; the second dumps the in-memory executables of these processes to the local file system; and, the last flow computes the hashes. This sequence illustrates normal GRR execution, and clearly results in additional round trips to the GRR server.

When compared to server-side analysis, the hyperagent’s SysDiff performance should be superior because repeated communication with the server is avoided, and because SysDiff can directly access each guest’s memory.

5.6 Evaluation and Results

Several experiments quantify the usability, resource usage, and performance of a GRR hyperagent. The effort required to port the agent into the hypervisor was also evaluated. The security benefits of an out-of-guest approach are adequately addressed by numerous prior VMI works [Bau15], and are not evaluated further here.

5.6.1 Evaluation Environment

The environment for the *Xen* portion of the performance evaluation consisted of a single Dell R920 hypervisor. The hypervisor ran Xen version 4.7.1 with a Fedora 25 dom0 and was furnished with four 2.3GHz Intel Xeon E7-4870 processors and 256GB RAM. The *Qubes* portion of the performance evaluation consisted of a single Dell Latitude 3340 hypervisor with two 1.6GHz Intel i5-4200U processors and 8GB RAM running Qubes 3.2. In both hypervisor environments, hyperagents ran in dom0 and communicated with the GRR server using a dom0 network interface. Each guest in the evaluation was a fully virtualized Fedora 25 guest with 1 vCPU, 1GB RAM, and 5GB of storage.

5.6.2 Development Work Factor

The effort required to port the GRR agent into a hyperagent was assessed first. For this purpose, the ported hyperagent’s codebase was compared to that of the original GRR C++ agent. Source code differences were then manually classified as belonging to the body of the agent, its actuators, the `libhyperagent` library, or features added in phase 3.

Table 5.3 shows the changes by category. In total, phases 1 and 2 modified 11% of the agent codebase (692 lines). The `libhyperagent` library itself accounted for another 857 lines. Note that the library can be shared by many other porting efforts. Finally, a combined 665 lines were added to

Table 5.3 Source Code Changes

<i>Component</i>	<i>Source Code Lines</i>			<i>SysDiff</i>
	<i>Body</i>	<i>Actuators</i>	<i>libhyperagent</i>	
GRR Server	109.3k	-	-	+148 (+0.1%)
C++ IG	4.3k	2.1k	-	-
Hyperagent	+182 (+2.8%)	+510 (+7.9%)	+857 (+13.5%)	+517 (+8.0%)

*Hyperagent code additions were made to the C++ In-Guest (IG) agent.
For reference, Python agent: 7.2k lines body, 5.1k lines actuators.*

Table 5.4 SysDiff Performance

	<i>t</i>	$\Delta\%$	<i>Per Agent</i>			
			<i>Msgs</i>	$\Delta\%$	<i>Net. payload</i>	$\Delta\%$
IG	22.2	-	$s \rightarrow$ 1185.6	-	$s \rightarrow$ 255.2 kB	-
			$s \leftarrow$ 2320.0	-	$s \leftarrow$ 322.0 kB	-
HA	4.5	20.3%	$s \rightarrow$ 1.0	0.08%	$s \rightarrow$ 0.1 kB	0.06%
			$s \leftarrow$ 1.0	0.04%	$s \leftarrow$ 1.3 kB	0.41%

*In Guest (IG) agent, Hyperagent (HA), GRR server “s.”
Network payloads exclude GRR keep-alives. Averaged across 10 trials.
Time in seconds. On 8 idle Linux guests with ≈ 35 processes each.*

the agent and server to support the SysDiff flow added in phase 3. A large majority of the GRR agent code was unchanged by the re-engineering process.

The re-engineering effort requires a developer who is able to identify actuators in the agent, and who should have a conceptual understanding of how VMI primitives work (e.g., read, write, and using kernel symbols). The availability of the `libhyperagent` library significantly reduces the effort required to use VMI.

5.6.3 Execution Time and Network Bandwidth

This experiment compared the performance of the hyperagent’s SysDiff flow with an equivalent GRR script that relies on in-guest agents. Figure 5.7 shows these two scenarios in detail. The same set of 8 Linux guests was used for both parts of this evaluation.

During this evaluation, GRR message types and sizes were recorded as they were sent and received by agents. The GRR messages and their size are used to quantify the difference in network traffic between the two approaches. The end-to-end duration (wall clock time) of the flow was also recorded.

Table 5.4 shows that locality and cross-VM visibility clearly benefit the hyperagent’s SysDiff flow, which returned results to the server 4.9x faster than the built-in flows. In addition, for the same end

Table 5.5 Agent Memory Usage

<i>Mean Cumulative Memory Usage (RSS)</i>					
	<i>In-Guest Agents</i>		<i>Hyperagent</i>		
	Python	C++	C++	MB/Guest	$\Delta\%$ IG C++
<i>Guests</i>	1	70.1	52.9	60.3	114.0%
	2	140.2	105.8	52.9	100.0%
	4	280.4	211.6	40.2	76.0%
	8	560.8	423.2	29.2	55.2%
	16	1,121.6	846.4	31.6	59.7%
	32	2,243.2	1,692.8	32.2	60.9%

*Memory units in MB. Statistics across 100 samples.
Shaded values are extrapolated.*

Table 5.6 In-guest Overheads

<i>Scenario</i>	<i>Sysbench Score</i>	$\Delta\%$
No agent	7246.0 $\pm$ 19.2	100.0%
“Hot” hyperagent	7223.0 $\pm$ 11.1	100.3%
In-guest agent	7200.9 $\pm$ 10.5	100.6%
“Cold” hyperagent	6071.2 $\pm$ 262.0	119.3%

*Units in events/sec at 0.95 Confidence Interval.
Higher is better. Statistics across 10 samples.*

result, the hyperagent generated roughly three orders of magnitude less traffic than the in-guest approach.

5.6.4 Performance and Overhead

In another set of experiments, the memory and processing impact of a hyperagent was assessed. For this purpose, the GRR *interrogate* flow was used. This flow collects basic host information when new agents register with the server. Due to its complexity, this flow is useful for comparing approaches. The interrogate flow exercises many of the code paths in the GRR agent, including most of the actuators. An interrogate flow consists of nine subordinate flows, which themselves trigger a total of 16 unique client actions on an agent, each retrieving information about a host’s network interfaces, disks, and users.

The first experiment in the set measured the duration of the interrogate flow for the Python GRR agent, the C++ GRR agent, and the hyperagent. The second experiment ran the interrogate flow across increasingly large sets of guests in order to quantify hyperagent memory consumption. Since agents running on separate guests are independent, memory is not shared and may be extrapolated for increasing numbers of in-guest agents. The hyperagent’s memory usage, however, was directly measured. The third experiment measured overheads by comparing the results of Sysbench’s⁵ CPU

⁵<https://github.com/akopytov/sysbench>

Table 5.7 Interrogation Latency. In Guest (IG) agent. HA (Hyperagent). Units in seconds. Statistics across 100 samples.

	<i>Xen</i>			<i>Qubes</i>		
	Py IG	C++ IG	Hyperagent	Py IG	C++ IG	Hyperagent
Time						
μ	10.41	9.84	10.14	11.56	14.83	9.58
σ	2.27	1.99	1.16	2.26	5.31	1.12

benchmark across three scenarios: the benchmark running on a guest without agents, on a guest with an in-guest agent, and on a guest with the hyperagent running underneath.

The latency results are shown in Table 5.7. For Xen, the hyperagent requires slightly more time to complete, which is expected due to the use of VMI. This is unlike Qubes, where the hyperagent performs much better than the in-guest agents due to the delays induced by the additional networking components in Qubes’ design (these only impact the in-guest agent’s traffic stream). Both results fall into ranges that would not likely be noticed by investigators.

The results of evaluating memory usage are shown in Table 5.5. For these conditions, there was a 40% improvement in hyperagent memory efficiency compared with 8 or more in-guest agents, illustrating the benefits of consolidating agents into a single hyperagent.

Lastly, Table 5.6 shows each agent’s in-guest performance overhead. The hyperagent measurements are broken into two cases, “hot” and “cold,” which indicate whether the guest’s disk had already been mounted by the hyperagent prior to the interrogate flow. The “hot” hyperagent impacts the guest less than the in-guest agent because the hyperagent’s flow processing is performed by the hypervisor, freeing up the guest vCPU to work on the benchmark. This experiment shows that once mounted, a hyperagent has little effect on in-guest overheads.

5.7 Discussion

5.7.1 Other Agents

The GRR agent was one of several agents studied as at part of this chapter. The methodology was also informed by incorporating `libhyperagent` into the ClamAV anti-virus scanner. A source code review of three other host agents (OSSEC, Nagios NRPE, and the Datadog agent) was also

performed. Although these agents differ in terms of purpose, design, and programming language, the methodology's core re-engineering ideas applies equally to them. Each agent has discrete, identifiable actuators, and each is also amenable to a consolidated, single-process hyperagent approach. See Appendix B for additional details.

5.7.2 Tradeoffs

A hyperagent gains its security and efficiency benefits as a tradeoff for increasing the complexity of the hypervisor. This tradeoff should be considered carefully. Good agent candidates include monitoring or security-related agents that gain the most from the increased privilege and concealment offered by a hyperagent.

5.7.3 Beyond the Datacenter

In addition to evaluating the GRR hyperagent on Xen, the evaluation included Qubes, a standalone desktop OS that encourages users to separate their computing tasks into workflows. These workflows are encapsulated inside disposable virtual machines. This concept of security by compartmentalization means that if one of the Qubes VMs becomes compromised, the damage is contained and unlikely to also compromise the underlying hypervisor and adjacent guests.

In-guest agents can be particularly cumbersome in Qubes. At the hypervisor level, Qubes VMs are designed such that the majority of their file system is backed by a shared copy-on-write root partition. Because this root partition typically houses the agent binaries and configuration file, multiple guests will run an identical agent configuration (a *software identity* [GR05] problem). While there are multiple ways to fix this problem, using a hyperagent may be the simplest.

5.7.4 Memory Deduplication

Modern hypervisors offer optional memory sharing features, where identical physical pages owned by adjacent guests are deduplicated and shared between guests. Memory sharing has been shown to offer significant savings [Gup08], so if this sharing is enabled, the potential for memory savings from agent consolidation may be lessened. However, while deduplication works well for certain classes of pages (e.g., read-only code pages), it induces computational overheads and has also been associated with security vulnerabilities [Ira14], leading some hypervisor vendors to disable it by default.

5.7.5 Adding VMI Functionality

`libhyperagent` contains the set of VMI-enabled functions that were required to port the GRR agent's actuators. Other agents would likely require different VMI functionality that does not yet exist

in the library, and adding this functionality would require VMI experience. Several works [DG13; DG11b] have demonstrated methods to automate VMI development, but manual development remains one of the most straightforward methods available [Bau15]. There are several disadvantages to manual development, including building VMI functions that only work against certain generations of guest kernels. As a part of this chapter, a repository of manually-derived VMI procedures has been made available to assist developers in extending `libhyperagent`.

5.7.6 Related Work

Hypervisor-based frameworks. Existing hypervisor-based tools such as *DRAKVUF* [Len14b], *NFM* [Sun14], and *Stackdb* [Joh14] allow plugins or customizations that could support agent functionality, but each would require an agent to be more thoroughly refactored against the framework's unique APIs. These frameworks were also not designed to host an entire agent, much less the agent consolidation that occurs during phase 2, and lack capabilities such as disk introspection. Forensics-specific hypervisors, such as *MiniSecHV* [Lut16] and *HyperSleuth* [Mar10], have similar issues.

The *XMHF* [Vas13] hypervisor introduced the concept of hypapps through an extensible interface that was used to port several other hypervisor-oriented security systems [Ses07; McC10] to run in its framework. While conceptually similar, this chapter and *XMHF* have divergent objectives and different intended use cases. First, *XMHF*'s execution model supports exactly one guest (such as a nested guest hypervisor). Furthermore, hypapps are integrated into *XMHF* at compile time and activated during system boot, meaning that making changes to a hypapp requires a system-wide disruption. From a development perspective, hypapps cannot take advantage of the standard libraries inherent in Xen's dom0, its ability to mount guest file systems, or its network stack for communication, each of which would require a far-reaching overhaul of an agent's codebase. Unlike the various sandboxing and containment options available in Xen, hypapps are an integral part of *XMHF* and operate at its highest privilege level. Finally, *XMHF*'s single-guest execution model precludes both agent consolidation and any analysis seeking to take advantage of cross-VM visibility.

Guest file system introspection. As previously mentioned in chapter 2.3, *Atomizer* [Jav] and *VMI-Honeymon* [Len12] both use `libguestfs` to access a guest's disk. Other out-of-guest disk access methods include intercepting and interpreting file systems and raw disk operations as was done by *vMON* [Li13], *DIONE* [MK12], open source tools such as *TSK*,⁶ and LibVMI's predecessor, *XenAccess* [Pay07]. Work which handles file systems directly must have a built-in understanding of each file system format likely to be encountered. Alternatively, `libhyperagent`, *KVMonitor* [KN14], and *V-Met* [MK17] use the `qemu-nbd` kernel module or `dm-thin` provisioning to mount the guest file system locally, transferring this burden to the dom0 OS and its file system drivers. The `qemu-nbd`

⁶<http://www.sleuthkit.org/>

approach also avoids the overheads of libguestfs's supermin appliance.

Regardless of the disk access approach, these other methods are also affected by the implications of delayed writes to disk. *DIONE* [MK12] specifically acknowledges delayed writes as a factor in their evaluation. Similarly, the work of *Krishnan et al.* [Kri10] introduced a heuristic, time-sensitive method to correlate system calls with disk activity, which might have been made more reliable if extraneous delayed writes were reduced.

5.8 Summary

This chapter introduced a methodology to port host agents to the hypervisor. These newly re-engineered hypervisor-based agents, called hyperagents, offer better isolation and concealment against in-guest threats. In return for a minimal impact on in-guest overhead, hyperagents are easier to administer across lots of guests, save memory and network bandwidth, and can meet and improve upon an in-guest agent's effectiveness. The effort to port an agent is acceptable, and is significantly aided by our structured methodology and accompanying `libhyperagent` library. This chapter shows that hyperagents represent a powerful and practical approach to defending virtualized systems. Most importantly, this methodology allows existing security systems to easily integrate hypervisor-based security functionality into their codebase.

CHAPTER

6

TENANT-DRIVEN CLOUD VMI

6.1 Introduction

This dissertation's final contribution centers on allowing tenants to employ their own HBS systems in the cloud. Through an open source cloud framework called *Furnace*, a tenant could leverage the same VMI capabilities available on a standalone hypervisor. which is essential to employing hypervisor-based security systems on public or private clouds.

Modern multi-tenant clouds offer their tenants accessible, affordable, and flexible computing resources. The industry is booming; some clouds are growing 40% year over year,¹ and public clouds such as Amazon Web Services (AWS), Microsoft Azure, and Google Compute Engine are relied upon to power customers' most critical systems. While clouds are increasingly essential, tenant security is a growing concern. After migrating to the cloud, tenants are unable to use Virtual Machine Introspection (VMI) for low-level security and system monitoring of their virtual machines. This lack of capability contrasts with developments outside of the cloud; since 2003, when *Livewire* [GR03] used a series of cleverly designed tools to infer guest activity, hypervisor-based techniques have gradually matured and are now adopted by major OS vendors,² antivirus companies,³ and hypervisors [Pay07].

Hypervisor-based tools are important; as attacks have become more sophisticated, defenders increasingly rely on these tools to provide powerful methods for malware analysis and foren-

¹2017 Roundup of Cloud Computing Forecasts, <https://goo.gl/pJ9uj2>

²Microsoft Virtualization-Based Security: <https://goo.gl/eixiqr>

³BitDefender Hypervisor Introspection: <https://goo.gl/MrZFQj>

sics [Len14b; DS13], kernel integrity enforcement [Yos17], surreptitious monitoring [Bus], and attack detection and response [Fra08]. Cloud providers do not currently provide an interface for their tenants (customers) to use this growing body of hypervisor-based tools [Bau15; Jai14; Sun15], and this absence of VMI in the cloud is a security gap.

The mechanism to provide tenants (or their managed security providers) with this access, a cloud VMI framework, allows the deployment of a tenant's VMI tools underneath their VMs. To be successful, a framework must satisfy two requirements. First, it must protect the cloud provider from potentially malicious tenant-supplied VMI tools. Second, it must enable these same tools to be full-featured, fast, and resource-efficient. Both requirements depend on how a framework provides access to the normally-inaccessible, privileged hypervisor APIs that VMI tools require.

There have been a variety of proposals for cloud VMI frameworks, including *CloudVMI's* [Bae14] network-based RPC, *LiveCloudInspector's* [ZR15] provider-curated VMI API, and *CloudPhylactor's* [Tau16] hypervisor access control scheme. No proposal has yet met the expectations of both cloud providers and tenants. CloudVMI is slow and insufficiently secure; LiveCloudInspector provides only limited tenant VMI capabilities; and, CloudPhylactor is too costly in resources.

This chapter introduces *Furnace*, an open source cloud VMI framework that leverages automated tool deployment, a software abstraction layer, and strong sandboxing to provide tenants the ability to run VMI apps directly on the cloud provider's hypervisors. A tenant writes their app using an extended version of the popular LibVMI API [Pay07], then submits the app through Furnace's cloud API. Furnace runs the app in a sandbox on the appropriate provider hypervisor.

Furnace is underpinned by its three-partition sandbox, which uses a set of well-accepted and well-tested security mechanisms to aggressively whitelist tenant app behavior. Tenant app code runs deprivileged in the sandbox's main partition and performs work by making IPC cross-calls to Furnace-provided resource brokers in adjacent partitions. Despite the overhead of sandboxing, an app achieves similar performance to a native VMI tool through an approach called *VMI call pipelining*; this pipelining reduces IPC-related overhead by exploiting the predictability of VMI calls.

Furnace is evaluated for its ease of use, security, and performance. Furnace's flexible API is used to port four VMI-related tools, including Rekall⁴ and a syscall tracer. Furnace's security claims hold against malicious tenant apps, sandbox enumeration, and fuzzing. Finally, benchmarking shows that a Furnace app approaches the speed of a native VMI tool and is up to 300x faster than CloudVMI.

