# ECONOMICALLY PROTECTING COMPLEX, LEGACY OPERATING SYSTEMS USING SECURE DESIGN PRINCIPLES

Bhushan Jain

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2022

## ABSTRACT

Bhushan Jain: Economically Protecting Complex, Legacy Operating Systems using Secure Design Principles
(Under the direction of Donald Porter)

In modern computer systems, complex legacy operating systems, such as Linux, are deployed ubiquitously. Many design choices in these legacy operating systems predate a modern understanding of security risks. As a result, new attack opportunities are routinely discovered to subvert such systems, which reveal design flaws that spur new research about secure design principles and other security mechanisms to thwart these attacks. Most research falls into two categories: encapsulating the threat and redesigning the system from scratch. Each approach has its challenge. Encapsulation can only limit the exposure to the risk, but not entirely prevent it. Rewriting the huge codebase of these operating systems is impractical in terms of developer effort, but appealing inasmuch as it can comprehensively eliminate security risks. This thesis pursues a third, understudied option: retrofitting security design principles in the existing kernel design. Conventional wisdom discourages retrofitting security because retrofitting is a hard problem, may require the use of new abstractions or break backward compatibility, may have unforeseen consequences, and may be equivalent to redesigning the system from scratch in terms of effort.

This thesis offers new evidence to challenge this conventional wisdom, indicating that one can economically retrofit a comprehensive security policy onto complex, legacy systems. To demonstrate this assertion, this thesis firstly surveys the alternative of encapsulating the threat to the complex, legacy system by adding a monitoring layer using a technique called Virtual Machine Introspection, and discusses the shortcomings of this technique. Secondly, this thesis shows how to enforce the principle of least privilege by removing the need to run setuid-to-root binaries with administrator privilege. Finally, this thesis takes the first steps to show how to economically retrofit secure design principles to the OS virtualization feature of the Linux kernel called containers without rewriting the whole system. This approach can be applied more generally to other legacy systems.

**TABLE OF CONTENTS**

# LIST OF TABLES

## LIST OF FIGURES

# CHAPTER 1: INTRODUCTION

Legacy systems in production are used in more hostile environments than what the designers and developers envisioned. Dennis Ritchie [172] said, "The first fact to face is that UNIX was not developed with security, in any realistic sense, in mind; this fact alone guarantees a vast number of holes." James Morris [14] said, "On one hand, we're doing things today with computers that were maybe pipe dreams 40 years ago, yet we're still relying on operating systems designed decades ago." It is almost impossible to code large, complex systems without creating bugs. On average, about 30 bugs are introduced per 1000 lines of code written in commercial software [100]. Some of these bugs are also vulnerabilities, as the bugs can be exploited by an attacker to manipulate the program or gain control of the system. Just like bugs, it is almost impossible to code large, complex systems without creating vulnerabilities.

Language or compiler-based security mechanisms can reduce the number of these vulnerabilities and their impact, but these mechanisms do not ubiquitously work for all code. These security mechanisms include memory safety, software fault isolation, code integrity, data-flow integrity, control-flow integrity, address space randomization, data randomization, penetration testing, and formal verification. However, these mechanisms do not work out-of-the-box for all code written in C. Most of these tools are effective for application binaries or other high-level languages such as Java, but the code of, say, the Linux kernel, is too big and complex to apply formal methods or many security mechanisms. For instance, providing memory safety in Linux is such a daunting task, that security researchers only recently started running Rust code inside Linux [187].

Saltzer and Schroeder devised design principles to help developers design systems and applications with security as the prime objective [175]. These secure design principles are economy of design, fail-safe defaults, complete mediation, open design, separation of privilege, least privilege, least common mechanism, and psychological acceptability. Modern OSes were designed and initially implemented for much less hostile environments than in modern deployments, so it is unsurprising that early design and implementation choices deviate from these principles at several points. Thus, taking the Linux kernel as a running example, the early stages of design and implementation of a legacy OS like Linux did not adhere to all of these security design principles. Moreover there has been a lot of resistance to add security features in Linux, especially from Linus Torvalds, the founder of Linux, who famously said “I don’t trust security people to do sane things.” [13]

With the benefit of several intervening decades of research on programming languages and security, designers and developers of new applications or systems with security as the primary objective can leverage programming language tools and security design principles. But, to improve the security of legacy codebases, there are one of the three options: (1) accept the risk, (2) try to encapsulate the risk to limit the exposure [114], or (3) rewrite the legacy code while following the new techniques and the security design principles [128]. However, these options are either hard or not practical. Firstly, the Linux kernel code is too big to even completely know the risk. Secondly, encapsulation can only limit the exposure to the risk, not completely prevent it. Finally, it is not practical to rewrite over 27 million lines of Linux. Approaches for hardening a legacy codebase must scale in terms of developer effort.

A better approach to protecting a legacy system like Linux would be to retrofit these security design principles in the existing kernel design, if we had economical and practical techniques. But alas, this is an open problem. We also need to rethink the design of the legacy system code to maintain the old abstractions and interfaces for backward compatibility, but make the code adapt to security design principles.

The thesis of this dissertation is that *security design principles can be economically, and practically enforced on complex, legacy kernel code, without the need to rewrite the whole operating system.* First, Chapter 2 will show how encapsulation of threats in the Linux kernel is handled using virtual machine introspection, and the limitations of the technique. Second, Chapter 3 will show how to reduce privilege escalation vulnerabilities in the Linux kernel by making just a few isolated changes, and rethinking the design to enforce the least privilege principle. Finally, Chapter 4 will use the security design principles and compiler tools to provide memory isolation for the containers feature of the Linux Kernel.

The first technical contribution of the thesis is a survey of virtual machine introspection techniques from more than 100 papers over 12 years, and organize the VMI design space to create a framework that can help reason about design choices while building new systems. Threats in the Linux kernel are often neutralized or contained by encapsulating the Linux kernel into a virtual machine, and monitoring that virtual machine for unusual behavior. Virtual machine introspection (VMI) adds a hypervisor layer and monitors the kernel after moving it into a VM. Some of these VMI solutions inspect the memory of a virtual machine by executing concurrently, and often asynchronously with the VM. These asynchronous VMI solutions passively monitor and detect attacks on the VM after the fact. On the other hand, some VMI solutions are synchronous, mediating guest operations inline to prevent security policy violations by protecting the VM memory. This thesis observes that all VMI solutions that prevent an attack are based on the secure design principle of complete mediation and the technique of memory protection. However, an attacker can also change the kernel data objects and the interpretation of kernel data structures to show the monitor a view of the system different from reality. This thesis observes that this stronger attack is often obviated by generous threat models, rendering encapsulation an imperfect solution.

The second technical contribution of the thesis is to enforce the Least Privilege Principle on the untrusted user by removing the need to run special setuid-to-root binaries with escalated privileges, and without changing the user experience. Unprivileged users of modern Unix systems access safe subsets of otherwise privileged system functionality through trusted, setuid-to-root binaries. The problem with setuid-to-root binaries is that they violate the Least Privilege Principle (LPP) [175], and create opportunities for privilege escalation attacks. Chen et al. [70] show that many privilege escalation attacks go through setuid-to-root binaries, even on SELinux [145] or AppArmor [28]. A study of 28 most popular setuid-to-root binaries, which account for all setuid-to-root binaries installed on roughly 89.5% of the Debian and Ubuntu systems surveyed using the Lintian reports [32], revealed that only eight system calls and two other system interfaces underlie the vast majority of root privilege requirements. This thesis solves this problem by designing and implementing a simple, efficient framework for migrating policies from setuid-to-root binaries into the kernel as linux security module (LSM) policies, obviating the need for these binaries to run with escalated privileges in nearly all situations, and removing a source of privilege escalation.

The third technical contribution of the thesis is to give containers VM-like memory isolation properties by economically isolating the memory of containers without sacrificing the containers' lighter weight. Docker and Linux containers (LXC) have been gaining popularity as a solution to trade off security isolation for efficiency and compatibility across different OS distributions. For instance, Google has adopted Docker for their cloud model to compete with Amazon cloud services [154]. Like VMs, containers encapsulate the threats in the Linux kernel, but containers are a more lightweight and faster alternative to VMs. VMs consume a lot of extra resources like memory footprint and take a while to start up, as they are booting a whole new Operating System.