This chapter makes the following contributions:

1. The design of an open source cloud VMI framework that meets the security, performance, and feature expectations of cloud providers and their tenants.
2. A practical, partitioned sandbox design that combines overlapping security mechanisms with a

⁴<http://www.rekall-forensic.com/>

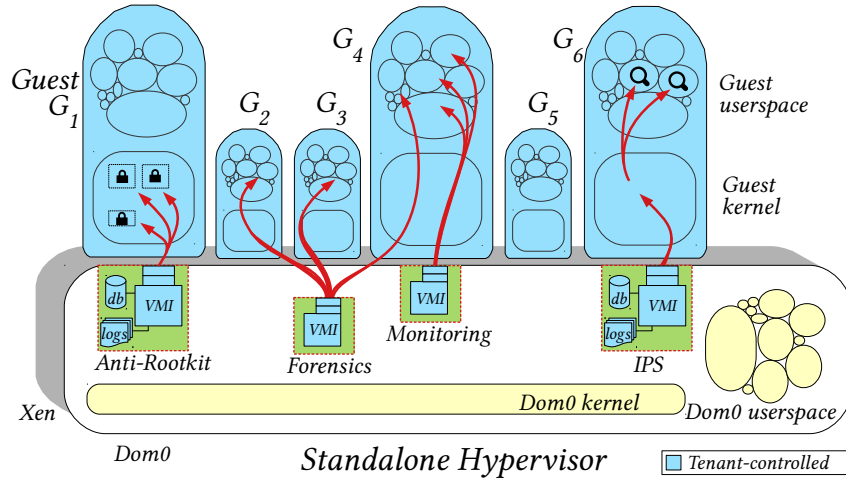


Figure 6.1 Under guest G_1 , an anti-rootkit VMI tool ensures kernel integrity. The forensics tool under G_2 , G_3 , and G_4 extracts process memory images. The monitoring tool under G_4 uses breakpoint injection to trace events occurring inside guest userspace processes. Finally, an intrusion prevention tool inspects new G_6 processes. The differing guest sizes in the diagram indicate the variety of instance types and operating systems found in modern clouds.

resource broker model.

3. A rich tenant API optimized for performance through VMI call pipelining.

6.2 Background

Virtual Machine Introspection (VMI) is a set of techniques to infer and influence a virtual machine's activity based on the interception, interpretation, and modification of a VM's raw bit patterns in its main memory, vCPU registers, and disk [GR03]. Some tools reason directly about a guest's raw memory contents, while others reconstruct a guest's high-level semantics [Bau15; Jai14; Sun15]. VMI tools can also subscribe to guest events that result in VM exits, such as guest vCPU register activity, software breakpoints, and extended page table violations. In general, VMI tools offer compelling advantages over building features into a guest operating system and installing agents in a guest's userspace:

- *Privilege*. An ability to freely read and modify guest memory and vCPUs.
- *Stealthiness*. The difficulty for a guest to detect underlying VMI tools.
- *Transience*. Tools can be quickly added and removed from under a guest.
- *Visibility*. All guest layers—drivers, kernel, process, data—are observable.
- *Impunity*. A general resistance to guest attempts at interference [Jai14; Bah10].

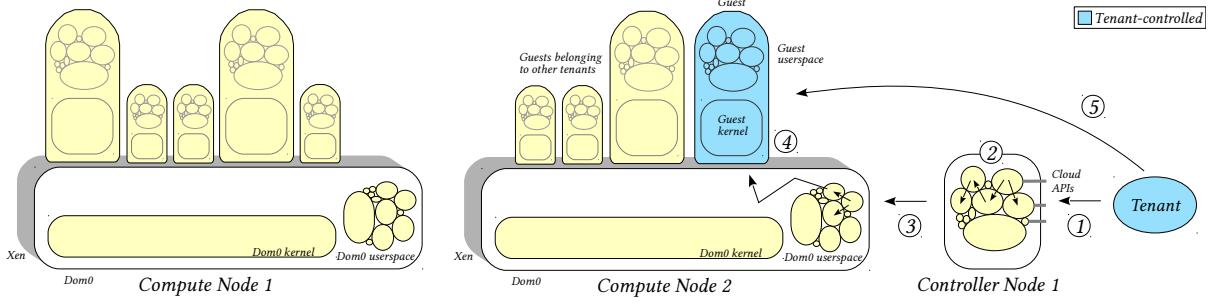


Figure 6.2 In modern clouds, tenants can use cloud APIs to request a new VM ①. This request is processed by cloud services running on a controller node. These services allocate resources, perform checks, and schedule changes ②. They schedule the new VM on a compute node, which locates or downloads the VM’s disk image, checks for and allocates resources, and prepares the VM’s network ③. Finally, the VM is booted ④ and the tenant accesses their new VM using a remote administration tool ⑤.

Most VMI tools are designed to run on a standalone hypervisor similar to the one shown in Figure 6.1, where they run as root and link directly into hypervisor VMI APIs. This chapter refers to these tools as *native VMI tools*. Specifically in the case of Xen, the hypervisor used in this work, native VMI tools typically run in Xen’s privileged management domain, dom0, where VMI APIs are exposed.

While VMI’s advantages often benefit security-related tools, VMI could support any task that involves monitoring a guest [Bus; Aru] or calling its functions [Car12].

This chapter introduces Furnace, a self-service framework that enables multiple cloud tenants to run VMI tools under their VMs without sacrificing either security or performance.

6.2.1 Modern Clouds

Major open source cloud ecosystems, including *OpenStack*,⁵ *Eucalyptus*,⁶ *Apache CloudStack*,⁷ and *OpenNebula*⁸ follow the basic cloud architecture pattern depicted in Figure 6.2. In this figure, a set of centralized *controller nodes* control a larger set of hypervisor-equipped *compute nodes*. The controller nodes interact with tenants and manage each compute node’s cloud resources (such as VMs). Internally, the controllers consist of loosely coupled cloud services, each responsible for managing aspects of the cloud such as scheduling VMs, IP space allocation, and controlling VM disk images.

Cloud providers rent compute resources to tenants according to the common infrastructure-, platform-, software-, and function-as-a-service abstraction layers [MG11]. Ultimately, each of these

⁵<https://www.openstack.org/>

⁶<https://github.com/eucalyptus/eucalyptus/wiki>

⁷<https://cloudstack.apache.org/>

⁸<https://opennebula.org/>

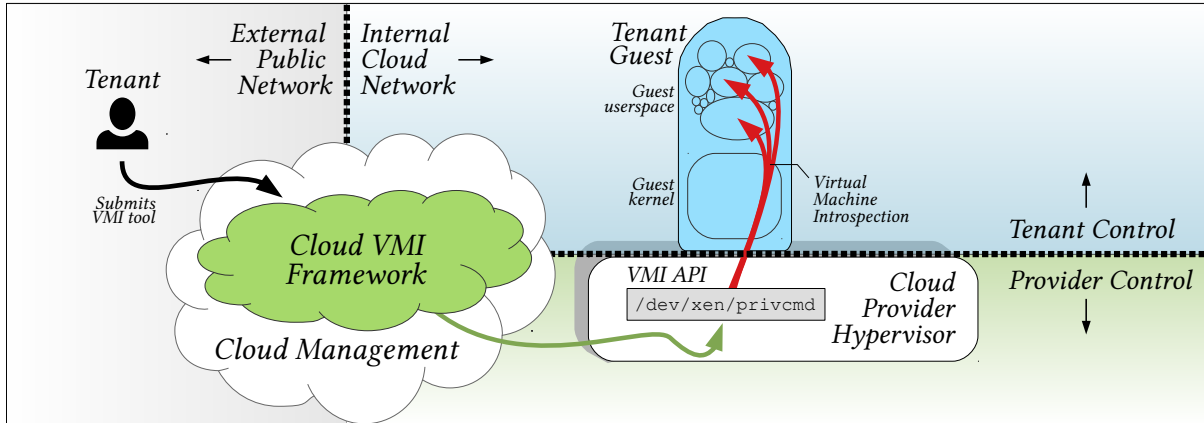


Figure 6.3 A cloud VMI framework’s objective is to provide safe, high-performance tenant access to a cloud provider hypervisor’s VMI API.

layers is underpinned by virtual machines supported by a compute node’s hypervisor.

6.3 Design Goals

A cloud VMI framework’s primary objective is to provide tenants with controlled access to the privileged hypervisor APIs necessary for VMI (see Figure 6.3). The framework must meet two expectations. First, if a cloud provider does not feel the framework provides this access safely, the framework will not be adopted. Second, the framework’s users (the tenants) must also feel it is fast, cheap, and useful enough to meet their goals.

Prior work. The idea of a cloud VMI framework is not new, yet existing proposals have flaws that can be categorized into three types:

- The RPC model.* One of the early frameworks was *CloudVMI (CV)* [Bae14], which exposed an RPC server on the hypervisor. CV was flexible and easy to use because tenant VMI tools simply had to be re-compiled against CV. At runtime, these tools could then connect to a hypervisor’s CloudVMI RPC server remotely across the network and have their VMI calls relayed to the hypervisor’s VMI APIs. Although not implemented by CV, this method could be made reasonably secure by RPC message gateways, authentication, and encryption. However, CV’s major flaw is speed. A RPC call is issued for each VMI call. Because VMI calls are analogous to individual memory accesses, this can result in hundreds of RPC network round trips per second.

- The preset model.* *LiveCloudInspector (LCI)* [ZR15] presents tenants with a preset list of VMI capabilities to choose from. Since the capabilities are developed and controlled by the cloud provider and activated via a simple, controlled interface, the risk that a malicious tenant could abuse them is reduced. LCI is likely faster than CV because (similar to a native VMI tool) its code runs directly

Table 6.1 Desirable Properties for a Cloud VMI Framework

Category		Furnace	CP [Tau16]	LCI [ZRI15]	CV [Bae14]	Native	
Safety	Secure	●	●	◐	●	○	Resistant to malicious tenant VMI tools.
	Abstract	●	○	●	○	○	Doesn't leak cloud platform details.
	Accountable	●	●	○	◐	○	Can control/measure VMI resource use.
Features	Fast	◐	●	●	○	●	Near-native VMI performance.
	Flexible	●	●	○	●	●	Allows tenant-supplied VMI tools.
	Cheap	●	○	●	○	●	Degree of resources req'd by VMI tool.

CloudPhylactor (CP). LiveCloudInspector (LCI). CloudVMI (CV).
 Partial fill indicates marginal or incomplete applicability.
 Filled circles are better.

on the hypervisor. However, LCI's problem is not with speed, but with choice. If a tenant's desired VMI action is not offered by the provider, they would have no reason to use the framework. This proposal also places the burden to create and maintain VMI capabilities on the cloud provider—a difficult proposition because typically VMI profiles must be maintained for every possible tenant kernel [Jai14].

—*The VM model.* Most recently, *CloudPhylactor* (CP) [Tau16] implemented per-VM security policies on the hypervisor that granted the ability for a tenant VM to perform VMI on a second, adjacent tenant VM. This method is fast because it uses the same APIs as native tools, and it is also flexible because a tenant is in full control of the VM performing VMI. CP's disadvantage is cost, both in terms of resources and effort. First, for each VM to be inspected, a tenant must provision (and pay for) a corresponding monitoring VM. Only a fraction of this VM's resources will end up directly supporting the VMI tool; the VM's kernel and userspace require resources as well. Second, the tenant is now responsible for managing, maintaining, and operating up to twice as many VMs along with the set of VMI tools in each.

Requirements. Table 6.1 shows the desirable properties for a VMI framework⁹ and compares Furnace with prior work. For illustration, the table also includes the native VMI tool model where tenants use privileged accounts on the hypervisor—this model lacks any protection from malicious users. Each property is briefly discussed below.

Safety. To protect the cloud, a *secure* cloud VMI framework should contain or filter tenant VMI code to ensure it cannot execute in an unintended or uncontrolled manner. A framework should also uphold the cloud's *abstraction* model so a tenant VMI tool is prevented from inferring private cloud implementation details, i.e., a tenant should not be able to abuse the framework to determine which hypervisor is hosting their VM or what other VMs are co-resident. Finally, a framework should *account* for, measure, and control a tenant VMI tool's resource usage. This is necessary for billing and to prevent the possibility of (intentional or accidental) resource exhaustion.

⁹This list is an expansion of the five framework problems described by *CloudPhylactor*.

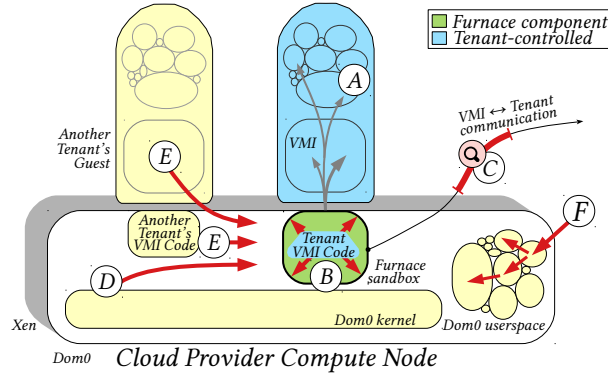


Figure 6.4 Threat Model.

Features. A tenant’s VMI tool should be suitably useful and powerful. A framework should strive to run tools as *fast* as a native VMI tool. Designs that impose heavy runtime overheads are undesirable. In order to be useful, a framework should be *flexible* and expressive, allowing tenants to easily design their own custom tools. Finally, a tenant VMI tool should be *cheap*, both easy to build and inexpensive to run. This means a cloud framework should automate the deployment, execution, and management of a tenant’s VMI tools, and, once running, the resources required to support the tool should be minimized.

Assumptions and Threat Model. Figure 6.4 shows a common cloud threat model [SL16] where the cloud management plane, hypervisor, and physical hardware are assumed to be secure. Additionally, because Furnace’s security model relies on a sandbox to safely contain tenant app code, it is assumed that the underlying mechanisms used to implement this sandbox are correct and secure. Through syscall filtering, only a small set of syscalls are permitted to be used inside the sandbox; these syscalls are also assumed to be secure. Given these assumptions, Furnace is designed to be resilient against the following attack vectors:

Ⓐ *Malicious VMI Input.* A malicious tenant or third-party malicious code seizes control of the VMI tool. To do so, the tenant carefully crafts malicious input—the guest’s main memory—in order to exploit a vulnerability in the tool [Tau16]. Ⓑ *Attacks against app APIs.* A malicious tenant exploits flaws in exposed VMI APIs, allowing the leaking of private cloud implementation details or permitting a sandbox escape that yields access to the hypervisor. Ⓒ *App network snooping:* A third-party attacker tampers with messages sent between a tenant’s VMI tool and external services, a concern for frameworks that provide support for this communication.

Several threats are out of scope. Furnace does not protect against a malicious Ⓓ or “curious” cloud administrator. Micro-architectural attacks Ⓔ and related side channels are considered out of scope, as are attacks targeting the cloud platform (e.g., OpenStack) itself Ⓕ. Finally, while the strong semantic gap [Jai14] remains a concern for VMI tools, it is not in scope for Furnace.

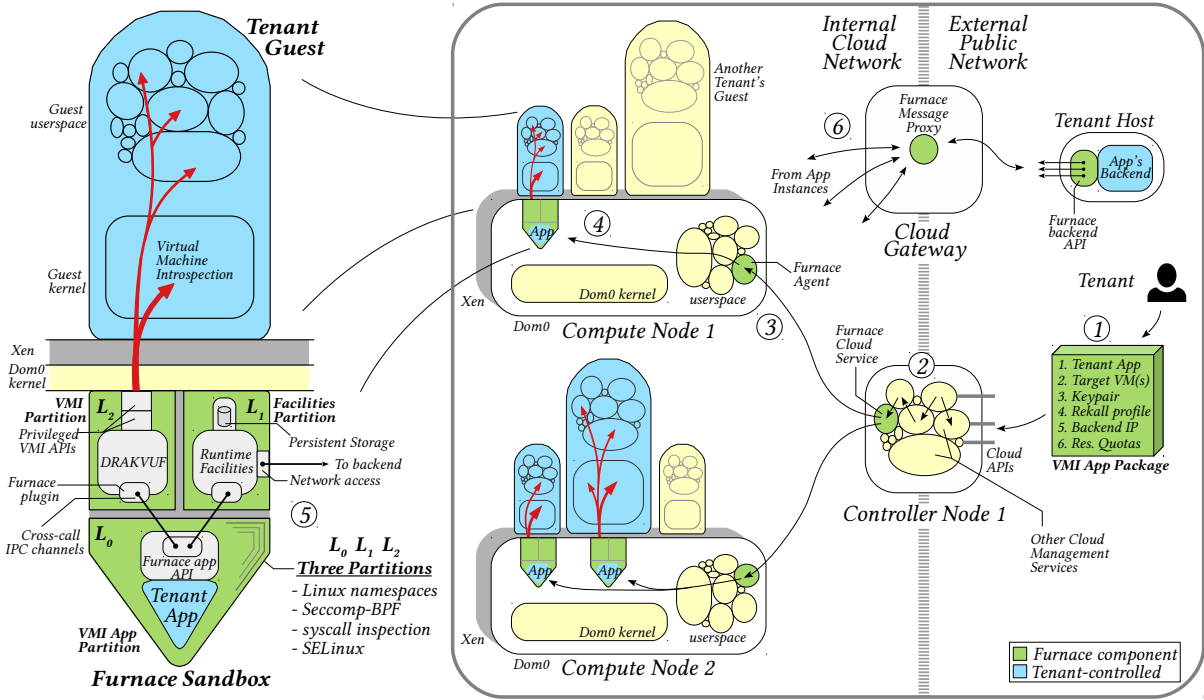


Figure 6.5 The Furnace architecture from the cloud provider's perspective.

6.4 Design

This section introduces Furnace as a safe, fast, and useful cloud VMI framework. Furnace is first described from the perspective of a cloud provider and later from that of a cloud tenant.

Cloud provider perspective. Figure 6.5 depicts Furnace's architecture as part of a typical cloud infrastructure. ① A tenant submits a VMI app package to the provider's public Furnace cloud service. The package contains the tenant app's source code and metadata such as the tenant's target VM and the app's allowed resource quotas. ② The Furnace cloud service parses the package contents, locates the tenant's target VM, and signals ③ the appropriate Furnace agent, which ④ provisions a Furnace sandbox under the VM and begins executing the tenant's app inside it. ⑤ The app runs as an unprivileged process inside the sandbox. ⑥ Using tenant-provided keys, Furnace provides a mechanism for an app to securely communicate with a single tenant-controlled backend outside of the cloud. At the cloud boundary, this communication is relayed onto the public network via the Furnace message proxy.