Although containers are lighter weight than virtual machines, they are more susceptible to attacks than VMs due to the design choice to share the same kernel in the host as well as the guest. The OS virtualization isolates containers using namespace by creating container-specific copies of selected global kernel objects, such as process descriptor list, and redirecting the use of these objects to the appropriate container-specific copy. The host exposes the system call interface to the container framework. Thus, the same system call vulnerabilities that can be exploited by an userspace application are exposed in containers. The problem is that by exploiting the host kernel vulnerabilities, the guest can access the whole host kernel, including the memory belonging to other guests as well as the host. To give containers isolation properties comparable to VMs, we need to redesign the kernel to provide the guest with access to only the necessary parts of host memory, and use complete mediation to enforce memory protection in the kernel. We take the first steps to show how the design principles for protection mechanisms and compiler tools can be retroactively fitted on the huge Linux kernel code by slightly rethinking the design, and making small changes.

Section 1.1 describes the security design principles and the Section 1.2 describes security mechanisms that complement these principles. In addition to following the security design principles, these security mechanisms are also necessary to build secure systems.

## 1.1 Security Design Principles

There is no way to build a perfectly secure system, mainly because of the difficulty in proving that no unauthorized actions can occur in the system. Even in systems designed and implemented with security as a prime objective, design and implementation flaws can allow a malicious user to circumvent the access constraints. For any system, the effectiveness of protection mechanisms is based on the system's ability to prevent security violations and unauthorized access. However, it is very difficult to guarantee that all unauthorized accesses are denied. Almost all security systems are built for a threat model that excludes a few threats. It is often possible to crash a system or deny system access to other users using a sophisticated attack. New vulnerabilities are often discovered in systems designed long ago.

In the absence of a methodical technique to build secure systems, Saltzer and Schroeder described a few design principles based on empirical evidence and experience. These design principles can guide the design and implementation of a system without security flaws. The security design principles are discussed as follows:

- **Separation of Privilege:** This principle requires that access is granted based on two or more conditions to enforce multi-factor authentication. Linux doesn't inherently support multi-factor authentication, but relies on third-party applications to provide this feature.

- **Least Privilege:** This principle states that every program and every user of the system should operate using the least set of privileges necessary to complete the job. A monolithic kernel like Linux violates this principle by running all its code at the privileged level, and due to features like setuid-to-root binaries that allow unprivileged users to use privileged kernel operations.

- **Least Common Mechanism:** This principle minimizes the number of mechanisms common to more than one user and is depended on by all users. Trusted computing base (TCB) is an ubiquitously used security metric to count the number of lines of code that is trusted by the system. Thus, enforcing least common principle can reduce the TCB to minimum. Linux violates this principle because any kernel data can be accessed anywhere in the kernel code. Thus, all 27.8 million lines of kernel code and data are common to all users and processes, and part of the TCB.

- **Economy of Mechanism:** This principle requires that the design and implementation of a system are kept as small as possible. A monolithic kernel like Linux violates this principle by adding all the drivers and subsystems to the kernel.

- **Fail-safe Defaults:** This principle requires that the access is based on permission instead of exclusion. Linux follows this principle by using discretionary access control, which allows access only based on the identity of the user.

- **Complete Mediation:** This principle states that every access to every object must be checked for authority. By convention, Linux design requires kernel developers to correctly check for authority, and sometimes these checks are missing, thus violating the complete mediation principle.

- **Open Design:** This principle requires that the design is not secret. Linux follows this principle by making its code open source.

- **Psychological Acceptability:** This principle states that it is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly. For some security mechanisms in Linux like SELinux, policies are written in such a specialized format, that only an expert can write a correct SELinux policy. Assessing this aspect of Linux is out of scope, as it would likely require user studies orthogonal to this thesis, but this thesis maintains ease of use by not changing any existing user interfaces.

## 1.2 Security Properties and Mechanisms in a Secure System

Although the above security design principles are crucial to creating a secure system, there are also some standard security properties and mechanisms that complement these principles. These properties and mechanism are commonly used in modern security literature [192], and tend to reflect more recent and specific techniques than the above security design principles.

- **Non-executable Data:** A control-flow-hijack exploit executes the attacker-specified malicious code. The Non-executable Data policy can prevent this attack by making data memory pages, like the stack or the heap, non-executable using the executable bit for memory pages in the hardware.
- **Memory Safety:** Memory safety is a property of some programming languages that prevents programmers from introducing certain types of bugs related to how the memory is used. All memory corruption exploits can be stopped by enforcing memory safety. However, both spatial and temporal errors must be prevented to achieve complete memory safety. Type-safe languages enforce the memory safety by checking object bounds at array accesses and using automatic garbage collection to prevent the programmer from explicitly destroying objects.
- **Code Integrity:** A Code Integrity policy ensures that program code is never overwritten. To achieve Code Integrity, all memory pages containing code are set to read-only using the hardware support present in all modern processors.
- **Data Integrity:** Data Integrity protects against control data hijacking and non-control data attacks, but not against covert channels that leak information. Although Data Integrity prevents data corruption, it only protects against invalid memory writes, not reads. Data Integrity is a weaker property than memory safety, because Data Integrity solutions enforce an approximation of spatial memory safety, and do not enforce temporal safety.
- **Data-flow Integrity:** Data-Flow Integrity (DFI) checks load/store instructions to detect the corruption of any data before it gets used. DFI restricts reads based on the last instruction that wrote the read location. For example, the policy ensures that the flag representing the supervisor mode was last written by the store instruction from the source code, and not by some attacker-controlled store. The

policy also ensures that the return address used by a return was last written by the corresponding call instruction.

- **Control-flow Integrity:** Control-flow Integrity (CFI) is a policy that detects and prevents attacks by restricting control-flow transfers to a limited set of allowed destinations. CFI statically determines the valid targets of not only calls, but also function returns, and thus enforces the resulting static control-flow graph. To prevent all control-flow hijacks, not only returns, but indirect calls and jumps have to be protected as well.

- **Address Space Randomization:** Address Space Layout Randomization (ASLR) is a memory address randomization technique, which randomly arranges the position of different code and data memory areas. The attacker cannot reliably divert control-flow if the address in the virtual memory space is not fixed. ASLR is the most comprehensive currently deployed protection against hijacking attacks.

- **Data Space Randomization:** Data Space Randomization (DSR) randomizes the representation of data stored in memory instead of the location of the data. It encrypts all variables, including pointers, and uses different keys. The code is instrumented to encrypt and decrypt variables when they are stored and loaded from memory. The protection of this mechanism is stronger, because encrypting all variables protects against control-flow hijacks, as well as data-only exploits.

- **Formal Verification:** Formal verification generally proves that an implementation satisfies a model, or specification, of the program's expected behavior. Assuming the specification is written to preclude insecure behavior (for some definition of secure), a formally-verified codebase will also be secure. Formal verification is a "holy grail" for developers; as E.W. Dijkstra pithily observed, "program testing can be used very effectively to show the presence of bugs but never to show their absence" [3]. Using formal verification, a developer can actually prove the absence of bugs in her program.

- **Penetration Testing:** Penetration testing is an empirical alternative to formal verification. The developers can recruit a team of expert attackers that test likely attack vectors. One of the successful examples of this type of security evaluation is the annual hacking contest Pwn2Own [34]. However, the cost of "bug bounties" can be high. For example, Google offered $1 million to hackers who can produce zero-day exploits against its Chrome browser [30].

- **Auditing:** Auditors can do a code review to check for any vulnerabilities, but is a labor-intensive process. Auditing ensures that proper security standards are met, which should reduce the existence of vulnerabilities in the code. While the goal of penetration testing is to find vulnerabilities, the goal of the auditor is to ensure safe practices are followed to prevent vulnerabilities.

- **System Hardening:** In addition to the above mechanisms, system hardening is a process of reducing the available attack surface. This includes using features like seccomp to prevent certain system calls from being accessed. More broadly, it involves only running the absolute minimum code needed for functionality, and removing unnecessary code or binaries from a system.

Often, figuring out how to apply the security design principles in a given context is not straightforward. Actually applying these principles is not as simple as typing some command or using a well-known tool; these principles require some creativity to even apply at all, and the manifestation looks different in different contexts. Additionally, depending on the context, enforcing the security mechanisms and properties can help adhere to one or more security design principles. For instance, memory safety can help achieve the principle of complete mediation, and following the economy of mechanism principle can help with auditing and formal verification.

## 1.3 Summary

This chapter showed how security was not a primary concern while designing legacy, complex systems. This chapter briefly described the security design principles and language or compiler-based security mechanisms. This chapter argued that retrofitting security design principles in the existing kernel design is a better solution than accepting the risk, encapsulating the risk, or rewriting the legacy code from scratch. This chapter briefly described how the thesis shows that encapsulating the risk is an incomplete solution, economically reduces privilege escalation vulnerabilities in Linux, and provides memory isolation for the containers feature of Linux. Finally, this chapter observes that applying the security design principles in a given context requires creativity, and the manifestation looks different in different contexts.

Chapter 2 observes how encapsulating the threats to the Linux kernel in a Virtual Machine is an imperfect solution. Chapter 3 discusses how to enforce the principle of least privilege in the Linux kernel by removing the need for running setuid-to-root binaries with elevated privileges. Chapter 4 demonstrates how we can economically retrofit security design principles and security mechanisms to a huge, complex codebase such as the Linux kernel. Finally, chapter 5 summarizes this thesis document.

# CHAPTER 2: ENCAPSULATING SECURITY THREATS USING VIRTUAL MACHINE INTROSPECTION

Virtualization has the potential to greatly improve system security by introducing a sensible layering—separating the policy enforcement mechanism from the component being secured. Most legacy OSes are both monolithic and burdened with a very wide attack surface. A legacy OS, such as Linux, executes all security modules in the same address space and with the same privilege level as the rest of the kernel [206]. When this is coupled with a porous attack surface, malicious software can often load code into the OS kernel that disables security measures, such as virus scanners and intrusion detection. As a result, users have generally lost confidence in the ability of the OS to enforce meaningful security properties. In public multi-tenant cloud computing, for instance, customers' computations are isolated using virtual machines rather than OS processes.

In contrast, hypervisors generally have a much narrower interface. Moreover, bare metal, or Type I [164], hypervisors generally have orders of magnitude fewer lines of code than a legacy OS. Table 2.1 summarizes the relative size of a representative legacy OS (Linux 3.13.5), and a representative bare-metal hypervisor (Xen 4.4), as well as comparing the number of reported exploits in both systems over the last 8 years. Perhaps unsurprisingly, the size of the code base and API complexity are strongly correlated with the number of reported vulnerabilities [185]. Thus, hypervisors are a much more appealing foundation for the trusted computing base of modern software systems.

This chapter focuses on systems that aim to assure the functionality required by applications using a legacy software stack, secured through techniques such as **virtual machine introspection (VMI)** [98]. A number of research projects observe that a sensitive application component, such as a random number generator or authentication module, requires little functionality, if any, from the OS, yet are vulnerable to failures of the OS [150, 151]. These projects are beyond the scope of this chapter, which instead focuses on systems that leverage virtualization to ensure security properties for applications that require legacy OS functionality.

| **Codebase** | **Lines of code** |
|---|---|
| Xen hypervisor 4.4 | 0.50 Million |
| Linux kernel 3.13.5 | 12.01 Million |
| **Codebase** | **No. of CVE** |
| Xen hypervisor | 24 |
| Linux kernel | 903 |

Table 2.1: Size and documented vulnerabilities of a representative bare-metal hypervisor (Xen) and legacy OS (Linux). Code sizes were calculated based on Xen 4.4 and Linux 3.13.5. CVEs were collected for all versions of these code bases over the period from 01/01/2006 to 03/03/2014.

VMI has become a relatively mature research topic, with over 100 projects. This chapter distills key design points from previous work on VMI—providing readers and system designers with a framework for evaluating design choices.

Moreover, we observe an unfortunate trend in the literature: many papers do not explicate their assumptions about the system, trusted computing base, or threat models. Although an attentive reader can often discern these facts, this trend can create confusion within the field. Thus, this survey carefully explicates the connection between certain design choices and the fundamental trust assumptions underlying these designs. One particularly salient observation is that all current solutions to the **semantic gap** problem [71] implicitly assume the guest OS is benign. Although this is a reasonable assumption in many contexts, it can become a stumbling block to the larger goal of reducing the size of the trusted computing base.

Finally, after identifying key design facets in previous work, this chapter identifies promising under-explored regions of the design space. The chapter discusses initial work in these areas, as well as the applicability of existing techniques and more challenging threat models.

The contributions and insights of this work are as follows:

- A thorough survey of research on VMI, and a distillation of the principal VMI design choices.
- An analysis of the relationship between design choices and implicit assumptions and trust. We observe that existing solutions to the semantic gap problem inherently trust the guest OS, often in direct contradiction to the underlying motivation for using VM introspection.
- The observation that the semantic gap problem has evolved into two separate issues: an engineering challenge and a security challenge. Existing solutions address the engineering challenge.
- Identifying a connection between techniques that protect memory and prevent attacks.

- Exploring the applicability of current techniques to new problems, such as removing the guest OS from the trusted computing base without removing OS functionality.
- Identifying additional points in the design space that are under-explored, such as hardware-support for mutual distrust among system layers and dynamic learning from an untrusted OS.

## 2.1 Background

The specific goals of VM introspection systems vary, but commonly include identifying if a malicious loadable kernel module, or **rootkit**, has compromised the integrity of the guest OS [163]; identifying malicious applications running on the system [142]; or ensuring the integrity or secrecy of sensitive files [113]. In these systems, a monitor tracks the behavior of each guest OS and either detects or prevents policy violations. Such a monitor may be placed in the hypervisor, a sibling VM, in the guest itself, or in the hardware, as illustrated in Figure 2.1. This process of looking into a VM is Virtual Machine Introspection (VMI).

A fundamental challenge to using VMI for security policy enforcement is that many desirable security policies are expressed in high-level, OS abstractions, such as files and processes, yet the hypervisor only has direct visibility into hardware-level abstractions, such as physical memory contents and hardware device operations. This disparity in abstractions is known as the **semantic gap**.

As an example of how the semantic gap creates challenges for introspection, consider how a hypervisor might go about listing the processes running in a guest OS. The hypervisor can access only hardware-level abstractions, such as the CPU registers and contents of guest memory pages. The hypervisor must identify specific regions of guest OS memory that include process descriptors, and interpret the raw bytes to reconstruct semantic information, such as the command line, user id, and scheduling priorities.

As a result of the semantic gap, much of the VMI development effort goes into reconstructing high-level semantic information from low-level sources. VMI tools attempt to reconstruct a range of information, including the set of running processes, sensitive file contents, and network sockets. For brevity, we limit this chapter to memory introspection, where the hypervisor draws inferences about guest behavior from the contents of memory and CPU registers. A range of work has also introspected disk contents [120, 193, 212] and network traffic [106, 124]; at this boundary, we limit discussion to in-memory data structures, such as those representing file metadata (`inode`) or a socket (`sk_buff`).