Cloud tenant perspective. Figures 6.6 and 6.7 shows Furnace from the tenant's perspective. A tenant first begins by writing their VMI app. Furnace's tenant programming model means the tenant could write up to two programs: the *app* and an optional *backend*. A typical tenant use case will likely

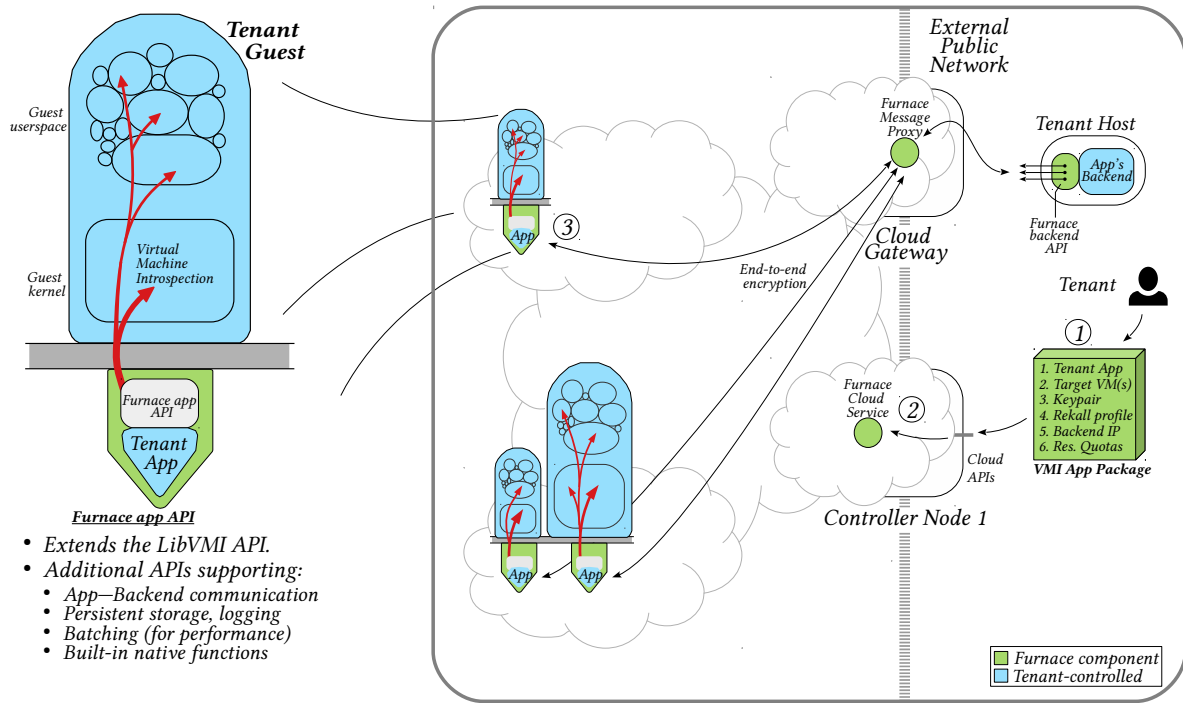


Figure 6.6 The Furnace architecture from the cloud tenant's perspective.

consist of many copies of the same VMI app each running under a different tenant VM and all communicating with a single backend running externally under the tenant's control. When ready to use, the app is packaged with metadata and submitted to Furnace, which starts copies of the app beneath each desired tenant VM. Once started, app instances optionally establish a two-way connection with the backend, which acts as a coordination activity between app instances and the rest of the tenant's security enterprise (e.g., logging, threat feeds, databases, external blacklists, etc.). While apps are tightly constrained inside their Furnace sandbox, the tenant can run the backend anywhere. Note the Furnace-enforced abstraction these figures; as far as the tenant is concerned, both VMI and app $\leftrightarrow$ backend messaging are performed by simple function calls. Furnace is responsible for handling the (sensitive) details related to function call execution. The fact the app runs in a sandbox can be ignored by the tenant. Additionally, because there is less infrastructure for the tenant to directly operate, maintain, and secure, tenants may find these abstractions safer and more scalable than prior work.

Sandbox overview. Figure 6.5 also shows the Furnace sandbox. In the sandbox's L_0 partition, the tenant's app indirectly performs privileged tasks through *IPC cross-calls* to provider-controlled resource brokers in adjacent sandbox partitions L_1 and L_2 . Many different apps can run under the same guest simultaneously; each app has its own L_0 and L_1 partitions while the L_2 partition is shared.

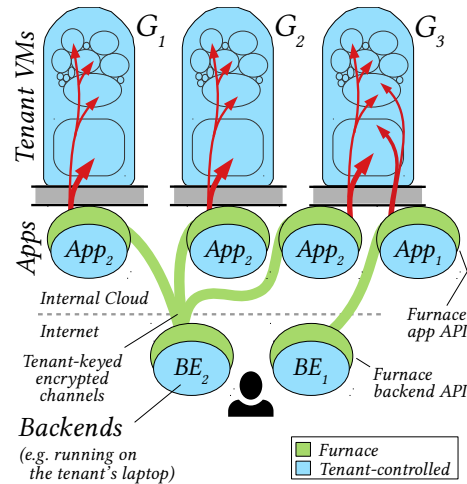


Figure 6.7 Tenant model.

Furnace VMI app API. Tenants already familiar with VMI can quickly and easily write Furnace VMI apps. The Furnace API inside L_0 provides the core component of the LibVMI API (56 functions) and 12 Furnace-specific functions. As the tenant app executes, the VMI calls it makes to the Furnace API are translated into synchronous cross-calls to the L_2 partition where the VMI is actually performed. In the opposite direction, events produced by the guest itself are sent from L_2 to the app in L_0 either synchronously (the guest is paused until the app responds) or asynchronously (the guest continues running), with delivery occurring via a callback that the app registers at startup. The third partition, L_1 , is used for communication with the app's backend and to provide persistent storage for the app.

Furnace's VMI component in L_2 is powered by *DRAKVUF* [Len14b], a modular hypervisor-based dynamic malware analysis system, and tenant apps gain all the benefits of *DRAKVUF*—stealthiness, multiplexed events, and its built-in features—without having to build and configure it themselves. Furnace integrates into upstream *DRAKVUF* via its plugin system.

Figure 6.8 contains example source code that demonstrates the basic operation of a process whitelisting app. The example app first ① registers for guest CR3 register events (process context switches). For each received event, the app ② inspects the guest process that caused it. If the process is new, the app consults ③ the backend's whitelisting rules, and terminates the process if it is not allowed.

Furnace's initial implementation fully supports VMI apps written in Python. However, due to its language-agnostic cross-call interface, Furnace can potentially support tenant apps written in any language that has ZeroMQ¹⁰ and Protocol Buffers¹¹ libraries.¹²

¹⁰<https://zeromq.org/>

¹¹<https://developers.google.com/protocol-buffers/>

¹²Similarly, while Furnace was developed on Xen, it can also support KVM when equipped with a KVM-compatible L_2 component.

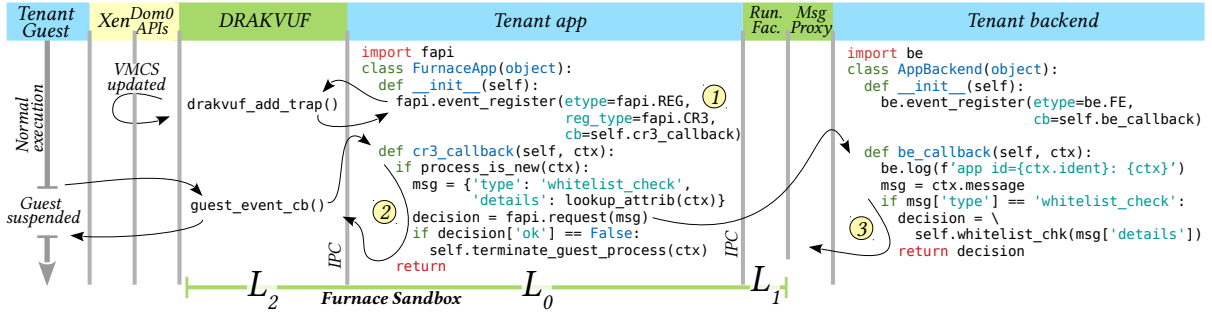


Figure 6.8 Furnace tenant programming model. This example tenant app and backend work together to intercept and whitelist new guest processes.

Cloud integration. Figure 6.5 also shows other Furnace components elsewhere in the cloud, including the public Furnace cloud service that tenants use to create and tear down VMI apps; the per-hypervisor Furnace agent that controls app sandboxes; and the Furnace message proxy that relays app $\leftrightarrow$ backend messages between the public Internet and the internal cloud network used by each sandbox's L_1 partition. The Furnace cloud service must integrate with the cloud platform to authenticate tenants, get VM state, and query for a VM's location. Furnace has the potential to be paired with many popular cloud platforms, and this integration is needed for a complete system.

Usage model. Furnace's intended users are cloud tenants that write and deploy their own apps directly. However, Furnace can be used in other scenarios, such as tenants choosing to delegate their Furnace privileges to a third party (such as a managed security provider), or cloud providers that wish to use Furnace exclusively (such as private clouds). Furnace also would work well in use cases where there is no cloud management layer, such as distributed VMI-based malware analysis across a set of hypervisors.

Design goal analysis. Furnace's sandbox-based approach represents a middle ground that mitigates the security risks of running directly on the hypervisor while avoiding the resource overheads associated with a dedicated VM. Furnace's VMI app model allows the tenant-provided app to run on the hypervisor at near-native speeds and communicate with its backend without knowledge of internal cloud networking details.

6.4.1 Security

Because Furnace runs the tenant's app on the hypervisor, the ability for Furnace to suppress potentially malicious tenant apps is its most important requirement. Furnace's approach is to place each tenant app in a sandbox that uses existing, well-tested security mechanisms to whitelist the following allowed app activities:

1. *VMI Operations:* An app can read and write to its assigned VM's memory and vCPU registers,

Table 6.2 Sandbox Comparison

	Userspace Isolation	Resource Quotas	Res. Broker Model	“Chroot”-like	Syscall Filtering	Contextual Security	Deref. Userspace FS Size
<i>Furnace</i>	✓	✓	✓	✓	✓	✓	≤8.9k
<i>Chrome</i>	✓	✗	✓	✗	✓	✗	N/A
<i>Lambda</i>	✓	✓	✗	✓	✓	✗	38.3k ¹³
<i>Mbox</i>	✗	✗	✗	✗	✓	✓	N/A

Supported: yes(✓) no(✗). FS size in # of files.

capture events from the VM, and pause/resume the VM.

2. *Communication*: An app can send arbitrary messages across the network to its associated tenant backend.
3. *General Computation*: An app can make limited use of hypervisor CPU and memory as a part of normal operation, including a limited ability to use the hypervisor’s local disk for storage.

An app must be strictly constrained to these tasks. The Furnace sandbox must ensure that a tenant app cannot learn anything about the cloud that cannot already be discovered by a normal tenant VM. This includes preventing the app from inferring sensitive information about its hypervisor, such as its file systems, network interfaces, running processes, and co-resident VMs.

Sandbox design. The Furnace sandbox uses a set of mutually reinforcing OS-based security mechanisms that make it unlikely that a single flaw in policy or implementation would allow untrusted code to escape. The sandbox incorporates well-tested mechanisms already successfully leveraged by the Chrome, AWS Lambda, and *Mbox* [KZ13] sandboxes, but is notable and distinct for two reasons. First, to a much more rigorous extent, the Furnace sandbox breaks apart the functionality of the untrusted tenant app so that a granular, tight-fitting security policy can be applied to each component. The most pronounced example of this segmentation is the sandbox’s use of two resource brokers in L_1 and L_2 , each with differing privileges. The Furnace sandbox also aggressively and contextually whitelists the behavior of each component. A profile of whitelisted behavior is generated beforehand by the cloud provider and any deviation results in Furnace terminating the app for violating policy. Table 6.2 shows a feature comparison with related sandboxes:

- *Independent namespaces*. Similar to container-based sandboxes such as AWS Lambda,¹⁴ each Furnace sandbox partition features an independent file system and employs userspace isolation.
- *Resource broker model*. Untrusted tenant code is limited to one primary external action—making

¹³<https://alestic.com/2014/11/aws-lambda-environment/>

¹⁴Lambda represents container-based sandboxes in part because it is routinely audited.

Table 6.3 Furnace Partitions

	Furnace Partition		Network access		Runs As	# Init. Syscalls	# Runtime Syscalls	VMI / dev access	FS Size
App L_0	No	U	58	31	No	8.9k			
Network L_1	Yes	U	42	28	No	8.9k			
VMI L_2	No	R	63	24	Yes	5.7k			
Unprivileged user (U). Root (R).									

cross-calls. Cross-calls are IPC messages to an external resource broker, which validates and acts upon the untrusted code's requests.

- *Syscall filtering.* Furnace employs granular syscall filtering similar to Mbox, and the full range of syscall arguments can be considered by a security policy.
- *Dynamic policy tightening.* Similar to *Pledge*,¹⁵ Furnace can swap syscall filtering security policies based on context and does so when transitioning to a more restrictive policy immediately prior to tenant app execution.

Sandbox implementation. Furnace's sandbox implementation achieves a base level of isolation and access control via Linux namespaces and SELinux. Seccomp-BPF is used to reduce the hypervisor kernel's exposure to an app's syscalls, and the syscalls that are allowed are inspected by a contextual syscall inspector that leverages the Linux kernel's `ptrace` interface. Finally, resource usage quotas (e.g., CPU and memory) are enforced by cgroups. Each of these sandbox security mechanisms protect against a subset of attacks and is essential to the overall security of the sandbox. When overlap between mechanisms does occur, it is because one mechanism nullifies a weakness in another.

Table 6.3 shows the details of each Furnace partition's unique security configuration, including the number of syscalls allowed at initialization and runtime (creating these security policies is discussed later in this section). Figure 6.9 further depicts sandbox initialization and how its components interact at runtime.

Linux namespaces. Namespaces provide kernel-enforced userspace isolation ideal for limiting an app's ability to observe the rest of the hypervisor. Linux's support of namespaces includes user IDs, cgroups, mount points, IPCs, and network interfaces. When Furnace uses these namespaces in combination, a VMI app lacks any practical visibility outside of its assigned namespace. Mount namespaces are used to “chroot” each sandbox partition into a minimal SELinux-labeled file system tailored for its use. Process and user ID namespaces further isolate code running in each partition.

¹⁵<https://man.openbsd.org/pledge.2>

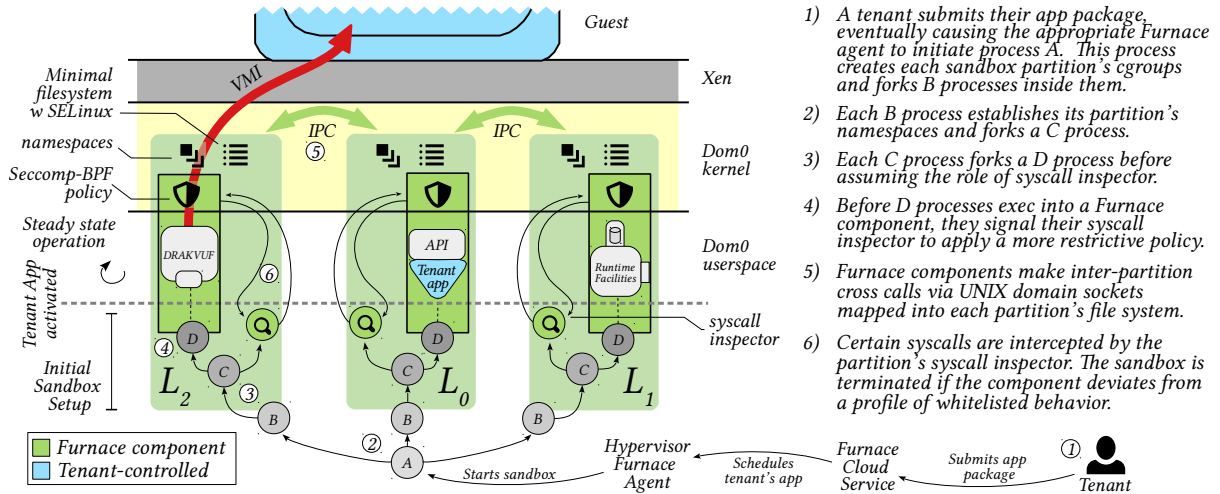


Figure 6.9 From bottom to top: the sandbox is assembled when a new app is started.

	Allowed by Seccomp-BPF	Syscall Inspector allows during runtime but checks arguments		Syscall Inspector allows only during initialization		
1	epoll_ctl	brk	madvise	access	getrandom	sched_getscheduler
2	epoll_wait	connect	mmap	arch_prctl	kill	sched_setscheduler
3	getpid	exit	mprotect	clone	lstat	set_robust_list
4	poll	exit_group	munmap	close	mkdir	set_tid_address
5	recvfrom	fcntl	open	dup	pipe2	sigreturn
6	sendto	fstat	rt_sigaction	dup2	prlimit64	socketpair
7	write	futex	sigaltstack	epoll_create	read	sysinfo
8		getpeername	socket	execve	readlink	uname
9		getsockopt	stat	getcwd	rt_sigprocmask	unlink
10		ioctl		getdents	sched_getparam	vfork
11		lseek				wait4

Figure 6.10 The syscall policy for the VMI app running in the sandbox's L_2 partition. The remaining ≈ 245 syscalls are denied.

The L_1 partition is the only partition given network access. To control resource quotas, Furnace places each sandbox partition in its own cgroup. Cgroups are used to limit the number of tasks allowed to exist in a partition, and how much CPU and memory these tasks can use. Minimum resource limits are statically determined based on profiling, and maximum limits are set by (and billed to) the tenant. Furnace also uses cgroups to limit apps to a single thread in order to prevent a known ptrace race condition involving multi-threaded code [Swi17].¹⁶

SELinux. Furnace uses a SELinux policy in each sandbox partition to provide a consistent access control policy. Each partition's file system is labeled with up to four Furnace types: one for each partition's processes, and one for files.

Seccomp-BPF. Furnace uses Seccomp-BPF to filter the syscalls made by a partition. If a process

¹⁶While this precludes multithreaded apps, §6.6.2 shows performance is acceptable. Heavy processing is better accomplished by the app's backend, which lacks this restriction.

attempts to make a prohibited syscall, it is terminated. Each Furnace partition has its own Seccomp-BPF policy. Of the hundreds of syscalls available in Linux, Table 6.2 and Figure 6.10 show that Furnace whitelists ≈ 60 syscalls in a partition's most permissive state: as it is initialized. Of these, ten are related to cross-calls or basic process functionality and allowed unconditionally, while the remainder are subject to inspection by a ptrace-equipped security monitor called the *syscall inspector*.

Syscall inspection. The syscall inspector provides two security functions: dereferencing userspace pointers passed as syscall arguments and dynamically tightening its current security policy based on context. One complication with Seccomp-BPF is that certain syscalls such as `open` cannot be fully filtered because Seccomp-BPF programs running in the Linux kernel cannot dereference userspace pointers [Swi17; Gar03]. To resolve this, Seccomp-BPF can partner with a ptrace-equipped userspace process that can. Furnace's three syscall inspectors monitor the Furnace component in each partition and make filtering decisions based on its syscall behavior. For example, if a tenant's app attempts to open a file not on the syscall inspector's whitelist, the app will be terminated. The syscall inspector applies its policies contextually. Based on the observation that many unique syscalls only occur while the partition and its processes are starting up, the syscall inspector initially uses a permissive security policy. Once initialized, the partition's policy is tightened.