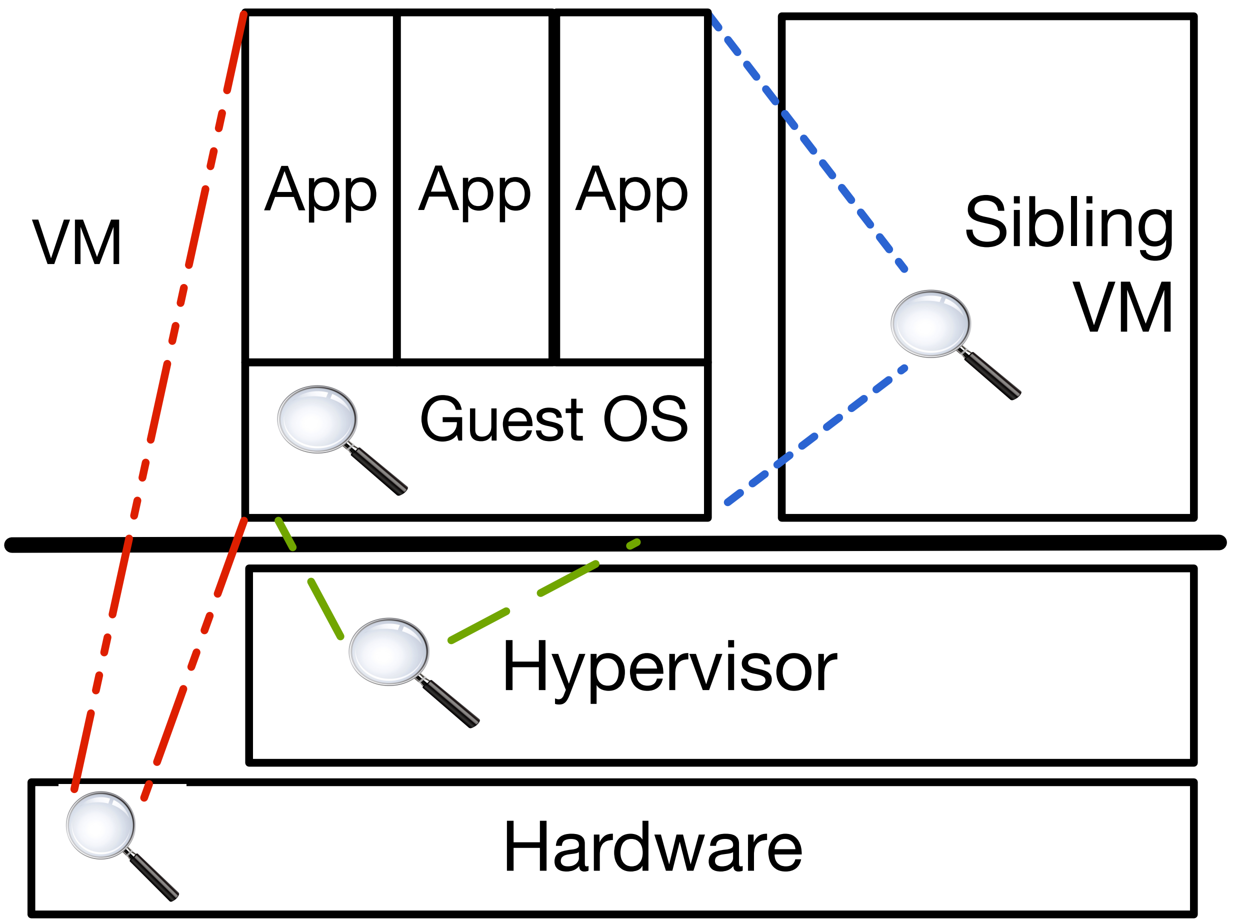


Figure 2.1: Monitor placement options in VMI: in a sibling VM, the hypervisor, in the guest OS itself, or in hardware. In-guest and hardware solutions require some assistance from the hypervisor.

As we discuss in the next section, many of these semantic reconstruction techniques rely on fragile assumptions or are best-effort. Unfortunately, errors in reconstructing semantic information can be exploited by malware to trick an introspection-based security monitor.

Continuing our example of listing processes in a guest OS, a typical introspection strategy would be to identify the definition of a process descriptor (e.g., a `task_struct` on Linux) from the source code, and then walk the list of runnable processes by following the global root of the process list `init_task`, overlaying this structure definition over the relevant memory addresses. This strategy faces a number of challenges. First, one must either assume all process descriptors are in this list—even in a compromised or malicious OS—or one must detect hidden processes, using techniques such as scanning all of guest memory looking for potential process descriptors or detecting inconsistencies between the currently loaded page tables and the purported process descriptor [121]. Hidden process detection [113] faces additional challenges, such as false positives from scanning memory during a critical section which temporarily violates some internal invariant the introspection tool is checking. In order to prevent the guest OS from using a hidden process descriptor, the introspection must identify all context switching code in the kernel, possibly including dynamically loaded code that manually context switches a hidden process. Finally, a rootkit might hide itself in a subtle and unexpected manner, such as loading itself as a thread in the address space of a benign system process, or placing its code in the memory image of a common library and scheduling itself by changing the address of a signal handling function.

These subtleties make robustly bridging the semantic gap quite a challenge. The next section organizes current strategies to solve this problem.

## 2.2 Bridges Across the Semantic Gap

Modern OSes are complex systems consisting of thousands of data structures, and many instances of each type. A typical running instance of the Linux kernel was found to have a core set of 29,488 data structure instances belonging to 231 different types that enable scheduling, memory management, and I/O [169]. Each of these structures consists of many fields. For instance, a *task_struct* in Linux 3.10 contains more than 50 fields [19], many of which are pointers to other structures. A key ingredient to any solution to the semantic gap problem is **reconstruction** of kernel data structures from memory contents.

This section begins with explaining techniques to reconstruct kernel data structures (2.2.1), followed by additional introspection techniques that do not directly reconstruct data structures (§2.2.2–2.2.3), and then techniques that assure the integrity of the kernel binary (§2.2.4). As the section explains each technique, it highlights the underlying trust assumption(s)—most commonly that the guest OS is benign. We will revisit these trust assumptions as we explain VMI attacks and defenses (§2.4). as well as discussing how one might adapt VMI to a stronger threat model where these assumptions do not hold (§2.5).

### 2.2.1 Learning and Reconstruction

Data structure reconstruction generally relies on a learn and search methodology. A **learning phase** is used to extract information relevant to data structures, generally a data structure **signature**. A signature can be used to identify and reconstruct data structure instances within kernel memory contents. Signatures are created using techniques such as expert knowledge, source analysis, or dynamic analysis—each described in this subsection (§2.2.1.1–2.2.1.3).

A second **search phase** identifies instances of the data structure. The two most common search strategies are to either linearly scan kernel memory or to traverse data structure pointers, starting with public symbols. Depending on the OS, public symbols may include debugging symbols or the dynamic linking tables exposed to loadable kernel modules. It is arguable which approach is more efficient, since many kernel data structures can have cyclic or invalid pointers, but may require traversing less total memory. However, the linear scan of kernel memory has the advantage that it is robust to "disconnected" structures or other attempts to obfuscate pointers. Both techniques can observe *transient* states when searching concurrently with OS operation, discussed further in §2.3.1.

Several linear scanning techniques limit the search space by introspecting on the kernel memory allocators—either by interpreting allocator data structures [113] or by placing debugging breakpoints on the allocator [169]. OS kernels commonly use a different slab or memory pool for each object type; this information can be used to further infer data structure types. An advantage of leveraging heap-internal information for search is more easily identifying transient data structures which have been freed but may be pointed to—a challenge for other search approaches. An inherent risk of this approach is missing data structures allocated in an unorthodox manner.

**Searching overheads.** In practice, searching for data structures in a kernel memory snapshot can take from tens of milliseconds [113] up to to several minutes [63]. Thus, most systems reduce overheads by searching periodically and asynchronously (§2.3.1). Periodic searches fundamentally limit these approaches to detecting compromises after the fact, rather than preventing policy violations. Moreover, these approaches can only reliably detect compromises that make persistent changes to a data structure. Transient malware can race between two searches of kernel memory.

The rest of this subsection describes the three major approaches to learning data structure signatures.

#### 2.2.1.1 Hand-crafted Data Structure Signatures

Introspection and forensic analysis tools initially used hand-crafted signatures, based on expert knowledge of the internal workings of an OS. For instance, such a tool might generate the list of running processes, similar to `ps`, by walking a global task list in Linux. Examples of this approach include Memparser [16], KNTLIST [12], GREPEXEC [6] and many others [2, 4, 5, 8, 15, 17, 18, 21, 22, 176, 177].

The most sophisticated frameworks for hand-crafted reconstruction use a wide range of subtle invariants and allow users to develop customized extensions. FACE/Ramparser [80] is a scanner that leverages invariants on values within a data structure, such as enumerated values and pointers that cannot be null. Ramparser can identify running processes (*task_struct*), open network sockets (*struct_sock*), in-kernel socket buffers (*sk_buff*), loaded kernel modules (struct *module*), and memory-mapped and open files for any given process (*vm_area_struct*). Similarly, Volatility [20] is a framework for developing forensics tools that analyze memory snapshots, with a focus on helping end-users to write extensions. Currently, Volatility includes tools that extract a list of running processes, open network sockets and network connections, DLLs loaded for each process, OS kernel modules, system call tables, and the contents of a given process's memory.

Hand-crafting signatures and data structure reconstruction tools creates an inherent limitation: each change to an OS kernel requires an expert to update the tools. For instance, a new version of the Linux kernel is released every 2–3 months; bugfix updates to a range of older kernels are released as frequently as every few weeks. Each of these releases may change a data structure layout or invariant. Similarly, different compilers or versions of the same compiler can change the layout of a data structure in memory, frustrating hand-written tools. Hand-written tools cannot keep pace with this release schedule and variety of OS kernels and compilers; thus, most introspection research has instead moved toward automated techniques.

#### 2.2.1.2 Source Code Analysis

Automated reconstruction tools may rely on source code analysis or debugging information to extract data structures definitions, as well as leverage sophisticated static analysis and source invariants to reduce false positives during the search phase. Examples of source code analysis tools include SigGraph [140], KOP [63], and MAS [82].

One basic approach to source analysis is to identify all kernel object types, and leverage points-to analysis to identify the graph of kernel object types. Kernel Object Pinpointer (KOP) [63] extended a fast aliasing analysis developed for non-security purposes [109], with several additional features, including: field-sensitivity, allowing KOP to differentiate accesses made to different fields within the same struct; context-sensitivity, differentiating different uses of `union` types and `void` pointers based on type information in code at the call sites; as well as inter-procedural and flow-insensitive analysis, rendering the analysis robust to conditional control flow, e.g., `if` statements. In applying the static analysis to a memory snapshot, KOP begins with global symbols and traverses all pointers in the identified data structures to generate a graph of kernel data structures.

A key challenge in creating this graph of data structures is that not all of the pointers in a data structure point to valid data. As a simple example, the Linux `dcache` uses deferred memory reclamation of a directory entry structure, called a `dentry`, in order to avoid synchronization with readers. When a `dentry` is on a to-be-freed list, it may point to memory that has already been freed and reallocated for another purpose; an implicit invariant is that these pointers will no longer be followed once the `dentry` is on this list. Unfortunately, these implicit invariants can thwart simple pointer traversal. MAS [82] addresses the issue of invalid pointers by extending the static analysis to incorporate value and memory alias checks.

Systems like MAS [82], KOP [63] and LiveDM [169] also improve the accuracy of type discovery by leveraging the fact that most OSes create object pools or slabs for each object type. Thus, if one knows which pages are assigned to each memory pool, one can reliably infer the type of any dynamically allocated object. We hasten to note that this assumption can be easily violated by a rootkit or malicious OS, either by the rootkit creating a custom allocator, or allocating objects of greater or equal size from a different pool and repurposing the memory. Thus, additional effort is required to detect unexpected memory allocation strategies.

SigGraph [140] contributed the idea that the graph structure of the pointers in a set of data structures can be used as a signature. As a simple example, the relationships of pointers among `task_struct` structures in Linux is fundamentally different than among `inode` structures. SigGraph represents graph signatures in a grammar where each symbol represents a pointer to a sub-structure. This signature grammar can be extended to encode arbitrary pointer graphs or encode sub-structures of interest. SigGraph is designed to work with a linear scan of memory, rather than relying on reachability from kernel symbols.

#### 2.2.1.3 Dynamic Learning

Rather than identifying code invariants from kernel source code, VMI based on dynamic analysis learns data structure invariants based on observing an OS instance [45, 88, 140].

By analogy to supervised machine learning, the VMI tool trains on a trusted OS instance, and then classifies the data structures of potentially untrusted OS instances. During the training phase, these systems often control the stimuli, by running programs that will manipulate a data structure of interest, or incorporating debugging symbols to more quickly discern which memory regions might include a structure of interest.

Tools such as Daikon [92] are used to generate constraints based on observed values in fields of a given data structure.

Several dynamic systems have created **robust signatures**, which are immune to malicious changes to live data structure instances [88]. More formally, a robust signature identifies any memory location that could be used as a given structure type without false negatives. A robust signature can have false positives. Robust signatures are constructed through fuzz testing during the training phase to identify invariants which, if violated, will crash the kernel [88, 140]. For instance, RSFKDS begins its training phase with a guest in a clean state, and then attempts to change different data structure fields. If the guest OS crashes, this value is used to generate a new constraint on the potential values of that field. The primary utility of robust signatures is detecting when a rootkit attempts to hide persistent data by modifying data structures in ways that the kernel doesn't expect. The key insight is that these attempts are only fruitful inasmuch as they do not crash the OS kernel. Thus, robust signatures leverage invariants an attacker cannot safely violate.

### 2.2.2 Code Implanting

A simpler approach to bridging the semantic gap is to simply inject code into the guest OS that reports semantic information back to the hypervisor. For example, Process implanting [104] implants and executes a monitoring process within a randomly-selected process already present in the VM. Any malicious agent inside the VM is unable to predict which guest process has been replaced and thus the injected code can run without detection. Rather than implant a complete process, SYRINGE [62] implants functions into the kernel, which can be called from the VM.

A challenge to implanting code is ensuring that the implanted code is not tampered with, actually executes, and that the guest OS components it uses report correct information. SIM [183] uses page table protections to isolate an implanted process's address space from the guest OS kernel. Section 2.2.4 discusses techniques to ensure the integrity of the OS kernel. Most of these implanting techniques ultimately rely on the guest kernel to faithfully represent information such as its own process tree to the injected code.

### 2.2.3 Process Outgrafting

In order to overcome the challenges with running a trusted process inside of an untrusted VM, process outgrafting [188] relocates a monitoring process from the monitored VM to a second, trusted VM. The trusted

VM has some visibility into the kernel memory of the monitored VM, allowing a VMI tools to access any kernel data structure without any direct interference from an adversary in the monitored VM.

Virtuoso [87] automatically generates introspection programs based on dynamic learning from a trusted VM, and then runs these tools in an outgrafted VM. Similarly, OSck [113] generates introspection tools from Linux source which execute in a monitoring VM with a read-only view of a monitored guest.

VMST [96] generalizes this approach by eliminating the need for dynamic analysis or customized tools; rather, a trusted, clean copy of the OS runs with a roughly copy-on-write view of the monitored guest. Monitoring applications, such as `ps`, simply execute in a complete OS environment on the monitoring VM; each system call executed actually reads state from the monitored VM. VMST has been extended with an out-of-VM shell with both execute and write capabilities [97], as well as accelerated by using memoization, trading some accuracy for performance [174]. This approach bridges the semantic gap by repurposing existing OS code.

The out-grafting approach has several open problems. First, if the monitoring VM treats kernel data as copy-on-write, the monitoring VM must be able to reconcile divergences in the kernel views. For example, each time the kernel accesses a file, the kernel may update the inode's `atime`. These atime updates will copy the kernel data, which must be discarded for future introspection or view of the file system will diverge. VMST does not address this problem, although it might be addressed by an expert identifying portions of the kernel which may safely diverge, or resetting the VM after an unsafe divergence. Similar to the limitations of hand-crafted introspection tools, each new OS variant may require hand-updates to divergent state; thus, automating divergence analysis is a useful topic for future work. Finally, this approach cannot handle policies that require visibility into data on disk—either files or swapped memory pages.

### 2.2.4 Kernel Executable Integrity

The introspection approaches described above assume that the executable kernel code does not change between creation of the introspection tools and monitoring the guest OS. Table 2.2 lists additional assumptions made by these techniques.