Pertinent to Furnace's threat model, each sandbox partition has its own syscall inspector, so abnormal behavior in the L_1 and L_2 partitions will also result in app termination.

Security policy generation. The cloud provider supplies Furnace with three security policies: 1) a SELinux type enforcement policy; 2) a set of combined Seccomp-BPF and contextual security policies used by the three syscall inspectors; and 3) namespace-related parameters such as resource quotas. Only the syscall inspector policies are expected to be periodically adjusted by the cloud provider, and only on the occasion that minor adjustments need to be made. These policies are meant to apply to all Furnace sandboxes (as opposed to custom per-tenant policies).

The syscall inspector policies are created by recording all system calls (via the `strace` utility) made by each partition at runtime under a variety of benign conditions—an approach similar to *SysTrace* [Pro03]. These system calls and their arguments are then analyzed and manually inspected. This manual inspection requires detailed familiarity with system calls and benefits from existing policy references (most notably Chromium's¹⁷). Only the behaviors observed during this profiling step become part of the partition's allowed syscall profile. Figure 6.10 shows the syscall policy applied to the tenant's VMI app running in the L_0 partition. The only syscalls a tenant app can make are either unconditionally allowed by Seccomp-BPF (column 1) or inspected by the syscall inspector (column 2).

Policies generated through profiling have been shown to be straightforward to generate and are a good starting point for refinement¹⁸ [SY17]. One disadvantage of profiling is the likelihood

¹⁷<https://chromium.googlesource.com/chromium/src.git/+master/sandbox/linux/>

¹⁸<https://github.com/Netflix/repokid>

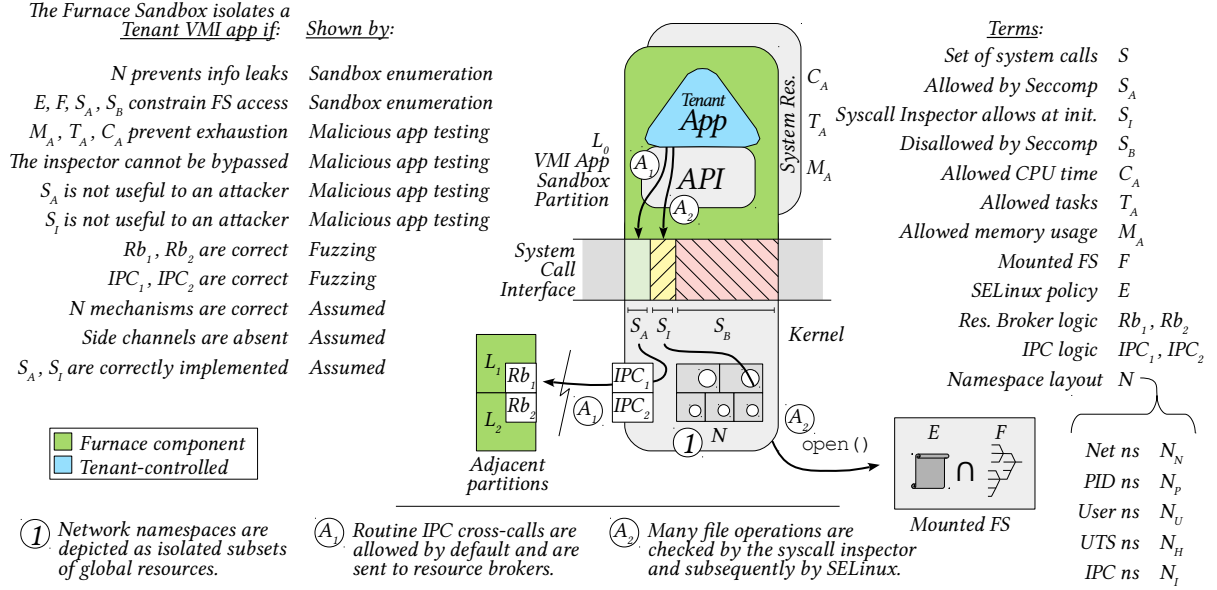


Figure 6.11 This sandbox security analysis depicts the order in which security mechanisms are encountered by code running in a sandbox partition and asserts the sandbox is reasonably secure if certain criteria are met. Furnace’s security evaluation will test elements of this assertion (the “shown by” column) to provide confidence that Furnace can successfully contain malicious VMI apps. *Notes: for brevity, only the L_0 partition is shown, but L_1 and L_2 are conceptually identical; the syscall inspector is not shown.*

that benign behaviors are missed and later cause unpredictable false positives at runtime. However, because the set of syscalls required by each partition is small, the profiles are quite robust, and these types of false positive were encountered rarely during testing. While a cloud provider can and should generate their own profiles (and analyze them for weaknesses [Jae03]), Furnace comes included with a default policy set.

Interlocking mechanisms. Some security mechanisms appear redundant. One clear example is the syscall inspector, which decides whether a file can be opened or not—something SELinux also does. The overall design could potentially be simplified if this redundancy was removed. However, each mechanism contributes to the overall security of the sandbox. The syscall inspector can choose its security policy dynamically at runtime, but lacks the more expressive and thorough approach of SELinux, which Furnace uses for broad, basic policies. These interlocking mechanisms help ensure that unknown weaknesses in security policies, implementation bugs, and profiling errors do not result in vulnerabilities. It is practical to permit some overlap in case one mechanism fails, and this point is discussed further in Furnace’s security evaluation.

Attacking the sandbox. Although restricted, an app is still allowed to perform cross-calls and make certain system calls. Furnace takes additional steps to minimize the chance these two interfaces can be successfully attacked. Figure 6.11 shows a security analysis that summarizes the sandbox’s

assumptions, security model, and how it will be evaluated.

Mitigating exposed kernel APIs. Flaws in existing syscalls have defeated Seccomp-BPF-based sandboxes in the past—most recently in Chrome.¹⁹ In this case, a missing access control check in the Linux kernel’s implementation of the `waitid()` syscall—a syscall purposely allowed by the Chrome syscall policy—led to a sandbox escape. This vulnerability demonstrates that, even in a small set of whitelisted system calls, implementation bugs can still occur and be exploited. While attacks like this remain possible, Furnace’s syscall filter is more restrictive than Chrome’s, reducing the number of potentially vulnerable syscalls.

Mitigating exposed IPC logic. Flaws in IPC logic are another source of sandbox escapes. These flaws are usually discovered through fuzzing [LL12; GS14]. Compared to the Chrome sandbox,²⁰ Furnace’s IPC interface is simpler, so Furnace’s IPC mechanism is more difficult to attack. Also, unlike Chrome, Furnace’s IPC does not use shared memory, making it immune to a class of IPC-related use-after-free vulnerabilities.²¹ Regardless, fuzzing remains an important part of Furnace’s security evaluation.

Dom0 complexity. Adding a new system such as Furnace to a cloud’s hypervisors inherently increases the cloud’s attack surface. However, not counting the Furnace sandbox’s external dependencies (DRAKVUF, LibVMI, protocol buffers, and ZeroMQ), Furnace’s combined 6k lines of sandbox-related code (2k Python Furnace API in L_0 , 1k Python facilities daemon in L_1 , and 3k C/C++ for Furnace’s DRAKVUF plugin in L_2) represent a small increase in dom0’s attack surface, especially when compared to the 183k lines of C in common Xen dom0 libraries.²²

6.4.2 Performance

While Furnace’s inter-partition IPC channels allow the tenant app to be deprivileged, IPC overhead adds latency to every cross-call. This message-passing has implications on both guest and app performance.

Batching and pipelining. By default, a single Furnace app VMI call is serialized into a single cross-call message that is sent between the L_0 and L_2 partitions. To minimize IPC overhead, Furnace includes a batching API that bundles multiple independent VMI calls into a single cross-call message. This amortizes IPC overhead across the entire batch, which would argue for bundling as many VMI calls as possible into a single message.

While this simple batching technique is beneficial, several commonly encountered guest kernel data structures are built in ways that force a VMI app’s batch sizes to remain small. One such example is walking a linked list in guest kernel memory, where each link in the list contains a pointer to the

¹⁹<https://salls.github.io/Linux-Kernel-CVE-2017-5123/>

²⁰<http://blog.azimuthsecurity.com/2010/08/chrome-sandbox-part-2-of-3-ipc.html>

²¹<https://nvd.nist.gov/vuln/detail/CVE-2017-14904>

²²<https://github.com/xen-project/xen/tree/master/tools>

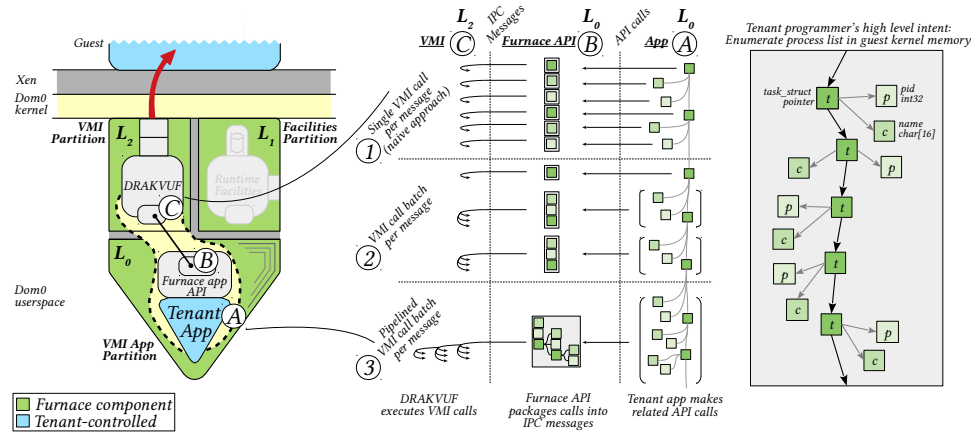


Figure 6.12 In addition to sending single VMI calls ① to L_2 , a tenant VMI app can be written to batch VMI calls ② to improve performance. Batched calls in the same message can also be *pipelined* ③ where an earlier call is fed into subsequent calls. Pipelining can provide significant speedups for long chains of predictable and interrelated VMI calls, such as the enumeration of a guest’s list of processes.

next link. With simple batching, the entire list cannot be walked in a single cross-call because the pointers in each of the links are not known in advance.

With this problem in mind, the Furnace API further supports *VMI call pipelining*. In this model, while the L_2 partition processes a batch of VMI calls in the same cross-call message, L_2 can be programmed to make available the results of an earlier VMI call as the input for subsequent calls. This enables a single pipelined cross-call to walk the entire length of a linked list in guest kernel memory at once. All three API modes—default behavior (no batching), batching, and pipelining—are shown in Figure 6.12.

If the length of the linked list is known, the message can include the exact number of calls needed to walk the list completely. If the size is not known, it can be estimated, and since the end of a linked list is trivial to detect, extraneous results past this mark can simply be discarded.

The tenant app programmer must choose to use pipelining. To do so, the programmer uses the Furnace API to generate a batch object, to which they then add and link VMI calls. An example of this API is shown in Figure 6.13. When this batch object is ready, the Furnace API sends a single multi-part cross-call message to L_2 , which extracts each VMI call from the message, resolves its dependencies, executes it, and stores the results in a multi-part reply.

Built-in native procedures. Furnace’s API can also be used to invoke a small set of built-in native VMI procedures present in DRAKVUF. While Furnace and DRAKVUF could support an arbitrary number of built-in procedures, adding a procedure requires DRAKVUF to be recompiled and therefore would not normally be available to a tenant.

```

1 c = list_head      # Head of guest kernel's process list
2 p = pid_offset     # Offset to pid in task_struct
3 t = tasks_offset   # Offset to tasks in task_struct
4 n = name_offset    # Offset to comm in task_struct
5 o = fapi.batch_new() # New batch object
6
7 o.add('00', fapi.PAUSE_VM)
8 o.add('01', fapi.READ_STR_VA, {'vaddr': c+n-t, 'pid': 0})
9 o.add('02', fapi.READ_32_VA, {'vaddr': c+p-t, 'pid': 0})
10 o.add('03', fapi.READ_ADDR_VA, {'vaddr': c, 'pid': 0})
11 # The results of the preceding calls are pipelined into the next set
12 for i in range(1, 120): # Batch another 119 sets
13     o.add(f'{i}1', fapi.READ_STR_VA, {'vaddr': n-t, 'pid': 0}, f'{i-1}3', 'result', 'vaddr', fapi.ADD)
14     o.add(f'{i}2', fapi.READ_32_VA, {'vaddr': p-t, 'pid': 0}, f'{i-1}3', 'result', 'vaddr', fapi.ADD)
15     o.add(f'{i}3', fapi.READ_ADDR_VA, {'vaddr': 0, 'pid': 0}, f'{i-1}3', 'result', 'vaddr', fapi.ADD)
16 o.add('99', fapi.RESUME_VM)
17
18 fapi.batch_send(o) # Synchronous cross-call: VMI call results stored in o
19
20 for r in o.results(): # Iterate through each call's results in order
21     if r['result'] = list_head: # The entire linked list was traversed
22         break
23     print(r) # A dict of the call's results, e.g., {'name': '171', 'result': 'chrome', 'status': 0}

```

Annotations:

- The identifier for the VMI call.
- The LibVMI call function to execute.
- An argument dictionary for the VMI call.
- The identifier for an earlier VMI call, its results will be used as the input for this call.
- From it, take this field and use it as input here.
- Add to, subtract from, or overwrite the existing value.

Figure 6.13 A tenant app source code snippet that demonstrates the use of Furnace’s VMI call pipelining API. Here, the tenant programmer creates a batch object *o*, populates it with several hundred VMI calls, then serializes and sends it to *L*₂ to be processed by calling `batch_send()`. By pipelining related calls together in the same cross-call message, IPC-related overhead can be greatly reduced.

6.5 Case Studies

This section describes four fully-functional examples of Furnace VMI apps. All four examples are useful for memory analysis and fine-grained system monitoring—two likely use cases for Furnace. Most are written in less than 100 lines of Python code. This small size is enabled by:

- *Inherent features.* Furnace handles many low-level VMI details automatically. Through DRAKVUF, apps inherit features enhancing their stealthiness, the ability to multiplex guest events across multiple apps (such as when two apps consume events from the same guest), and built-in VMI procedures.
- *Automated deployment.* App deployment is handled by Furnace; the tenant has no concern for the process required to build, deploy, and run their app.
- *Familiar API.* Furnace’s Python API function signatures are identical to LibVMI’s Python library. Existing LibVMI tools written in Python can be placed in a Furnace app and run with minor modification.

App: Memory analysis. *Rekall* is a popular memory analysis framework. It is typically invoked from the command line to analyze file-based memory dumps. To use Rekall, a tenant leverages Furnace’s built-in Rekall app. The Rekall app must be paired with a tenant-provided backend, which issues Rekall commands to it. A 65 line backend was built for this purpose.

When a command is received, the Rekall app invokes Rekall in *L*₀. A Furnace-specific address

space [DG11a] was added to Rekall that converts Rekall's high-level reads and seeks into their equivalent cross-calls to L_2 , allowing Rekall to run against the tenant's live virtual machine from within Furnace. Running against live memory in place avoids the requirement to snapshot and download a memory dump out of the cloud, a potentially slow and bandwidth-intensive operation unsuitable for many security use cases. In terms of raw performance, comparing Furnace's Rekall app to Rekall operating against an equivalent file-based memory dump reveals their comparable speeds (4.05 sec to analyze 4GB of memory compared to Furnace's 20.88 sec).

App: Capturing and exporting guest memory. While Furnace's Rekall app is useful because it quickly analyzes live guest memory, for certain use cases it is also often desirable to obtain an offline copy. To demonstrate capturing and exporting guest memory out of the cloud, a tenant app and backend were developed (87 and 67 lines, respectively).

The app idles until the tenant's backend instructs it to perform a memory capture. The app is parameterized to compress and upload some or all of guest memory, and does so asynchronously in 4MB chunks, pausing briefly after every segment to listen for new instructions from the backend.

From a throughput performance standpoint, this app stresses each partition, the proxy, and the tenant's backend. During testing, it was found that the average end-to-end goodput was 40.47MBps (≈ 101 sec for a VM with 4GB RAM) for a backend running in a nearby LAN—conditions similar to if the app and backend both resided in the same cloud.

App: System call inspection. Furnace provides API support for tenant apps that consume syscall traces (debuggers, syscall analysis, etc.). A guest's system calls are challenging to capture due to their volume and the overhead associated with intercepting them [Fis15]. While there is nothing preventing a tenant app from performing syscall tracing directly, managing its low-level details (injecting, single-stepping, and restoring breakpoints in guest memory) from the L_0 partition would be inefficient. Instead, Furnace leverages DRAKVUF's syscall interception plugin. This plugin handles the details of tracing and sends only the results, a stream of syscall events (including the involved registers, calling process, user, and memory addresses), to the tenant app.

A 35 line tenant app was built to activate syscall tracing through the Furnace API and stream the resulting trace to its backend. During performance testing, it was found that overhead related to Furnace's cross-calls introduces a 5% in-guest performance penalty when compared to DRAKVUF tracing by itself. The small size of this syscall app indicates the ease at which a tenant can leverage Furnace to perform a complicated built-in VMI procedure. Furnace's deployment model also means it is easy for a tenant to temporarily activate an app under a guest for a specific purpose.

App: Process event monitoring. *Arav* [Bus] is a monitoring tool that uses VMI to extract events from a guest's userspace processes. Using a plugin specific to the guest program being monitored, Arav traces the program's function calls and reconstitutes its semantics. The original C source code for Arav (1219 lines) was ported to a 300 line Furnace app and a 40 line backend.

We use Arav to demonstrate Furnace’s potential for integration with the tenant’s security enterprise. Arav app instances were deployed under 10 Linux guests to capture and send a real-time stream of guest SSH logins/logouts and a guest kernel `commit_creds` function trace through Arav’s backend into a (simulated) tenant security event manager. As a substitute for more advanced analysis, this simple manager periodically picked a username at random from the stream and used the Furnace cloud service to deploy a Rekall app under the same guest. The Rekall app ran analysis that was then returned to the event manager.

This scenario highlights the ease at which a tenant security architecture can programmatically compose and deploy multiple Furnace apps. Because an app is decoupled from its backend, the backend can be run on tenant infrastructure and directly feed information into other tenant systems.

6.6 Evaluation and Results

Furnace was further evaluated in two ways. First, Furnace’s security properties were tested through the execution of apps containing malicious code. Second, Furnace’s performance was measured and compared with that of native VMI apps and another cloud VMI framework, CloudVMI.

The evaluation environment consisted of a single Dell R920 hypervisor. The hypervisor ran Xen version 4.8.1 with a Fedora 26 dom0 and was furnished with four 2.3GHz Intel Xeon E7-4870 processors and 256GB RAM. Furnace sandboxes were provisioned in dom0, which also supported the Furnace cloud service, hypervisor agent, and message proxy. The evaluation’s guests were fully virtualized Fedora 25 guests each with 1 vCPU, 1GB RAM, and 5GB storage.

6.6.1 Security Evaluation

The first experiments tested the security of Furnace’s sandbox with a specific focus on the attack vectors described in the threat model: preventing or mitigating attacks against Furnace’s VMI app APIs, sandbox construction, and resource brokers. Each test emulated an attacker’s actions: first by probing the sandbox for weaknesses, then by attempting a direct escape, and finally by attempting to find and exploit vulnerabilities in the cross-call interface. These experiments are informed by previous vulnerabilities in related sandboxes [LL12; GS14].

Sandbox enumeration. Serverless runtimes such as AWS Lambda are frequently audited by the security community²³ [KJ17]. An auditing app was built to perform the same steps of enumeration, such as attempting to read kernel logs, inspecting `procfs` for CPU and memory info, listing file systems and running processes, and identifying the system, kernel version, current working directory, and other environment variables.

²³<https://www.denialof.services/lambda/>

Result: The sandbox was found to be clean, with few details available to the app. Neither the `/dev/kmsg` device, `procfs`, nor `sysfs` are mapped into L_0 , so kernel logs, CPU and memory info, file system info, and process info are not available. Even if they were, attempts to read them (non-whitelisted behavior) would still result in the syscall inspector terminating the app. A minimum set of environment variables are artificially set in the sandbox during startup, further preventing enumeration.

Attacks by malicious apps. A set of malicious VMI apps that attempted to find weaknesses in the Furnace sandbox were implemented, and their execution was monitored. Recall that although Furnace components are initialized inside a permissive security policy, this policy is tightened prior to app execution.

Open arbitrary files. Attempts by an app to open files in directories mounted in the sandbox—`/tmp`, `/dev`, `/var`, etc.—were trapped into the syscall inspector. If the file was not on the inspector’s whitelist, the inspector terminated the app. Additionally, SELinux prevented file accesses not explicitly allowed by policy. Finally, because Furnace mounts a custom filesystem as the app partition’s root file system, there were no sensitive hypervisor files visible to apps.

Open a socket. Attempts to create a TCP socket or connect to a remote socket were also intercepted by the syscall inspector. While these calls are allowed during partition startup, once the app begins execution, they are forbidden. The app’s L_0 partition also lacks a network interface.

Spawn a process or thread. Prior to app execution, the syscall inspector disallowed the `clone` family of syscalls. Furnace also used cgroups to limit the maximum number of tasks running in a single partition.

Execve. Any attempt to replace a process with another via `execve` was trapped and resulted in the app being terminated by the syscall inspector.

Resource exhaustion. An app allocated increasingly large amounts of memory and stored it to disk and also entered an infinite loop. CPU and memory abuses are prevented by Linux cgroups. Disk usage is controlled by L_1 , which refuses to store additional data past the quota set in the app package manifest.

Attacks on cross-calls. Any crash in the sandbox’s two IPC interfaces would indicate a potentially exploitable bug. A malicious app used *protofuzz*²⁴ to fuzz the interfaces. While this was a black-box test, the malicious app was provided the Furnace protobuf schema so it could directly create and send IPC messages. The app first broadly tested the IPC interface by generating random protobuf messages. Next, the app used a more intelligent approach [Tak08] by mutating a set of well-formed seed messages that exercise resource broker logic, such as the code for VMI call pipelining, more thoroughly.

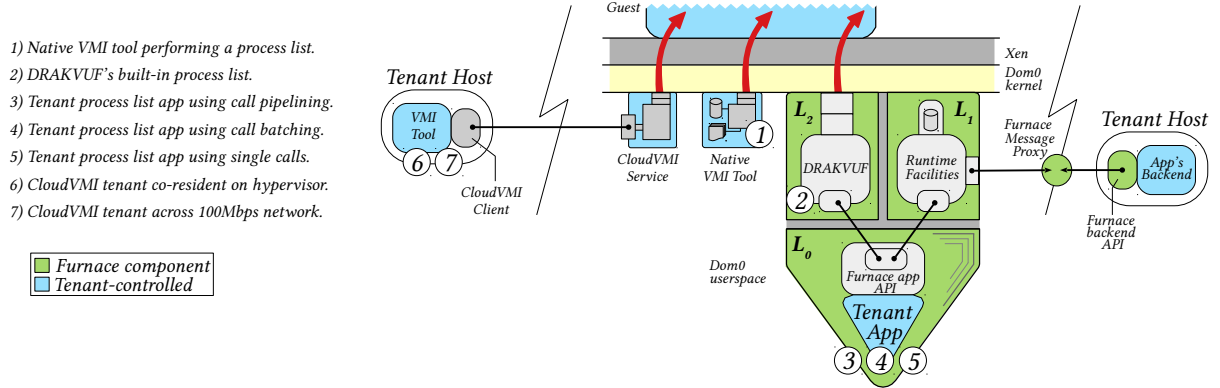
Result: Over 110 hours of fuzzing (150M mutated IPC messages) was unable to crash or cause

²⁴<https://github.com/trailofbits/protofuzz>

Table 6.4 VMI Performance Comparison

Approach	Guest Effective Performance		Time ms	App/Tool/CV Client				CV Service		Description
	calc/sec [†]	Δ%		Calls	Msgs RT	Mem MB	CPU jiffie	Mem MB	CPU jiffie	
1 Native VMI	327.9	100.0	0.13±0.13	301		25.1	21.5			LibVMI example code
2 DK Native	326.2	99.7	0.84±0.02	328	1	85.0	264.4			Built-in DK procedure
3 Pipeline* app	311.4	95.0	1.71±0.01	356	1	85.6	301.2			App w/ optimized API
4 Batch app	294.1	89.7	35.54±0.21	301	100	74.9	314.2			App w/ optimized API
5 Single app	283.4	86.4	98.52±0.72	304	304	85.4	324.5			App w/ unoptimized API
6 CV Local	311.8	95.1	34.93±0.24	315	315	2.0	<0.1	34.9	22.6	RPC client on local VM
7 CV Remote	258.3	78.8	541.96±4.43	315	315	2.0	0.3	34.8	21.2	RPC client across network

Round trip (RT). CloudVMI (CV). DRAKVUF (DK). [†]3 sec Sysbench CPU test. *Pipeline size: 120.
All measurements are averages with 0.95 confidence interval. Measured on a guest with 100 processes.

**Figure 6.14** Process list performance measurements were taken across a variety of VMI platforms, including a native VMI tool, CloudVMI, and Furnace VMI apps. This figure's circled points correspond to the rows in Table 6.4.

unexpected behavior in the L_1 and L_2 partitions, supporting the conclusion that the interfaces are unlikely to permit escape.

6.6.2 Performance Evaluation

The second set of experiments compared Furnace's performance with that of a native VMI tool and the CloudVMI framework. As has been done in previous work, the measurements focused on LibVMI's process list code example, which measures the time required to traverse the guest kernel's linked list of active processes. Several functionally identical process list programs were written to investigate differences between VMI approaches.

A variety of measurements were taken during these experiments: the time necessary for a VMI approach to perform the traversal; the slowdown caused when an approach pauses the guest to ensure guest memory consistency; the number of VMI calls issued by the approach, and, if applicable, the number of cross-call messages; and finally, the approach's resident memory and CPU time usage. Time stamps were taken immediately before and after each traversal to avoid measuring startup delays. To measure the performance penalty each VMI approach has on a guest, the sysbench CPU

benchmark²⁵ was ran for 3 seconds inside a guest, during which time the process list traversal was invoked once. Comparing the average amount of work sysbench accomplished with the baseline measurement in line 1 is used to quantify each approach's performance impact on the guest. While useful for comparing approaches, this metric focuses on a single use case and is not suited to be a general measure of VMI performance.

Intuitively, CV's RPC-based method is highly sensitive to network latency. CloudVMI was measured using two topologies: line 6 ran the CV client on a VM co-resident on the same hypervisor as the target guest, as would occur in the scenario where the cloud scheduler happened to provision both on the same hypervisor—a best case scenario for CV performance; Alternatively, line 7 placed the client across a network throttled at 100Mbps—a less ideal scenario.

Result. The results in Table 6.4 confirm the intuition that Furnace apps (lines 3–5) should be slower than a native VMI tool (line 1). When comparing optimized Furnace APIs (lines 3–4) to the unoptimized API (line 5), there is a clear pattern: as the number of VMI calls per cross-call message increase (line 3 shows 356:1), performance improves because fewer cross-calls are needed and less overhead occurs. Calling DRAKVUF's built-in procedures (line 2) yields Furnace's best performance, followed closely by Furnace's pipelined app. The fastest CloudVMI measurement is roughly equivalent to the Furnace batching app, but its slowest is over 5x slower than the slowest Furnace app. In addition to its security weaknesses, CloudVMI's performance also lags behind Furnace.

Furnace does require more memory and CPU time than CV. However, Furnace's results better reflect its true cost; there are no hidden costs as occur in CV or CloudPhylactor (both require a host/VM to run on, which comes with its own resource and maintenance costs). For example, with CloudPhylactor, a tenant monitoring VM with 1GB RAM would require 12x more memory than the equivalent Furnace app ($\approx 85\text{MB}$ per table 6.4).

Finally, although not shown in the table, raw app CPU performance was also tested and found to be unaffected by the sandbox. This measurement is important because the use of `ptrace` has been shown to increase overheads [KZ13]. This was confirmed by running the recursive algorithm for the Fibonacci sequence inside an app and comparing its wall clock performance with a functionally identical Python program running in `dom0`.

6.7 Discussion

Additional security measures. Although outside this chapter's scope, there are several additional security measures that would make Furnace resistant to specific types of threats, including certain micro-architectural and side-channel attacks. While Furnace does not directly implement its own

²⁵<https://github.com/akopytov/sysbench>

defenses against these attacks, we consider Furnace to be both compatible with many defenses against them [Moo15].

Provisioning Furnace Sandboxes in a VM. Instead of running in the hypervisor’s management domain, Furnace sandboxes could alternatively be provisioned in a privileged Furnace VM. One benefit of this approach is maintenance; encapsulating Furnace components in an isolated VM clearly separates them from the hypervisor, allowing Furnace-specific patches and updates to be applied independent of dom0. Additionally, because Furnace and XSM policies (such as those also used by CloudPhyactlor) likely do not interfere, this approach could also provide another layer of security by separating Furnace sandboxes from other sensitive dom0 data.

VMI App Control. Similar to how modern mobile app stores vet new apps, a cloud provider could implement a mandatory checkpoint for new apps so they can be manually or automatically reviewed, analyzed, signed, and archived by the cloud provider before use. Such apps could be analyzed using heuristics that detect the presence of certain side-channel attacks (i.e., tight loops using high precision timers) or mobile code (where the VMI app executes code snippets sent to it at runtime from the tenant’s backend).

Autonomous App Backends. The cloud provider could require that the app backend be ran autonomously. In this scenario, the backend would included in the app package and be automatically provisioned inside a tenant-billed container or VM—similar to OpenStack’s concept of service VMs.²⁶ This would allow a cloud provider to inspect the backend before it is used, and also make it more difficult for the tenant to modify the backend’s behavior at runtime.

Binary-based Restrictions. The Furnace sandbox is in contrast with the approach taken by *Native Client* [Yee09], which places controls on the runtime at a binary level. Instead of Native Client’s approach of re-compiling a tenant’s untrusted VMI app with built-in restrictions, Furnace’s sandbox focuses on constraining the app as it exists. Attacks that would be normally impossible with a Native Client binary would instead crash inside the Furnace sandbox. These two approaches are compatible, and adopting an approach similar to Native Client could be used to prevent certain low-level behaviors made by an app.

Language-based Restrictions. For interpreted languages such as Python, certain high-risk library modules (e.g., the ctypes library) could be removed from the Python standard library. This is conceptually similar to the approach taken by Google Compute Engine’s Python Runtime Environment.²⁷ Similarly, shared libraries could be removed the file system mounted into each Furnace sandbox partition.

Furnace Domain Specific Language. Alternatively, future work could yield a Domain Specific Language (DSL) to be used in lieu of a standard language for VMI apps. This DSL could further limit a tenant’s ability to attack the cloud.

²⁶<https://wiki.openstack.org/wiki/Horizon-NFV-configuration>

²⁷<https://cloud.google.com/appengine/docs/standard/python/runtime>

IPC performance. The round trip for a simple cross-call from L_0 to L_2 ranged between 190–230 μ s in our experimental setup. This includes protobuf serialization, ZMQ transport, and travel through UNIX domain sockets. Optimizing Furnace’s IPC channels could improve this performance. The Chrome sandbox recently experienced a 50% IPC speedup after similar IPC optimizations.²⁸

6.7.1 Related Work

Cloud VMI frameworks. In addition to the previously mentioned *CloudVMI* [Bae14], *CloudPhylactor* [Tau16], and *LiveCloudInspector* [ZR15], there have been other proposals to facilitate cloud forensics, malware analysis, and security. *CloudIDEA* [Fis15] proposed a wide-ranging cloud analysis system consisting of several VMI-based inputs, a dedicated analysis infrastructure, and an integrated decision engine. Compared to Furnace’s tenant self-service model, CloudIDEA’s focus was on a centralized, provider-driven approach. *FROST* [DS13] proposed methods for tenants to gain access to low-level cloud VM artifacts on OpenStack clouds, including firewall logs and memory dumps. Both CloudIDEA and FROST are similar to the preset model described in §6.3. Self-service clouds [SG12; But12; Ngu16] proposed allowing tenants to assign VMs special privileges to perform VMI and intercept block and network I/O. Furnace shares the tenant-empowering spirit of this proposal but not the large hypervisor architectural changes it implies. Finally, Furnace has parallels with software-defined networking (SDN) frameworks, including *FRESCO* [Shi13], which sought to enable modular SDN app development.

Constrained execution environments. Constraining untrusted code can be performed at a variety of locations [Shu16]. The namespace-based isolation used by Furnace is also leveraged by popular application containers (e.g., Docker, LXC, and rkt) and other sandbox projects such as Flatpak, Sandstorm, nsjail, and Firejail. The Furnace sandbox is distinct from these projects due to its use of `ptrace` and its resource broker model. Finally, while the use of `ptrace` has a mixed history as a sandbox mechanism [Swi17; Gar03], new unprivileged sandboxing mechanisms under development, including LandLock,²⁹ offer alternatives.

6.8 Summary

This chapter proposed Furnace, a framework that enables cloud tenants to run VMI applications under their cloud VMs. Compared to prior proposals, Furnace provides better features for cloud tenants, including support for arbitrary tenant VMI apps, speed improvements and cost reductions, and less infrastructure for a tenant to maintain. Furnace also meets the expectations of cloud providers: it is safe, accountable, and protective of cloud abstractions. Furnace’s sandbox withstood

²⁸<https://chromium.googlesource.com/chromium/src/+/master/mojo>

²⁹<https://landlock.io>

a multi-part security evaluation, supporting the conclusion that Furnace is a practical, adoptable cloud VMI framework.

CHAPTER

7

DIRECTIONS FOR HYPERVISOR-BASED SECURITY

This chapter summarizes this dissertation's findings, discusses HBS's current trajectory, and suggests areas of future work.

7.1 Dissertation Summary

Virtualization is increasingly important in modern systems, and implementing security measures at this level gives defenders important advantages. However, as explained in Chapter 1, there are existing limitations that have hampered the adoption of hypervisor-based security, including performance problems, the difficulty in integrating systems at the virtualization layer, and HBS's unavailability in the cloud.

In Chapter 2, a review on prior work led to a discussion on VMI techniques and related approaches. The advantages and disadvantages of implementing security at the operating system, hypervisor, and processor management levels allowed the argument to be made that hypervisor-based security has important advantages.

This dissertation then described the progress made toward more efficient hypervisor-based security systems that are easier to build, deploy, and utilize across a variety of platforms, especially the cloud.

First, two efficient HBS systems were introduced. Both systems focused on security-focused outcomes using VMI techniques which scale to a large number of adjacent guests and impose a low in-guest overhead. Chapter 3 introduced Goalkeeper, a HBS system that enforced guest process whitelisting policies. With the proper design, Goalkeeper showed that enforcing security policies inside many guest simultaneously from the hypervisor is practical. Chapter 4 introduced Arav, a HBS system that traced guest process function calls in real-time. Arav's approach was especially useful because it allowed defenders to monitor guests which are normally difficult to monitor using traditional means. Together, these systems exemplify effective designs for scalable HBS systems.

Next, HBS development and integration challenges were addressed. In Chapter 5, a development methodology was proposed to re-engineer an existing best-of-breed host agent into a hyperagent. Hyperagents take advantage of VMI's strengths while retaining their prior characteristics—especially the years of practitioner experience and best practices inherent in the original agent. This methodology allows VMI to augment an existing security enterprise, and provides a practical on-ramp for VMI tools in production.

Finally, the problem of cloud adoption was studied in Chapter 6. Due to a variety of reasons, HBS systems are presently unavailable in the cloud for tenant use. The proposed solution to this problem, Furnace, provides an open source cloud VMI framework for multi-hypervisor, multi-tenant cloud ecosystems, enabling a tenant to safely run self-submitted VMI applications underneath their cloud VMs. Furnace's design was informed by prior proposals, but takes a different approach by sandboxing tenant code on the cloud provider's hypervisor. As an adoptable, practical, open source framework, Furnace is designed to encourage adoption by cloud providers.

While wide HBS adoption cannot be achieved overnight, this dissertation provided concrete examples and methodologies that aim to spur further development in this area. To encourage future work, many of its artifacts are publicly available. Arav's ability to monitor of FreeBSD guests was merged into the upstream LibVMI project.¹ `libhyperagent` and many of Goalkeeper's VMI components were released as artifacts.² Finally, the Furnace cloud VMI framework is available.³

7.2 Trends

This section discusses several VMI trends: VMI tools have become easier to write, are compatible with an increasing number of platforms, and are increasingly offered by commercial security vendors and open source projects.

New platforms. VMI has recently been applied to several interesting new platforms, processors, and hypervisors.

¹<https://github.com/libvmi/libvmi/pull/551>

²<https://github.com/mbushou/hyperagents>

³<https://github.com/mbushou/furnace>

There is ongoing work to bring an upstream VMI API to KVM.⁴ Prior to this, VMI performance on KVM suffered due to the requirement to use the QEMU Machine Protocol (QMP) interface. Efforts to improve performance required a non-upstream patch for LibVMI,⁵ or required direct application integration into QEMU or KVM source code. Many prior works based on KVM took the latter route and were therefore difficult to run in production.