In order to uphold the assumption that the kernel has not changed, most hypervisor-based security systems must also prevent or limit the ability of the guest OS to modify its own executable code, e.g., by overwriting executable pages or loading modules. This subsection summarizes the major approaches to ensuring kernel binary integrity.

| **Technique** | **Assumptions** | **Monitor Placement** | **Systems** |
|---|---|---|---|
| **Hand-crafted data structure signatures** (Expert knowledge) | • Expert knowledge of OS internals for known kernel version<br>• Guest OS is not actively malicious | Sibling VM, hypervisor, or hardware | [2, 4, 5, 6, 8, 12, 15, 16, 17, 18, 21, 22, 176, 177] |
| **Automated learning and reconstruction** (Source analysis or offline training) | • Benign copy of OS for training<br>• OS will behave similarly during learning phase and monitoring<br>• Security-sensitive invariants can be automatically learned<br>• Attacks will persist long enough for periodic scans | Sibling VM, hypervisor, or hardware | [45, 63, 82, 88, 140, 169] |
| **Code implanting** (VMM protects monitoring agent inside guest OS) | • Malicious guest schedules monitoring tool and reports information accurately | Guest with hypervisor protection | [62, 104, 183] |
| **Process outgrafting** (Reuse monitoring tools from sibling VM with shared kernel memory) | • Live, benign copy of OS behaves identically to monitored OS | Sibling VM | [87, 96, 97, 188] |
| **Kernel executable integrity** (Protect executable pages and other code hooks) | • Initial benign version of monitored OS<br>• Administrator can allow-list safe modules | Hypervisor | [142, 160, 171, 181, 202] |

Table 2.2: VMI techniques, monitor placement (as illustrated in Figure 2.1, and their underlying trust assumptions.

#### 2.2.4.1 The (Write $\oplus$ Execute) Principle

The W $\oplus$ X principle prevents attacks that write the text segment by enforcing a property where all the pages are either writable or executable, but not both at the same time. For instance, SecVisor [181] and NICKLE [171] are hypervisors that enforce the W $\oplus$ X principle by setting page table permissions on kernel memory. SecVisor will only set executable permission on kernel code and loadable modules that are approved by an administrator, and prevent modification of this code.

Although non-executable (NX) page table bits are ubiquitous on modern x86 systems, lack of NX support complicated the designs of early systems that enforced the W $\oplus$ X principle. Similarly, compilers can mix code and data within the same page, although security-conscious developers can also restrict this with linker directives.

#### 2.2.4.2 Allow-listing Code

As discussed above, SecVisor and NICKLE policies require a notion of approved code, which is represented by an allow-list of code hashes created by the administrator.

Patagonix [142] extends this property to application binaries, without the need to understand the structure of processes and memory maps. Patagonix leverages no-execute page table support to receive a trap the first time data from a page of memory is loaded into the CPU instruction cache. These pages are then compared against a database of allow-listed application binary pages.

Although allow-listing code can prevent loading unknown modules which are the most likely to be malicious, the approach is limited by the administrator's ability to judge whether a driver or OS kernel is malicious *a priori*.

#### 2.2.4.3 Object Code Hooks

A practical limitation of the W $\oplus$ X principle is that many kernels place function pointers in data objects that must be writable. These function pointers are used to implement a crude form of object orientation. For instance, the Linux VFS allows a low-level file system to extend generic routines for operations such as reading a file or following a symbolic link.

Lares [160] implemented a simple page-protection mechanism on kernel object hooks, but incurred substantial performance penalties because these executable pointers are in the same page as fields which the guest kernel must be able to write, such as the file size and modification time. HookSafe [202] addresses this problem by modifying OS kernel code to relocate all hooks to a read-only, shadow memory space. All code that calls a hook must also check that the requested hook is in the shadow memory space, and a policy must approve before a code is added to the hook section. The hook redirection and checking code is in the kernel's binary text, and is read-only. HookSafe identifies locations where hooks are called through dynamic learning (§2.2.1); this could likely be extended with static analysis for more complete coverage.

Ultimately, these techniques are approximating the larger property of ensuring control flow integrity (CFI) of the kernel [35]. Ensuring CFI is a broad problem with a range of techniques. For instance, Program Shepherding [127] protects the integrity of implanted functions [62] (§2.2.2), using a machine code interpreter to monitor all control transfers and guarantee that each transfer satisfies a given security policy. Discovering efficient CFI mechanisms is a relevant, but complimentary problem to VMI.

## 2.3 Prevention vs. Detection

Some introspection tools prevent certain security policy violations, such as execution of unauthorized code, whereas others only detect a compromise after the fact. Clearly, prevention is a more desirable goal, but many designs accept detection to lower performance overheads. This section discusses how certain design choices fundamentally dictate whether a system can provide detection or prevention.

**Prevention** requires a mechanism to identify and interpose on a low-level operation within a VM that violates a system security policy. Certain goals map naturally onto hardware mechanisms, such as page protections on kernel code or hooks [160, 171, 181, 202]. Other goals, such as upholding data structure invariants the kernel code relies upon, are open questions.

As a result, violations of more challenging properties are currently only **detected** after the fact by VMI tools [45, 86, 87, 96, 113, 140, 142, 162, 163, 169, 174, 183]. In general, there is a strong connection between approaches that periodically search memory and detection. Periodic searching is a good fit for malware that persistently modifies data structures, but can miss transient modifications. To convert these approaches to prevention techniques would require interposing on every store, which is prohibitively expensive. Moreover, because some invariants span multiple writes, even this strawman approach would likely yield false negatives without even deeper analysis of the code behavior.

Current detection systems usually just power off a compromised VM and alert an administrator. Several research projects identify how systems can recover from an intrusion or other security violation [65, 89, 125, 126]. In general, general-purpose solutions either incur relatively high overheads to track update dependencies (35% for the most recent general-purpose, single-machine recovery system [126]), or leverage application-specific properties. Improving performance and generality of recovery systems is an important direction for future work.

### 2.3.1 Asynchronous vs. Synchronous Mechanisms

**Synchronous** mechanisms mediate guest operations inline to prevent security policy violations, or receive very low latency notification of changes. All prevention systems we surveyed [160, 171, 181, 183, 202] use synchronous mechanisms, such as page protection or code implanting. Several low-latency detection systems use customized hardware, discussed further in §2.3.2. A few systems also use synchronous mechanisms on commodity hardware for detection [132, 142, 169], but could likely lower their overheads with an asynchronous mechanism.

**Asynchronous** mechanisms execute concurrently with a running guest and inspect its memory. These systems generally introspect into a snapshot of memory [45, 86, 140, 162] or a read-only or copy-on-write view of guest memory [87, 96, 113, 118, 163, 174]. All surveyed asynchronous systems detect rootkits after infection through passive monitoring.

On one hand, the synchronous systems gain a vantage point over their counterparts against transient attacks but increase the overhead for the guest OS being protected. On the other hand, asynchronous systems introduce lower monitoring overhead but miss cleverly built transient attacks; asynchronous systems are also limited due to the inherent race condition between the attacker and the detection cycle.

Synchronous and asynchronous mechanisms make fundamental trade-offs across the performance, frequency of policy-relevant events, risk, and assumptions about the behavior of the system. Synchronous mechanisms tend to be more expensive, and are generally only effective when the monitored events are infrequent, such as a change in the access pattern to a given virtual page. The cost of an asynchronous search of memory can also be quite high (ranging from milliseconds [113] to minutes [63]), but the frequency can be adjusted to an acceptable rate—trading risk for performance. Both synchronous and asynchronous systems make potentially fragile assumptions about the system to improve performance, such as knowing all hook locations or assuming all objects of a given type are allocated from the same slab. These risks could be reduced in future work by identifying low-frequency events that indicate a policy violation, are monitorable without making fragile assumptions about the system, and introduce little-to-no overheads in the common case.