Additionally, as a result of the Meltdown and Spectre vulnerabilities, there was an immediate reaction to bring Xen-based VMI to AMD-based CPUs.⁶ Prior to this effort, many VMI techniques were only available on Intel chips.

Microsoft Virtualization-based Security. Outside current academic VMI work, Windows's recent adoption of VBS⁷ [Yos17] further indicates its usefulness as a security technique. VBS contains an interesting mix of elements from both VMI and TrustZone concepts and is used in Windows to enforce kernel integrity and protect certain privileged processes.

Desktop Isolation. Qubes⁸ and OpenXT⁹ are desktop-oriented operating systems that use virtualization to isolate workloads at different trust levels. Since they inherently use virtualization, they are suitable platforms for VMI. As an initial investigation in this area, Qubes was used in Chapter 5 to evaluate the hyperagent methodology.

Finally, the increased use of embedded virtualization¹⁰ opens up new opportunities for VMI. Xen in particular has focused on being an attractive hypervisor for such applications.¹¹ It is anticipated that virtualization will become more important as embedded devices continue to become more sophisticated.

Commercial endeavors. Several recently-released commercial products have adopted VMI techniques. Companies such as VMRay¹² use VMI techniques to analyze potentially malicious code and are often deployed using a cloud-based or middlebox approach [Wil13; Wil12; Wil07].

Zentific,¹³ Bromium,¹⁴ Joe Security,¹⁵ and BitDefender Hypervisor Introspection,¹⁶ instead focus on protecting end user workstations, and take special care to avoid performance problems.^{17,18}

Tool development. Several prior works have indicated the improving ease at which new VMI tools

⁴<https://marc.info/?l=kvm&m=149762057618807>

⁵<https://lists.nongnu.org/archive/html/qemu-devel/2015-10/msg05898.html>

⁶<https://lists.xenproject.org/archives/html/xen-devel/2018-02/msg00107.html>

⁷<https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs>

⁸<https://www.qubes-os.org/>

⁹<https://openxt.org/>

¹⁰<https://www.zededa.com/development/linux-foundation-announces-embedded-virtualization-platform/>

¹¹<https://dornerworks.com/xen/virtuosity/virtuosity>

¹²<https://www.vmrays.com/>

¹³<https://www.zentific.com/>

¹⁴<https://www.bromium.com/platform/our-technology.html>

¹⁵<https://www.joesecurity.org/joe-sandbox-technology>

¹⁶<https://www.bitdefender.com/business/hypervisor-introspection.html>

¹⁷<https://www.youtube.com/watch?v=Qd8J3oYPVF0>

¹⁸<https://www.youtube.com/watch?v=7wi6mznRn3E>

can be developed [Bau15; Wan15a; Jai14; Sun15]. Corroborating this trend, in the last year, several open source efforts have integrated VMI into other security projects. *ivmi*¹⁹ allows DRAKVUF to be controlled remotely. Radare2²⁰ now features VMI support in its extras repository.²¹ VMI event support was recently integrated into Rekall²² [PV17]. Recent work by *Bhatt et al.* used VMI for cybersecurity education [Bha18].

7.3 Future Work

Cloud adoption. Chapter 6 discussed the benefits that VMI could bring to cloud workloads. Implementing an adoptable cloud VMI framework, Furnace, is a first step in this direction. Additional work remains.

Despite the possibility that a major cloud provider deliberately chooses to adopt a cloud VMI framework outright, perhaps the first logical step toward broader cloud adoption is success in a private cloud. Large corporations often maintain private clouds, which may be more suitable for initial testing due to their more controlled environment and potentially less dangerous clientele (the company’s employees). There is modest precedence for this, as NFM [Sun14] was deployed to the IBM Research Compute Cloud [Sun16]. Efforts to integrate Furnace with cloud management platforms popular in private clouds (OpenStack, Eucalyptus, Apache CloudStack, and OpenNebula) is ongoing work.

Automated VMI translation. Chapter 5 focused on a methodology to equip traditional agent-based systems with VMI. After the methodology was applied, these hyperagents take advantage of VMI’s benefits yet retain a similar look and feel, allowing external systems that interface with the agent system (including human operators) to remain unchanged.

There are opportunities for future work in this area, especially with regard to automatically and transparently translating an actuator’s syscalls into the equivalent VMI operation. If successful, such a translation layer would greatly simplify the process of creating a hyperagent. However, while this may be relatively straightforward for reads and writes to the guest’s disk, it is more difficult for operations on the guest’s memory.

Reconciling with hardware. The last several years have seen both the further proliferation of hardware-based security features as well as the disclosure of a remarkable number of hardware vulnerabilities.

The ability to perform VMI is inextricably linked to the hardware features inherent in a hypervisor’s CPUs (Chapter 1’s discussion on VMI primitives touched on this fact). It is hardware features

¹⁹<https://github.com/v-p-b/ivmi>

²⁰<https://www.radare.org>

²¹<https://github.com/Wenzel/radare2-extras>

²²<https://github.com/fireeye/rvmi>

that allow EPT violations, breakpoint injection, and register-based events to be used in conjunction with VMI. It should therefore not be surprising that it is also hardware features that can prevent VMI. Processor designers have not ignored potential cloud threat models where the cloud provider (or a lone administrator) is itself malicious [Rut15]. In turn, both AMD and Intel processors have features that prevent certain aspects of VMI. AMD's Secure Encrypted Virtualization (SEV) encrypts a VM's main memory, rendering memory analysis through VMI ineffective. Similarly, Intel Software Guard Extensions (SGX) creates a hardware-protected enclave around a program that makes external snooping difficult [VB17; SL18].

Additionally, the microarchitectural attacks in early 2018 cast uncertainty particularly in the multi-tenancy space. Spectre [Koc18], Meltdown [Lip18], and related attacks [Tri18; Che18; MR18] could allow tenant's to observe hypervisor memory and the memory of adjacent tenant guests. Ensuring VMI remains possible in these increasingly complex hardware environments will be important in the coming years. New hardware also means new opportunities, and so harnessing new VMI techniques from these future CPUs will remain an interesting research direction.

The strong semantic gap problem. Perhaps the most urgent academic endeavor is the strong semantic gap problem [Jai14]. This problem stems from the implicit assumption that most modern VMI tools make: the guest operating system will faithfully observe the properties it had when the semantic mapping was generated. Prior work has shown this assumed trust to be exploitable [Bah10; Len14a].

VMI's ability to tap into hardware events points to a potential solution. Is there a way to provably show that a combination of VMI, hardware invariants, and hardware features could prevent an attacker from hiding inside a guest? Various works have gone back and forth on the attacker's difficulty in fooling VMI tools, but to a degree this balance will always be affected by changes to hardware. Intuitively, as processor designers react to recent vulnerabilities and continue to refine their own trusted execution environments, this landscape stands to change frequently in the coming years.

7.4 Concluding Remarks

Throughout this body of work, I developed a deep sense of respect for the kernel, hardware, and hypervisor development communities. As a part of observing and working within these communities, I have come to regard VMI as a valuable security enabler that, like many techniques, carries its bad with its good.

Implementing hypervisor-based security through VMI is unapologetically systems-heavy, but also benefits from a theoretical perspective (such as when crossing the semantic gap). VMI is immensely powerful, but is limited to virtualized infrastructure and adds to the complexity of the hypervisor. It is a useful security primitive, but seems like the perennial outsider, looked upon

cautiously (or with agitation) by kernel developers. It's reasonably quick for what it does, yet doesn't quite seem fast enough.

VMI has a compelling, intuitive concept, yet requires specialized knowledge to develop. It functions as a surreptitious source of data for machine learning, threat feeds, and telemetry, yet it's in a crowded field, so serious academic attention is also paid to other data sources. Finally, it is lauded for its access to raw guest data, except that hastily coded VMI tools can be tricked by sophisticated adversaries.

Hypervisor-based security systems are powerful, useful security capabilities that have the potential to tip the balance toward the defender. This dissertation advances the state of the art in a direction toward a security environment that can more fluidly and seamlessly integrate VMI techniques into the larger security picture.

BIBLIOGRAPHY

- [Ace13] Aceto, G. et al. "Cloud Monitoring: A Survey". *Computer Networks* **57.9** (2013), pp. 2093–2115.
- [AA06] Adams, K. & Agesen, O. "A Comparison of Software and Hardware Techniques for x86 Virtualization". *ACM SIGOPS Operating Systems Review* **40.5** (2006), pp. 2–13.
- [Ahn12] Ahn, J. et al. "Revisiting Hardware-assisted Page Walks for Virtualized Systems". *Proceedings of the 39th Annual International Symposium on Computer Architecture*. ISCA '12. Portland, Oregon: IEEE Computer Society, 2012, pp. 476–487.
- [And72] Anderson, J. P. *Computer Security Technology Planning Study. Volume 2*. Tech. rep. Anderson (James P) and Co Fort Washington PA, 1972.
- [Aru] Arulraj, L. et al. "Improving Virtualized Storage Performance with Sky". *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '17.
- [Aza09] Azab, A. M. et al. "HIMA: A Hypervisor-Based Integrity Measurement Agent". *Computer Security Applications Conference, 2009. ACSAC '09. Annual*. 2009.
- [Aza10] Azab, A. M. et al. "HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity". *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS '10. Chicago, Illinois, USA: ACM, 2010, pp. 38–49.
- [Bae14] Baek, H. W. et al. "CloudVMI: Virtual Machine Introspection as a Cloud Service". *2014 IEEE International Conference on Cloud Engineering* (2014), pp. 153–158.
- [Bah10] Bahram, S. et al. "DKSM: Subverting Virtual Machine Introspection for Fun and Profit". *29th IEEE Symposium on Reliable Distributed Systems*. 2010.
- [Bar03] Barham, P. et al. "Xen and the Art of Virtualization". *ACM SIGOPS Operating Systems Review*. Vol. 37. 5. ACM. 2003, pp. 164–177.
- [Bau15] Bauman, E. et al. "A Survey on Hypervisor-Based Monitoring: Approaches, Applications, and Evolutions". *ACM Computing Surveys* (2015).
- [Bay15] Bays, L. R. et al. "Virtual Network Security: Threats, Countermeasures, and Challenges". *Journal of Internet Services and Applications* (2015).
- [Bel05] Bellard, F. "QEMU, a Fast and Portable Dynamic Translator." *USENIX Annual Technical Conference, FREENIX Track*. 2005, pp. 41–46.

- [Bha18] Bha, M. et al. "Using Virtual Machine Introspection for Operating Systems Security Education". *49th ACM Technical Symposium on Computer Science Education (SIGCSE)*. Baltimore, Maryland, USA, 2018.
- [Bha08] Bhargava, R. et al. "Accelerating Two-dimensional Page Walks for Virtualized Systems". *SIGPLAN Not.* **43.3** (2008), pp. 26–35.
- [BR17] Bushouse, M. & Reeves, D. "Goalkeeper: Comprehensive Process Enforcement from the Hypervisor". *Computers & Security* (2017).
- [BR18] Bushouse, M. & Reeves, D. "Hyperagents: Migrating Host Agents to the Hypervisor". *Proceedings of the Eighth ACM on Conference on Data and Application Security and Privacy*. CODASPY '18. Tempe, Arizona, USA, 2018.
- [Bus] Bushouse, M. et al. "Arav: Monitoring a Cloud's Virtual Routers". *Proceedings of the 12th Annual Conference on Cyber and Information Security Research*.
- [But12] Butt, S. et al. "Self-service Cloud Computing". *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. Raleigh, North Carolina, USA: ACM, 2012, pp. 253–264.
- [Car12] Carbone, M. et al. "Secure and Robust Monitoring of Virtual Machines Through Guest-assisted Introspection". *International Workshop on Recent Advances in Intrusion Detection (RAID)*. Springer. 2012.
- [Che16] Chen, D. D. et al. "Towards Automated Dynamic Analysis for Linux-based Embedded Firmware". *Proceedings of the Network and Distributed Systems Security Symposium*. 2016.
- [Che18] Chen, G. et al. "SgxPectre Attacks: Leaking Enclave Secrets via Speculative Execution". *arXiv preprint arXiv:1802.09085* (2018).
- [CN01] Chen, P. M. & Noble, B. D. "When Virtual is Better than Real". *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*. IEEE. 2001, pp. 133–138.
- [Che08] Chen, X. et al. "Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malware". *IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. 2008.
- [Coh11] Cohen, M. I. et al. "Distributed Forensics and Incident Response in the Enterprise". *Digital Investigation* **8** (2011), S101–S110.
- [CD16] Costan, V. & Devadas, S. "Intel SGX Explained." *IACR Cryptology ePrint Archive* **2016** (2016), p. 86.

- [Cri16] Cristalli, S. et al. "Micro-Virtualization Memory Tracing to Detect and Prevent Spraying Attacks". *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016, pp. 431–446.
- [Cui16] Cui, L. et al. "Piccolo: A Fast and Efficient Rollback System for Virtual Machine Clusters". *2016 45th International Conference on Parallel Processing (ICPP)*. 2016.
- [Cui15] Cui, L. et al. "Lightweight Virtual Machine Checkpoint and Rollback for Long-running Applications". *Algorithms and Architectures for Parallel Processing: 15th International Conference, ICA3PP 2015, Zhangjiajie, China, November 18-20, 2015, Proceedings, Part III*. 2015.
- [Del10] Delaet, T. et al. "A Survey of System Configuration Tools." *Proceedings of the 23rd Conference on Large Installation System Administration*. Vol. 10. LISA'10. 2010, pp. 1–8.
- [DK14] Delgado, B. & Karavanic, K. L. "Performance implications of System Management Mode". *2013 IEEE International Symposium on Workload Characterization (IISWC)* **00** (2014), pp. 163–173.
- [Den] Deng, Z. et al. "SPIDER: Stealthy Binary Program Instrumentation and Debugging via Hardware Virtualization". *2013 Proceedings of the 29th Annual Computer Security Applications Conference*.
- [Din08] Dinaburg, A. et al. "Ether: Malware Analysis via Hardware Virtualization Extensions". *Proceedings of the 15th ACM Conference on Computer and Communications Security*. CCS '08. 2008.
- [DG11a] Dolan-Gavitt, B. et al. *Leveraging Forensic Tools for Virtual Machine Introspection*. Tech. rep. Georgia Institute of Technology, 2011.
- [DG11b] Dolan-Gavitt, B. et al. "Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection". *IEEE Symposium on Security and Privacy (SP)*. IEEE. 2011, pp. 297–312.
- [DG13] Dolan-Gavitt, B. et al. "Tappan Zee (North) Bridge: Mining Memory Accesses for Introspection". *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*. CCS '13. ACM, 2013.
- [DG15] Dolan-Gavitt, B. et al. "Repeatable Reverse Engineering with PANDA". *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*. PPREW-5. Los Angeles, CA, USA: ACM, 2015, 4:1–4:11.
- [Duf06] Duflot, L. et al. "Using CPU System Management Mode to Circumvent Operating System Security Functions". *CanSecWest/core06* (2006).
- [Dun02] Dunlap, G. W. et al. "ReVirt: Enabling Intrusion Analysis Through Virtual-machine Logging and Replay". *SIGOPS Oper. Syst. Rev.* **36**.SI (2002), pp. 211–224.

- [DS13] Dykstra, J. & Sherman, A. T. "Design and Implementation of FROST: Digital Forensic Tools for the OpenStack Cloud Computing Platform". *Digital Investigation* **10** (2013), S87–S95.
- [Egi] Egi, N. et al. "Fairness Issues in Software Virtual Routers". *2008 In Proc. of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow*.
- [Enc08] Enck, W. et al. "PinUP: Pinning User Files to Known Applications". *2008 Annual Computer Security Applications Conference (ACSAC)*. 2008, pp. 55–64.
- [FD11] Fernandes, N. C. & Duarte, O. C.M. B. "XNetMon: A Network Monitor for Securing Virtual Networks". *2011 IEEE International Conference on Communications (ICC)*. 2011.
- [Fis15] Fischer, A. et al. "CloudIDEA: A Malware Defense Architecture for Cloud Data Centers". *On the Move to Meaningful Internet Systems: OTM 2015 Conferences*. Springer International Publishing, 2015, pp. 594–611.
- [Fra08] Fraser, T. et al. "VICI Virtual Machine Introspection for Cognitive Immunity". *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*. 2008.
- [Fu16] Fu, Y. "Bridging the Semantic Gap in Virtual Machine Introspection via Binary Code Reuse". PhD thesis. The University of Texas at Dallas, 2016.
- [FL13] Fu, Y. & Lin, Z. "Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection". *ACM Transactions on Information System Security* **16.2** (2013), 7:1–7:29.
- [Gal13] Galvin, P. B. et al. *Operating System Concepts*. John Wiley & Sons, Inc., 2013.
- [Gar03] Garfinkel, T. "Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools". *NDSS*. Vol. 3. 2003, pp. 163–176.
- [GR03] Garfinkel, T. & Rosenblum, M. "A Virtual Machine Introspection Based Architecture for Intrusion Detection". *Proceedings of the Network and Distributed Systems Security Symposium*. 2003.
- [GR05] Garfinkel, T. & Rosenblum, M. "When Virtual Is Harder than Real: Security Challenges in Virtual Machine Based Computing Environments." *HotOS*. 2005.
- [Gik10] Gikas, C. "A General Comparison of FISMA, HIPAA, ISO 27000 and PCI-DSS Standards." *Information Security Journal: A Global Perspective* **19.3** (2010), pp. 132 –141.
- [Gol73] Goldberg, R. P. *Architectural Principles for Virtual Computer Systems*. Tech. rep. Harvard University Division of Engineering and Applied Physics, 1973.
- [GS14] Gorenc, B. & Spelman, J. "Thinking outside the sandbox: Violating trust boundaries in uncommon ways". *Black Hat USA* (2014).