A final issue with executing introspection concurrently with the execution of an OS is false positives arising because of **transient states**. In general, an OS may violate its own invariants temporarily while executing inside of a critical section. A correct OS will, of course, restore the invariants before exiting the critical section. If an introspection agent searches memory during a kernel critical section, it may observe benign violations of these invariants, which will resolve quickly. Approaches to this problem include simply looking for repeated violations of an invariant (leaving the system vulnerable to race conditions with an attacker), or only taking memory snapshots when the OS cannot be in any critical sections (e.g., by preempting each CPU while out of the guest kernel).

VMI systems face fundamental trade-offs between performance and risk, often making fragile assumptions about the guest OS.

### 2.3.2 Hardware-assisted Introspection

Several research prototypes have employed customized hardware for introspection [132, 144, 157], or applied existing hardware in novel ways [42, 162, 200]. The primary division within the current design space of hardware-assisted introspection is whether the introspection tool uses memory snapshots or snoops on a bus. Snooping can monitor memory regions at finer granularity than page protections, reducing overheads.

#### 2.3.2.1 Snapshotting

One strategy for hardware-assisted introspection is using a PCI device to take RAM snapshots, which are sent to a second machine for introspection (monitored and monitor, respectively). For instance, Copilot [162] adds an Intel StrongARM EBSA-285 Evaluation Board to the monitored machine's PCI bus. The PCI device on the monitored machine uses DMA requests to retrieve a snapshot of host RAM, which is sent to the monitor machine upon request over an independent communication link. The monitor periodically requests snapshots and primarily checks that the hash of the kernel binary text and certain code pointers, such as the system call table, have not changed from known-good values.

Unfortunately, a memory snapshot alone isn't sufficient to robustly reconstruct and interpret a snapshot. Of particular importance is the value of the `cr3` register, which gives the physical address of the root of the page tables. Without this CPU register value, one cannot reliably reconstruct the virtual memory mapping. Similarly, a system can block access to regions of physical memory using an IOMMU [39, 58].

HyperCheck [200] augments physical memory snapshots with the contents of the `cr3` register, using the CPU System Management Mode (SMM) [9]. SMM is an x86 CPU mode designed primarily for firmware, power management, and other system functions. SMM has the advantage of protecting the introspection code from the running system as well as giving access to the CPU registers, but must also preempt the system while running (i.e., this is a synchronous mechanism). The processor enters SMM when the SMM interrupt pin (SMI) is raised, generally by the Advanced Programmable Interrupt Controller (APIC). The hypervisor is required to create SMI interrupts to switch the CPU to SMM mode. Upon entering SMM, the processor will launch a program stored in system management RAM (SMRAM). SMRAM is either a locked region of system DRAM, or a separate chip, and ranges in size from 32 KB to 4 GB [9] Outside of SMM, SMRAM may not be read or written. Within SMM, the integrity checking agent has unfettered access to all RAM and devices, and is not limited by a IOMMU or other attacks discussed previously. Unfortunately, SMM mode also has the limitation that Windows and Linux will hang if any software spends too much time in SMM, bounding the time introspection code can take.

HyperSentry [42] further refines this model by triggering an SMI handler from an Intelligent Platform Management Interface (IPMI) device. IPMI devices generally execute system management code, such as powering the system on or off over the network, on a device hidden from the system software.

A limitation of any SMM-based solution, including the ones above, is that a malicious hypervisor could block SMI interrupts on every CPU in the APIC, effectively starving the introspection tool. For VMI, trusting the hypervisor is not a problem, but the hardware isolation from the hypervisor is incomplete.

Each of these systems focus on measuring the integrity of system software—e.g., checking that the executable pages have a known-good hash value. At least in SMM mode, more computationally expensive introspection may be impractical. Because all of these operations operate on periodic snapshots, which may visibly perturb memory access timings, a concern is that an adversary could predict the snapshotting interval and race with the introspection agent. In order to ensure that transient attacks cannot race with the snapshot creation, more recent systems have turned to snooping, which can continuously monitor memory changes.

#### 2.3.2.2 Snooping

A number of recent projects have developed prototype security hardware that snoops on the memory bus [132, 144, 157]. These systems have the useful function of efficiently monitoring writes to sensitive code regions; unlike page protections, snooping systems can monitor writes at the finer granularity of cache lines, reducing the number of needless checks triggered by memory accesses adjacent to the structure being monitored. These systems can also detect updates to memory from a malicious device or driver by DMA, which page-level protections cannot detect.

Although most prototypes have focused on detecting modifications to the kernel binary itself, KI-Mon also watches for updates to object hooks [132], and there is likely no fundamental reason other solutions could not implement this.

Because these snooping devices aim to be very lightweight, they cannot then check data structure invariants or code integrity, but instead signal a companion snapshotting device (as discussed above) to do these checks. However, a specific memory event triggering asynchronous checks is a clear improvement over periodic snapshots, in both efficiency and risk of races with the attacker. A small complication with snooping-triggered introspection is that invariants often span multiple cache lines, such as `next.prev == next` in a doubly-linked list. If an invariant check is triggered on the first write in a critical section, the system will see many false positives. KI-Mon addresses this by waiting until the system quiesces. However, quiescence is an OS-kernel-specific convention, and can be violated by an adversarial kernel.

We note that these systems do not use commodity hardware, but are implemented in simulators or FPGAs. Section 2.5.2 argues that this is a promising area of research that deserves more attention, but more work has to be done to demonstrate the utility of the approach before it will be widely available. Similarly, these systems have initially focused on attack detection, but it would be interesting to extend these systems to recovering from a detected attack.

| Snooping is useful for finer-grained memory monitoring. |
|---|

### 2.3.3 Memory Protection: A Necessary Property for Prevention

We end this section by observing that all prevention systems employ some form of memory protection to synchronously interpose on sensitive data writes. For example, HookSafe [202] and Lares [160] use memory

| **Attack** | **Defense** | **Trust Assumption** |
|---|---|---|
| Write text Segment | Hypervisor-enforced W $\oplus$ X. | Initial text segment benign. |
| KOH (code and hooks) | Memory protect hooks from text section modification, or allow-list loadable modules. | Pristine initial OS copy and administrator's ability to discern trustworthy kernel modules. |
| DKOM (heap) | Identify data structure invariants, detect violations by scanning memory snapshots. | • Guest kernel exhibits only desirable behavior during training, or source is trustworthy.<br>• All security-relevant data structure invariants can be identified a priori.<br>• All malware will leave persistent modifications that violate an invariant.<br>• All invariants can be checked in a single search.<br>• Attackers cannot win races with the monitor. |
| DKSM | Prevent Bootstrapping through KOH or ROP. | OS is benign; behaves identically during training and classification. |

Table 2.3: VMI attacks, defenses, and underlying trust assumptions.

protection to guard against unexpected updates to function pointers. In contrast, it isn't clear how to convert an asynchronous memory search from a detection into a prevention tool. The most likely candidate is with selective, fine-grained hardware memory bus snooping, described above. Thus, if attack prevention is a more desirable goal than detection after-the-fact, the community should focus more effort on discovering lightweight, synchronous monitoring mechanisms.

All current prevention systems rely on synchronous memory protection.

## 2.4 Attacks, Defense, and Trust

This section explains the three major classes of attacks against VMI, known defenses against those attacks, and explains how these attacks relate to an underlying trust placed in the guest OS. These issues are summarized in Table 2.3.

### 2.4.1 Kernel Object Hooking

A Kernel Object Hooking (KOH) attack [11] attempts to modify function pointers (hooks) located in the kernel text or data sections. An attacker overwrites a function pointer with the address of a function provided by the attacker, which will then allow the attacker to interpose on a desired set of kernel operations. In some sense, Linux Security Modules provide similar hooks for security enhancements [206]; the primary difference is that KOH repurposes other hooks used for purposes such as implementing an extensible virtual file system (VFS) model. The defenses against KOH attacks generally depend on whether the hook is located in the text or data segment.

#### 2.4.1.1 Text Section Hooks

The primary text section hooks are the system call table and interrupt descriptor table. For instance, an attacker could interpose on all file `open` calls simply by replacing the pointer to the `sys_open()` function in the system call table.