- [Gor15] Gorobets, M. et al. “Attacking Hypervisors via Firmware and Hardware”. *Black Hat USA Conference*. 2015.
- [Grr] *GRR Rapid Response*. <https://github.com/google/grr>.
- [Gup08] Gupta, D. et al. “Difference Engine: Harnessing Memory Redundancy in Virtual Machines”. *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI’08. San Diego, California: USENIX Association, 2008, pp. 309–322.
- [Han] Han, S. et al. “Myth and Truth about Hypervisor-Based Kernel Protector: The Reason Why You Need Shadow-Box” ().
- [HM] Harris, L. C. & Miller, B. P. “Practical Analysis of Stripped Binary Code”. *2005 ACM SIGARCH Computer Architecture News* ().
- [Hen17] Henderson, A. et al. “DECAF: A Platform-Neutral Whole-System Dynamic Binary Analysis Platform”. *IEEE Transactions on Software Engineering* **43.2** (2017), pp. 164–184.
- [HB17] Hetzelt, F. & Buhren, R. “Security Analysis of Encrypted Virtual Machines”. *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE ’17. Xi’an, China: ACM, 2017, pp. 129–142.
- [HC13] Hizver, J. & Chiueh, T. C. “Cloud-Based Application Whitelisting”. *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*. 2013.
- [HC14] Hizver, J. & Chiueh, T.-c. “Real-time Deep Virtual Machine Introspection and Its Applications”. *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE ’14. ACM, 2014.
- [Int16] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer’s Manual*. Vol. 3C. 2016.
- [Ira14] Irazoqui, G. et al. “Wait a minute! A fast, Cross-VM attack on AES”. *International Workshop on Recent Advances in Intrusion Detection (RAID)*. Springer. 2014, pp. 299–319.
- [Jae03] Jaeger, T. et al. “Analyzing Integrity Protection in the SELinux Example Policy”. *Proceedings of the 12th Conference on USENIX Security Symposium*. SSYM’03. 2003.
- [Jae06] Jaeger, T. et al. “PRIMA: Policy-reduced Integrity Measurement Architecture”. *Proceedings of the Eleventh ACM Symposium on Access Control Models and Technologies*. SACMAT ’06. Lake Tahoe, California, USA: ACM, 2006, pp. 19–28.
- [Jai14] Jain, B. et al. “SoK: Introspections on Trust and the Semantic Gap”. *2014 IEEE Symposium on Security and Privacy*. 2014.

- [Jav] Javaid, S. et al. "Atomizer: Fast, Scalable and Lightweight Heap Analyzer for Virtual Machines in a Cloud Environment". *6th Layered Assurance Workshop, 2012*.
- [Joh14] Johnson, D. et al. "Composable Multi-Level Debugging with Stackdb". *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. VEE '14*. Salt Lake City, UT, 2014, pp. 213–225.
- [Jon06] Jones, S. T. et al. "Antfarm: Tracking Processes in a Virtual Machine Environment". *in Proc. of the USENIX Annual Technical Conf.* 2006.
- [Kar90] Karger, P. A. et al. "A VMM Security Kernel for the VAX Architecture". *IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE. 1990, pp. 2–19.
- [Kar91] Karger, P. A. et al. "A Retrospective on the VAX VMM Security Kernel". *IEEE Transactions on Software Engineering* **17.11** (1991), pp. 1147–1165.
- [KF91] Kelem, N. L. & Feiertag, R. J. "A Separation Model for Virtual Machine Monitors". *Proceedings of the Symposium on Research in Security and Privacy*. IEEE. 1991, pp. 78–86.
- [KZ13] Kim, T. & Zeldovich, N. "Practical and Effective Sandboxing for Non-root Users". *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX '13)*. San Jose, CA, 2013.
- [Kir14] Kirat, D. et al. "BareCloud: Bare-metal Analysis-based Evasive Malware Detection." *USENIX Security Symposium*. 2014, pp. 287–301.
- [Kiv07] Kivity, A. et al. "KVM: The Linux Virtual Machine Monitor". *Proceedings of the Linux Symposium*. Vol. 1. 2007, pp. 225–230.
- [KW08] Kleidermacher, D. & Wolf, M. "MILS virtualization for Integrated Modular Avionics". *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*. 2008, pp. 1.C.3–1–1.C.3–8.
- [Koc18] Kocher, P. et al. "Spectre Attacks: Exploiting Speculative Execution". *ArXiv e-prints* (2018). arXiv: 1801.01203.
- [KN14] Kourai, K. & Nakamura, K. "Efficient VM introspection in KVM and Performance comparison with Xen". *20th Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE. 2014, pp. 192–202.
- [Kri10] Krishnan, S. et al. "Trail of Bytes: Efficient Support for Forensic Analysis". *Proceedings of the 17th ACM Conference on Computer and Communications Security*. ACM. 2010, pp. 50–60.
- [KJ17] Krug, A. & Jones, G. "Hacking Serverless Runtimes: Profiling Lambda, Azure, and more". *Black Hat USA* (2017).

- [Law96] Lawton, K. P. “Bochs: A Portable PC Emulator for Unix/X”. *Linux Journal* **1996**.29es (1996), p. 7.
- [Lea17] Leach, K. et al. “Scotch: Combining Software Guard Extensions and System Management Mode to Monitor Cloud Resource Usage”. *International Workshop on Recent Advances in Intrusion Detection (RAID)*. 2017.
- [Len12] Lengyel, T. K. et al. “Virtual Machine Introspection in a Hybrid Honeypot Architecture”. *5th Workshop on Cyber Security Experimentation and Test (CSET)*. 2012.
- [Len14a] Lengyel, T. K. et al. “Pitfalls of Virtual Machine Introspection on Modern Hardware”. *1st Workshop on Malware Memory Forensics (MMF)*. 2014.
- [Len14b] Lengyel, T. K. et al. “Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System”. *Proceedings of the 30th Annual Computer Security Applications Conference. ACSAC '14*. ACM, 2014.
- [Lev92] Levine, J. R. et al. *lex & yacc*. O'Reilly Media, Inc., 1992.
- [Li13] Li, N. et al. “vMON: An Efficient out-of-VM Process Monitor for Virtual Machines”. *10th International Conference on High Performance Computing and Communications*. IEEE. 2013, pp. 1366–1373.
- [Lig14] Ligh, M. et al. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. Wiley, 2014.
- [Lip18] Lipp, M. et al. “Meltdown”. *ArXiv e-prints* (2018). arXiv: 1801.01207.
- [Lit08] Litty, L. et al. “Hypervisor Support for Identifying Covertly Executing Binaries.” *USENIX Security Symposium*. 2008, pp. 243–258.
- [LL12] Liu, Z. & Lovet, G. “Breeding Sandworms: How to fuzz your way out of Adobe Reader X's sandbox”. *Black Hat EUROPE* (2012).
- [Los98] Loscocco, P. A. et al. “The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments”. *Proceedings of the 21st National Information Systems Security Conference*. Vol. 10. 1998, pp. 303–314.
- [Lov10] Love, R. *Linux Kernel Development*. Pearson Education, 2010.
- [Lu17] Lu, K. et al. “Flexible Page-level Memory Access Monitoring Based on Virtualization Hardware”. *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '17. Xi'an, China: ACM, 2017, pp. 201–213.
- [Lut16] Lutas, D. et al. “Secure Virtual Machine for Real Time Forensic Tools on Commodity Workstations”. *Innovative Security Solutions for Information Technology and Communications*. Cham: Springer International Publishing, 2016, pp. 193–208.

- [MD73] Madnick, S. E. & Donovan, J. J. "Application and Analysis of the Virtual Machine Approach to Information System Security and Isolation". *Proceedings of the Workshop on Virtual Computer Systems*. ACM. 1973, pp. 210–224.
- [MR18] Maisuradze, G. & Rossow, C. "Speculose: Analyzing the Security Implications of Speculative Execution in CPUs". *arXiv preprint arXiv:1801.04084* (2018).
- [MK12] Mankin, J. & Kaeli, D. "Dione: A Flexible Disk Monitoring and Analysis Framework". *International Workshop on Recent Advances in Intrusion Detection (RAID)*. Springer. 2012, pp. 127–146.
- [Mar10] Martignoni, L. et al. "Live and Trustworthy Forensic Analysis of Commodity Production Systems". *International Workshop on Recent Advances in Intrusion Detection (RAID)*. Springer. 2010, pp. 297–316.
- [May06] Mayer, F. et al. *SELinux by Example: Using Security Enhanced Linux*. Pearson Education, 2006.
- [McC10] McCune, J. M. et al. "TrustVisor: Efficient TCB Reduction and Attestation". *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE. 2010, pp. 143–158.
- [MG11] Mell, P., Grance, T., et al. "The NIST definition of cloud computing" (2011).
- [Mir17] Miramirkhani, N. et al. "Spotless Sandboxes: Evading Malware Analysis Systems using Wear-and-Tear Artifacts". *IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, pp. 1009–1024.
- [Mis16] Mishra, P. et al. "NvCloudIDS: A Security Architecture to Detect Intrusions at Network and Virtualization Layer in Cloud Environment". *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. 2016.
- [MK17] Miyama, S. & Kourai, K. "Secure IDS Offloading with Nested Virtualization and Deep VM Introspection". *22nd European Symposium on Research in Computer Security*. ESORICS. 2017.
- [Moo15] Moon, S.-J. et al. "Nomad: Mitigating Arbitrary Cloud Side Channels via Provider-Assisted Migration". *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2015, pp. 1595–1606.
- [Ngu16] Nguyen, H. et al. "Exploring Infrastructure Support for App-based Services on Cloud Platforms". *computers & security* **62** (2016), pp. 177–192.
- [Pan17] Pan, J. et al. "Digtool: A Virtualization-Based Framework for Detecting Kernel Vulnerabilities". *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 149–165.

- [Pay08] Payne, B. D. et al. "Lares: An Architecture for Secure Active Monitoring Using Virtualization". *IEEE Symposium on Security and Privacy*. 2008.
- [Pay07] Payne, B. D. et al. "Secure and Flexible Monitoring of Virtual Machines". *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)* (2007).
- [PV17] Pfoh, J. & Vogl, S. "rVMI: A New Paradigm for Full System Analysis". *Black Hat USA Conference*. 2017.
- [Pfo11] Pfoh, J. et al. "Nitro: Hardware-Based System Call Tracing for Virtual Machines". *Advances in Information and Computer Security: 6th International Workshop, IWSEC 2011, Tokyo, Japan, November 8-10, 2011. Proceedings*. 2011.
- [PG74] Popek, G. J. & Goldberg, R. P. "Formal Requirements for Virtualizable Third Generation Architectures". *Communications of the ACM* **17.7** (1974), pp. 412–421.
- [Pro03] Provos, N. "Improving Host Security with System Call Policies." *USENIX Security Symposium*. 2003, pp. 257–272.
- [QS07] Quynh, N. A. & Suzaki, K. "Xenprobes, a Lightweight User-Space Probing Framework for Xen Virtual Machine". *USENIX annual technical conference proceedings*. 2007.
- [Rua14] Ruan, X. *Platform Embedded Security Technology Revealed: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*. Apress, 2014.
- [Rut06] Rutkowska, J. "Introducing Blue Pill". *Invisible Things Laboratory* (2006).
- [Rut15] Rutkowska, J. "Intel x86 Considered Harmful". *the Invisible Things Lab* (2015).
- [SY17] Sanders, M. & Yue, C. "Automated Least Privileges in Cloud-based Web Services". *Proceedings of the Fifth ACM/IEEE Workshop on Hot Topics in Web Systems and Technologies*. HotWeb '17. San Jose, California: ACM, 2017, 3:1–3:6.
- [SS94] Sandhu, R. S. & Samarati, P. "Access Control: Principle and Practice". *Communications Magazine, IEEE* (1994).
- [SL18] Schwarz, M. & Lipp, M. "When Good Turns Evil: Using Intel SGX to Stealthily Steal Bitcoins". *Black Hat Asia* (2018).
- [Ses07] Seshadri, A. et al. "SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes". *ACM SIGOPS Operating Systems Review*. Vol. 41. 6. ACM. 2007, pp. 335–350.
- [SL16] Sgandurra, D. & Lupu, E. "Evolution of Attacks, Threat Models, and Solutions for Virtualized Systems". *ACM Comput. Surv.* **48.3** (2016), 46:1–46:38.

- [Sha] Shacham, H. "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)". *Proceedings of the 14th ACM Conference on Computer and Communications Security*.
- [Shi15] Shin, E. C. R. et al. "Recognizing Functions in Binaries with Neural Networks". *24th USENIX Security Symposium*. 2015.
- [Shi13] Shin, S. et al. "FRESCO: Modular Composable Security Services for Software-Defined Networks". *NDSS*. 2013.
- [Shu16] Shu, R. et al. "A Study of Security Isolation Techniques". *ACM Comput. Surv.* **49.3** (2016), 50:1–50:37.
- [SH12] Sikorski, M. & Honig, A. *Practical Malware Analysis*. No Starch Press, 2012.
- [Sol16] Solutions, V. E. "Verizon 2016 Data Breach Investigations Report". *verizon.com verizon.com* (2016).
- [Son08] Song, D. et al. "BitBlaze: A New Approach to Computer Security via Binary Analysis". *Information Systems Security: 4th International Conference, ICISS 2008, Hyderabad, India, December 16-20, 2008. Proceedings*. 2008.
- [ST15] Souza, W. A. R. d. & Tomlinson, A. "SMM-Based Hypervisor Integrity Measurement". *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing*. 2015, pp. 362–367.
- [Sou16] Souza, W. A. R. de. "On Using the System Management Mode for Security Purposes". PhD thesis. University of London, 2016.
- [Spe99] Spencer, R. et al. "The Flask Security Architecture: System Support for Diverse Security Policies". *Proceedings of the 8th Conference on USENIX Security Symposium - Volume 8*. SSYM'99. Washington, D.C.: USENIX Association, 1999, pp. 11–11.
- [Spe16] Spensky, C. et al. "LO-PHI: Low-Observable Physical Host Instrumentation for Malware Analysis". *Proceedings of 2016 Network and Distributed System Security Symposium (NDSS'16)*. 2016.
- [SG12] Srivastava, A. & Ganapathy, V. "Towards a Richer Model of Cloud App Markets". *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*. ACM. 2012, pp. 25–30.
- [Sun16] Suneja, S. "Non-intrusive Virtual Systems Monitoring". PhD thesis. University of Toronto, 2016.
- [Sun14] Suneja, S. et al. "Non-intrusive, Out-of-band and Put-of-the-box Systems Monitoring in the Cloud". *ACM SIGMETRICS Performance Evaluation Review*. Vol. 42. 1. ACM. 2014, pp. 249–261.

- [Sun15] Suneja, S. et al. “Exploring VM Introspection: Techniques and Trade-offs”. *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '15. Istanbul, Turkey: ACM, 2015, pp. 133–146.
- [Swi17] Swiecki, R. “Promises and Pitfalls of Sandboxes”. Presented at CONFidence. 2017.
- [Lin] *System V Application Binary Interface*. <https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf>.
- [Tak08] Takanen, A. et al. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2008.
- [Tau16] Taubmann, B. et al. “CloudPhylactor: Harnessing Mandatory Access Control for Virtual Machine Introspection in Cloud Data Centers”. *2016 IEEE Trustcom/BigDataSE/ISPA*. 2016, pp. 957–964.
- [Tri18] Trippel, C. et al. “MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols”. *arXiv preprint arXiv:1802.03802* (2018).
- [Uhl05] Uhlig, R. et al. “Intel Virtualization Technology”. *Computer* **38**.5 (2005), pp. 48–56.
- [VB17] Van Bulck, J. et al. “Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution”. *Proceedings of the 26th USENIX Security Symposium*. USENIX Association. 2017.
- [Vas13] Vasudevan, A. et al. “Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework”. *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. SP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 430–444.
- [VV09] Velte, A. & Velte, T. *Microsoft Virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.
- [Wal02] Waldspurger, C. A. “Memory Resource Management in VMware ESX Server”. *ACM SIGOPS Operating Systems Review* **36**.SI (2002), pp. 181–194.
- [Wan15a] Wang, C. et al. “Exploring Efficient and Robust Virtual Machine Introspection Techniques”. *Algorithms and Architectures for Parallel Processing*. Springer, 2015.
- [Wan15b] Wang, G. et al. “Hypervisor Introspection: A Technique for Evading Passive Virtual Machine Monitoring.” *WOOT*. 2015.
- [Wil16] Wilhelm, F. “XenPwn: Breaking Paravirtualized Devices”. *Black Hat USA Conference*. 2016.
- [Wil07] Willems, C. et al. “Toward Automated Dynamic Malware Analysis Using CWSandbox”. *IEEE Security Privacy* **5**.2 (2007), pp. 32–39.

- [Wil12] Willems, C. et al. “Down to the Bare Metal: Using Processor Features for Binary Analysis”. *Proceedings of the 28th Annual Computer Security Applications Conference. ACSAC '12*. Orlando, Florida, USA: ACM, 2012, pp. 189–198.
- [Wil13] Willems, C. et al. “Cxpinspector: Hypervisor-based, hardware-assisted system monitoring”. *Ruhr-Universitat Bochum, Tech. Rep* (2013), p. 12.
- [Wu14] Wu, R. et al. “System Call Redirection: A Practical Approach to Meeting Real-World Virtual Machine Introspection Needs”. *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 2014, pp. 574–585.
- [Wur10] Wurster, G. D. “Security Mechanisms and Policy for Mandatory Access Control in”. PhD thesis. Carleton University Ottawa, 2010.
- [Xue08] Xue, F. “Attacking Antivirus”. *Black Hat Europe Conference*. 2008.
- [Yee09] Yee, B. et al. “Native Client: A Sandbox for Portable, Untrusted x86 Native Code”. *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE. 2009, pp. 79–93.
- [Yin07] Yin, H. et al. “Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis”. *Proceedings of the 14th ACM Conference on Computer and Communications Security. CCS '07*. ACM, 2007.
- [Yoo17] Yoon, M.-K. et al. “VirtualDrone: Virtual Sensing, Actuation, and Communication for Attack-Resilient Unmanned Aerial Systems”. *Proceedings of the 8th International Conference on Cyber-Physical Systems*. ACM. 2017, pp. 143–154.
- [Yos17] Yosifovich, P. et al. *Windows Internals 7th Ed*. Microsoft Press, 2017.
- [ZR15] Zach, J. & Reiser, H. P. “LiveCloudInspector: Towards Integrated IaaS Forensics in the Cloud”. *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer. 2015, pp. 207–220.
- [Zha13] Zhang, F. et al. “SPECTRE: A dependable introspection framework via System Management Mode”. *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2013, pp. 1–12.
- [Zha14] Zhang, F. et al. “Hypercheck: A Hardware-assisted Integrity Monitor”. *Dependable and Secure Computing, IEEE Transactions on* (2014).
- [Zha15] Zhang, F. et al. “Using Hardware Features for Increased Debugging Transparency”. *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE. 2015.
- [Zha17] Zhao, S. et al. “Seeing Through The Same Lens: Introspecting Guest Address Space At Native Speed”. *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 799–813.

APPENDICES

APPENDIX

A

GOALKEEPER WHITELISTING MECHANISM

A.1 The Whitelisting Check

Goalkeeper's policy ruleset determines whether processes are authorized or unauthorized and is oriented around a set of per-process *whitelisting rules*. Inside each whitelisting rule, a set of *whitelisting conditions* specify the allowable states of that process, and a set of *options* specify Goalkeeper's responses when the rule is encountered or violated. Figure A.1 gives a high-level view of how a rule is used to evaluate a guest process.

First, ① Goalkeeper intercepts a guest process and looks up its corresponding policy rule using the guest process's full file path as the key. If there is no corresponding rule and a default rule has not been specified, the guest process will be labeled as unauthorized. Each condition in the policy rule will be checked, and the process attribute in question must be in the condition's set of acceptable attributes. For example, ② the *comm* condition passes because `top` $\subseteq$ `{top, (top)}`, and ③ the *parent* condition passes because `/usr/bin/zsh` $\subseteq$ `{/usr/bin/bash, /usr/bin/zsh}`. ④ The rule lacks *children* or *map-count* conditions, so these are not checked and pass by default, meaning that only conditions present in the rule can cause the rule to fail. A rule without any conditions will always pass. In this example, ⑤ the process's *entry-point-hash* condition fails because it does not match the rule's hash, and because all conditions must be satisfied, this process will fail its

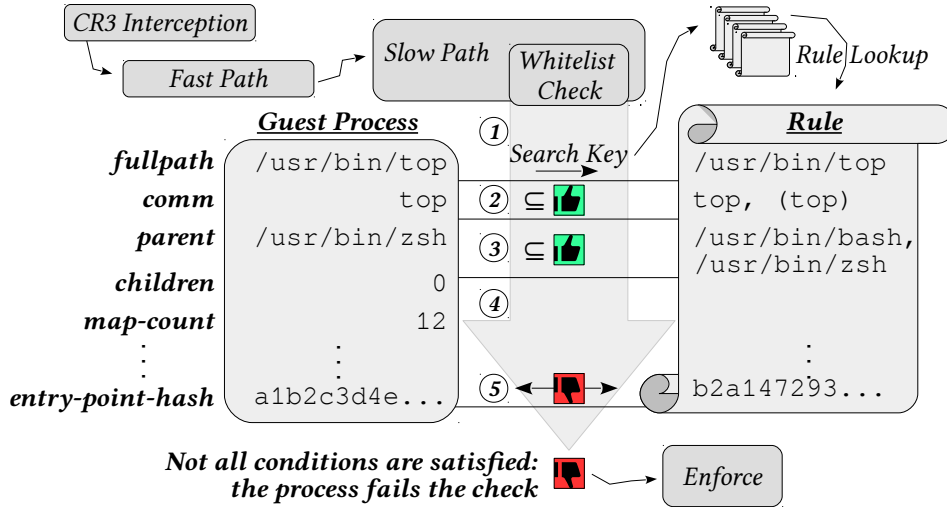


Figure A.1 The whitelisting check.

whitelisting check.

Algorithm 1 describes the whitelisting check in detail, including when and how options are used. In the DECIDE procedure, the *call-function* option allows the rule to delegate its decision to a Python function (for example to check an external database). The result is sent to logging facilities specified by the *log* option.

Algorithm 1 Whitelist Check

```

1: procedure WHITELISTCHECK(proc, ruleset)
2:   rule ← LOOKUPRULE(proc, ruleset)
3:   if rule = ∅ then
4:     rule ← GETDEFAULT(ruleset)
5:   decideObj ← DECIDE(proc, rule)
6:   for each log in GETLOGS(rule) do
7:     LOG(log, proc, rule, decideObj)
8:   if FAILED(decideObj) then
9:     termseq ← GETTERMSEQ(rule)
10:    STARTENFORCEMENT(proc, termseq)
11:   return decideObj
12: procedure DECIDE(proc, rule)
13:   decideObj ← INITIALIZETOPASS()
14:   for each condition in rule do
15:     decideObj ← TEST(condition, proc, decideObj)
16:     if FAILED(decideObj) then
17:       break
18:   for each func in GETFUNCS(rule) do
19:     decideObj ← func(proc, rule, decideObj)
20:   return decideObj

```

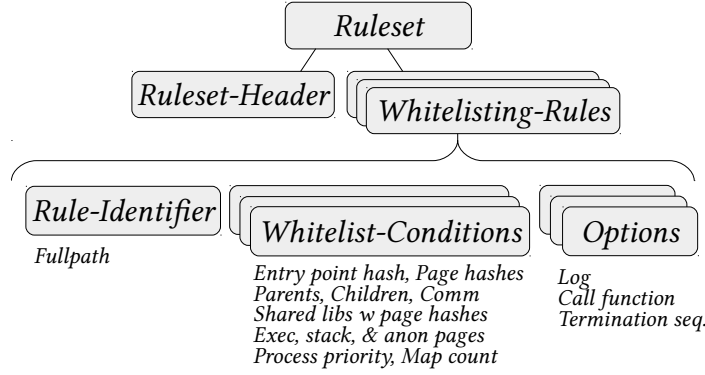


Figure A.2 Grammar hierarchy.

A.2 Grammar Specification

The Goalkeeper context-free grammar was implemented using lex and yacc [Lev92] according to the summarized EBNF specification below (its hierarchy is depicted in Figure A.2).

```

<ruleset> := <ruleset-header> <whitelisting-rule>*1
<ruleset-header> := '-' NAME ',' <rule-id> '-'
<comment> := ('#' | ';' ) STRING+
<whitelisting-rule> := <rule-id> <whitelist-cond>+
<rule-id> := '[' PATH ']'
<whitelist-cond> := <entry-hash-kv> | <page-hash-kv>
    | <argv-kv> | <username-kv> | <parent-name-kv>
    | <children-name-kv> | <shared-libs-kv> | <cswitch-kv>
    | <priv-anon-pages-kv> | <priority-kv> | <map-count-kv>
    | <exec-pages-kv> | <stack-pages-kv> | <comm-kv>
<entry-hash-kv> := 'entry-point-hash' '=' HASH,+
<page-hash-kv> := 'page-hash' '=' <indexed-hash>,+
<comm-kv> := 'comm' '=' NAME,+
<argv-kv> := 'arguments' '=' <argument-set>,+
<username-kv> := 'username' '=' NAME,+
<parent-name-kv> := 'parent' '=' <rule-id>,+
<children-name-kv> := 'children' '=' '0' | <rule-id>,+
<shared-libs-kv> := 'shared-libs' '=' (PATH
    ('{' <indexed-hash>+ '}' )? ),+
<cswitch-kv> := 'context-switches' '=' <uint-set>
<priv-anon-pages-kv> := 'priv-anon-pages' '=' <uint-set>
<priority-kv> := 'priority' '=' <uint-set>
<map-count-kv> := 'map-count' '=' <uint-set>
<exec-pages-kv> := 'exec-pages' '=' <uint-set>
  
```

¹The entity to the left of * can be repeated any number of times, + must occur 1 or more times, and ? is optional (0 or 1 times). A prepended comma (for example: ,+) indicates that commas separate multiple values.

```

<stack-pages-kv> := 'stack-pages' '=' <uint-set>
<argument-set> := ('' STRING+ '') | ('r'' REGEX '')
<uint-set> := UINT | (UINT '-' UINT)
<indexed-hash> := UINT ':' <single-hash>
<options> := 'options' '=' '{'
    <term-seq-kv>? <log-kv>? <call-kv>? '}'
<term-seq-kv> := 'policy-violation-sequence' '=' <term-seq>
<term-seq> := 'disable' | ('sigterm' | 'sigkill' | 'corrupt-stack' | 'corrupt-text'),+
<log-kv> := 'log' '=' LOG,+
<call-kv> := 'call-function' '=' PYTHON_FUNC,+

```

Summarized tokens.

NAME: Series of 1+ alphanum, underscore, or hyphens.

STRING: Series of 1+ ASCII printables (inc. space).

LOG: A valid log method defined by the Goalkeeper admin.

PYTHON_FUNC: A valid callable Python function.

HASH: A valid SHA256 hash.

UINT: An integer between 0 and $2^{32} - 1$.

PATH: A valid absolute file system path.

REGEX: A valid regular expression.

A.3 Example Policy

```

# Ruleset name and default rule.
--example_fedora_25_ruleset, unknown-process--
# Rule for the top utility.
[/usr/bin/top]
arguments      = "top", r"top -d [0-9] +"
comm           = top
username       = bob, alice, charlie
parent         = /usr/bin/bash, /usr/bin/zsh
children       = 0
exec-pages     = 1600-1650
context-switches = 0-15
stack-pages    = 34
; This hash is specific to procs-ng-3.3.10-11.fc24.x86_64.rpm.
entry-point-hash =
    9b94143f1085c973c11ef6df94424e8ad2fa6cf3eaa07aef1f58176920c571ec
page-hash      = {
    0:eca47b3bb7ad4afe6687f84224a8df127f1f596a6cf3505a7dc2921179e39045,
    1:81aa74eec166e7f647073ab059b4c5667f4f02dedc00569a3a95984bb4044970,
    2:af1900ce69d0d596a7c8c33fbf9ffd01d46e6f939918606764cc053186efddcf
}

```

```

shared-libs      =
/usr/lib64/libsystemd.so,
/lib64/libc.so.6: {
    0:058a12ac2139594697b43376d8981f16a2794a9f435c26f82a09ae5cff1510ce,
    33:9627e679dfb03975126a04c0d84cae1b2e5efdfb366231e19101206a80040e67
},
/lib64/libdl.so.2
# Rule for the zsh shell.
[/usr/bin/zsh]
comm             = zsh
username         = bob, alice, charlie
children         = /usr/bin/top
priv-anon-pages  = 10-15
priority         = 120
map-count        = 2-10
options          = {
    policy-violation-sequence = sigterm, corrupt-stack
    call-function             = checkForScript
    log                       = sendToSplunk
}
# Default rule to be applied if no other rules match.
[unknown-process]
; This condition will always fail.
username         = invalid
options          = {
    policy-violation-sequence = sigterm, corrupt-stack
    call-function             = checkSiem, checkDatabase
    log                       = sendToSplunk, console
}

```

APPENDIX

B

OTHER POTENTIAL HYPERAGENTS

This appendix describes how four other agents could be re-engineered into hyperagents. For brevity, a discussion on phase 3 is omitted and the diagrams accompanying each agent focus on the “top” and “bottom” portions of Figure B.1.

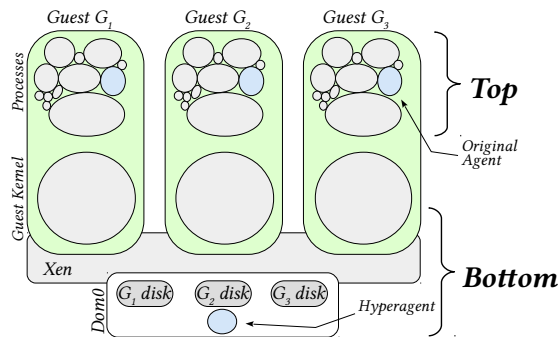
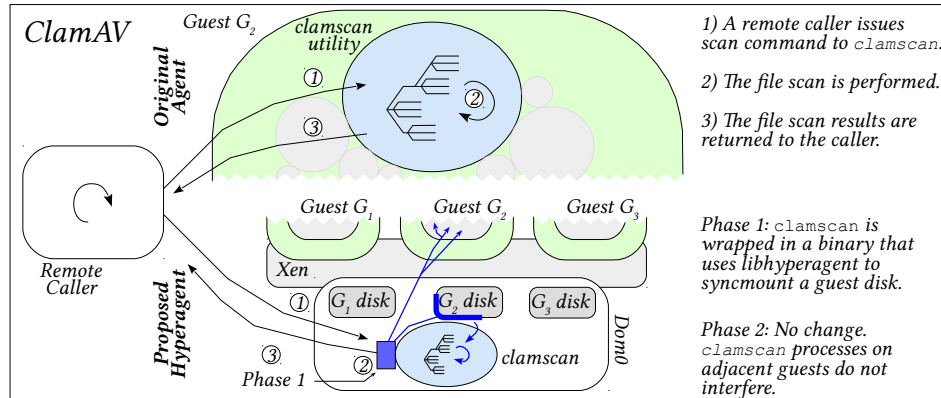


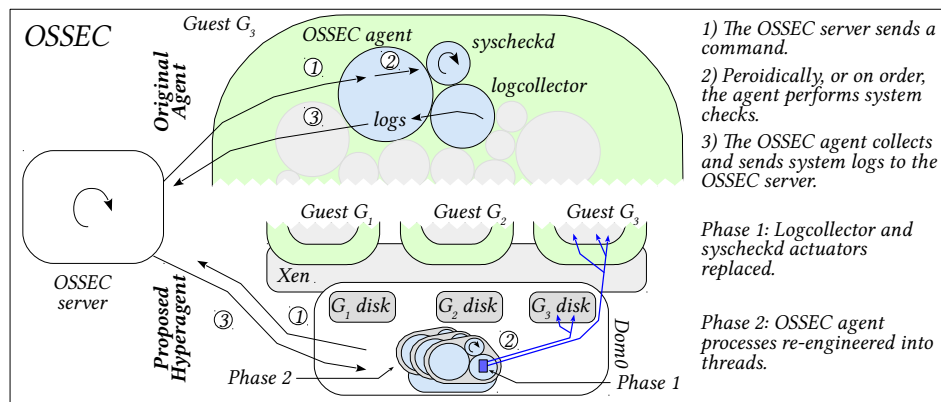
Figure B.1 The following figures reference these indicated areas.

B.1 Security: ClamAV Scanner



*ClamAV*¹ is a file-based anti-virus scanner that iteratively scans files in a guest file system hierarchy. ClamAV scanning activity can be controlled by a GUI, a standalone command line utility, or a networked agent daemon called `clamd`. **Phase 1:** A small libhyperagent-enabled binary is used to “wrap” the ClamAV command line utility, `clamscan`, such that a guest filesystem is temporarily SyncMounted immediately prior to initiating a scan. **Phase 2:** Since multiple `clamscan` instances can already run concurrently without interference, phase 2 did not require changes.

B.2 Security: OSSEC Intrusion Detection



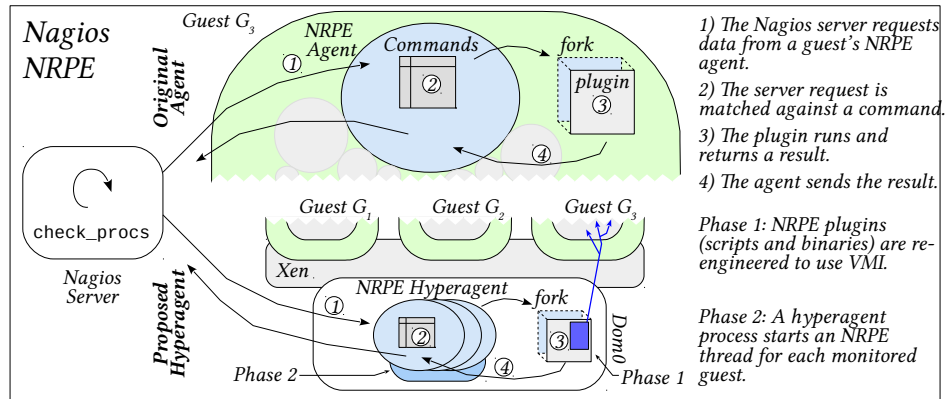
*OSSEC*² is an open source log analysis project that consists of several interrelated programs which collect, parse, analyze, and respond to the contents of a host’s log messages. OSSEC can also initiate file integrity scans that look for configuration problems and indicators of rootkits. **Phase 1:** OSSEC’s core log collection and analysis functionality relies on file-system-based actuators which are already

¹<https://github.com/vrtadmin/clamav-devel>

²<https://github.com/ossec/ossec-hids>

supported by `libhyperagent`. Other OSSEC features would require additions to `libhyperagent`, including inotify-like VMI functionality to detect changes in files, VMI functions that walk a guest's device tree, and the ability to spawn in-guest binaries (or temporarily hijack existing processes in a manner similar to work queue interception). Examples of these techniques can be found in prior work [Joh14; Len14b]. **Phase 2:** Much like the GRR agent, a hyperagent management layer could be introduced to instantiate OSSEC agent threads that monitor each guest.

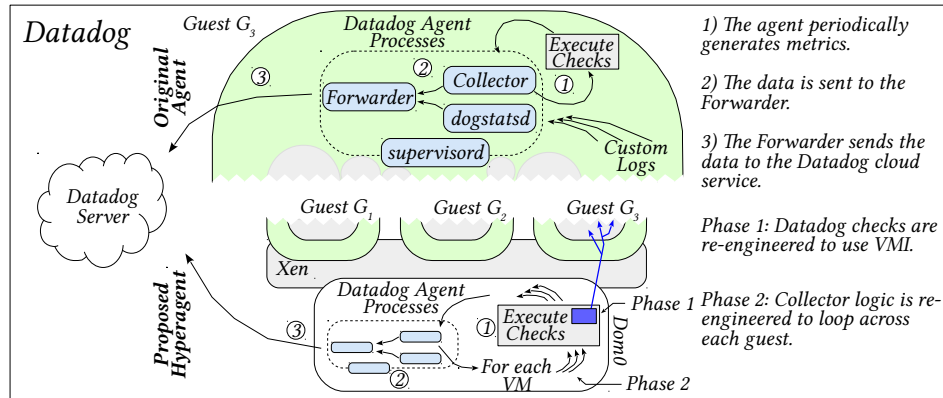
B.3 Monitoring: Nagios NRPE



The Nagios Remote Plugin Executor³ (NRPE) host agent allows a remote Nagios server to query local host state. The NRPE agent responds to a server's requests by looking up and invoking the the request's corresponding NRPE plugin (a binary) as a child process. **Phase 1:** Because each plugin communicates with the main NRPE process via standard output, the plugins are the only agent components that need to be re-engineered in phase 1. The majority of the official plugins are written in C and contain a single actuator site that could be re-engineered to leverage `libhyperagent` to inspect a guest. **Phase 2:** A hyperagent management layer would be introduced to instantiate a NRPE thread for each monitored guest. Unlike the GRR agent, the NRPE agent makes use of global variables, which would need to be isolated on a per thread basis.

³<https://github.com/NagiosEnterprises/nrpe>

B.4 Monitoring: Datadog



The Datadog agent⁴ sends host metrics and events to the Datadog cloud-based monitoring service. At runtime, the Datadog agent collector periodically loops through a configurable series of AgentCheck objects, each of which gathers metrics by reading `/proc` (or equivalent), invoking a program, or interacting with a local service. **Phase 1:** Each AgentCheck object contains an actuator site, and most sites simply wrap a shell command. `libhyperagent` could be leveraged in building a second set of VMI-enabled binaries that mimic the syntax of the original commands. **Phase 2:** Unlike GRR, the DataDog agent's design does not require a new management layer. Instead, modifications could be made to the agent loop that calls each AgentCheck object. A second outer loop could be added that calls the set of AgentCheck objects for each monitored VM.

⁴<https://github.com/DataDog/dd-agent>