In older OSes, these hooks were in the data segment despite not being dynamically changed by most OSes. In order to prevent malware from overwriting these hooks, most kernels now place these hooks in the read-only text segment. As discussed in §2.2.4.1, a sufficient defense is hypervisor-imposed, page-level Write $\oplus$ Execute permissions.

#### 2.4.1.2 Data Section Hooks

Kernel data section hooks are more difficult to protect than text section hooks. Data section hooks place function pointers in objects, facilitating extensibility. For instance, Linux implements a range of different socket types behind a generic API; each instantiation overrides certain hooks in the file descriptor for a given socket handle.

The fundamental challenge is that, although these hooks generally do not change during the lifetime of the object, they are often placed in the same page or even cache line with fields that do change. Because most kernels mix hooks which should be immutable with changing data, most hardware-based protection mechanisms are thwarted.

In practice, these hooks are very useful for rootkits to hide themselves from anti-malware tools inside the VM. For instance, the Adore-ng [76] rootkit overrides the `lookup()` and `readdir()` functions on the

`/proc` file system directory. Process listing utilities work by reading the sub-directories for each running process under `/proc`; a rootkit that overrides these functions can filter itself from the `readdir()` system call issued by `ps`.

In order to defend against such attacks, the function pointers need to be protected from modification once initialized. Because of the high-cost of moderating all writes to these data structures, most defenses either move the hooks to different locations which can be write-protected [202], or augment hooks in the kernel with checks against an allow-list of trusted functions [163].

**Trust.** Protecting the kernel code from unexpected modifications at runtime is clearly sensible. Underlying these defenses is the assumption that the kernel is initially trusted, but may be compromised later. The more subtle point, however, is that all of the VMI tools discussed in §2.2 assume that the kernel text will not change. Thus, preventing text section modification is effectively a prerequisite for current VMI techniques.

Defenses against KOH on data hooks generally posit trust in the ability of an administrator to correctly identify trustworthy and untrustworthy kernel modules. Dynamic kernel module loading requires changing the kernel control flow by writing to the code hooks. As explained in Section 2.2.4, KOH defenses [142, 171, 181] assume that kernel modules are benign in order to provide some meaningful protections without solving the significantly harder problem of kernel control flow integrity in the presence of untrusted modules.

KOH defenses generally assume benign kernel modules.

Finally, we note that some published solutions to the KOH data section problem are based on best-effort dynamic analysis, which can miss hooks that are not exercised. There is no fundamental reason this analysis should be dynamic, other than the unavailability of source code. In fact, some systems do use static analysis on the source code to identify code hooks [113], which can identify all possible data section hooks.

### 2.4.2 Dynamic Kernel Object Manipulation

Manipulating the kernel text and code hooks are the easiest attack vector against VMI; once KOH defenses were developed, attackers turned their attention to attacks on the kernel heap. Dynamic Kernel Object Manipulation (DKOM) [59] attacks modify the kernel heap through a loaded module or an application accessing `/dev/mem` or `/proc/kcore` on Linux. DKOM attacks only modify data values, and thus are distinct from modifying the control flow through function hooks (KOH).

A DKOM attack works by violating latent assumptions in unmodified kernel code. A classic example of a DKOM attack is hiding a malicious process from a process listing tools, such as `ps`. The Linux kernel tracks processes in two separate data structures: a linked list for process listing and a tree for scheduling. A rootkit can hide malicious processes by taking the process out of the linked list, but leaving the malicious process in the scheduler tree. The interesting property is that loading a module can be sufficient to alter the behavior of unrelated, unmodified kernel code, because any module can write to any kernel data structure.

DKOM attacks are hard to prevent because they are a metaphorical needle in a haystack of expected kernel heap writes. As a result, most practical defenses attempt to identify data structure invariants — either by hand, static, or dynamic analysis — and then detect data structure invariant violations asynchronously. Because an attacker can create objects from any memory, not just the kernel heap allocator, data structure detection is also a salient issue for detecting DKOM attacks (§2.2.1). Thus, a robust, asynchronous DKOM detector must search all guest memory, increasing overheads, and tolerate attempts to obfuscate a structure.

**Trust.** DKOM defenses introduce additional trust in the guest beyond a KOH defense, and make several assumptions which an attacker can could be violated by an attacker. Most DKOM defenses work by identifying security-related data structure invariants. Because it is difficult for the defender to ever have confidence that all security-relevant invariants have been identified, this approach will generally be best-effort and reactive in nature. Deeper source analysis tools could yield more comprehensive invariant results, but more research is needed on this topic. Many papers on the topic focus on a few troublesome data structures, such as the `task_struct`, yet Linux has several hundred data structure types. It is unclear whether any automated analysis will scale to the number of hiding places afforded to rootkits by monolithic kernels, or whether detection tools will always be one step behind attackers. That said, even a best-effort defense has value in making rootkits harder to write.

Another problematic assumption is that all security-sensitive fields of kernel data structures have invariants that can be easily checked in a memory snapshot. For instance, one might assume that any outgoing packets come from a socket that appears in the output of a tool such as netstat (or a VMI-based equivalent). Yet a malicious Linux kernel module could copy packets from the heap of an application to the outgoing IP queue—a point in the networking stack which doesn't maintain any information about the originating socket or process. Thus, memory snapshots alone couldn't easily identify an inconsistency between outgoing packets and open sockets, especially if the packet could have been sent by a different process, such as a process with an open raw socket. Although the problem in this example could be mitigated with continuous monitoring, such monitoring would substantially increase runtime overheads; in contrast, most DKOM defenses rely on infrequent scanning to minimize overheads. In this example, the data structure invariant spans a sequence of operations, which can't be captured with one snapshot.

A single snapshot cannot capture data structure invariants that span multiple operations.

Third, DKOM defenses *cement trust that the guest kernel is benign*. These defenses train data structure classifiers on a clean kernel instance or derive the classifiers from source code, which is assumed to only demonstrate desirable behavior during the training phase. This approach is similar to using symbolic execution to verify that a cryptographic client's message sequence is consistent with its known implementation [74]. Although we hasten to note that this assumption may be generally reasonable, it is not beyond question that an OS vendor might include a backdoor that such a classifier would learn to treat as expected behavior.

In order to ensure that the guest kernel is benign, DKOM defenses generally posit a KOH defense. Learning code invariants is of little use when an attacker can effectively replace the code. The interesting contrast between KOH and DKOM defenses is that DKOM defenses can detect invalid data modifications *even in the presence of an untrustworthy module*, whereas common KOH defenses rely on module allow-listing. Thus, if a DKOM defense intends to tolerate untrusted modules, it must build on a KOH defense that is robust to untrusted modules as well, which may require substantially stronger control flow integrity protection.

KOH defenses are a building block for DKOM defenses, but often make different trust assumptions about modules.

Finally, these detection systems explicitly assume *malware will leave persistent, detectable modifications* and implicitly assume *malware cannot win races with the detector*. DKOM detectors rely on invariant violations being present in the view of memory they analyze—either a snapshot or a concurrent search using a read-only view of memory. Because DKOM detectors run in increments of seconds, short-lived malware can easily evade detection. Even for persistent rootkits, a reasonably strong adversary may also have access to similar data structure classifiers and aggressively search for invariants missed by the classifier.

If a rootkit can reliably predict when a DKOM detector will view kernel memory, the rootkit has the opportunity to temporarily repair data structure invariants—racing with the detector. Reading a substantial portion of guest memory can be is highly disruptive to cache timings—stalling subsequent writes on coherence misses. Similarly, solutions based on preempting the guest OS will leave telltale "lost ticks" on the system clock. Even proposed hardware solutions can be probed by making benign writes to potentially sensitive addresses and then observing disruptions to unrelated I/O timings. Given the long history of TOCTTOU and other concurrency-based attacks [60, 209], combined with a likely timing channel induced by the search mechanism and recent successes exploiting VM-level side channels [211], the risk of an attacker successfully racing with a detector is concerning. More recent works show vulnerability of SGX to several types of side channel attacks, such as traditional cache timing and page table side channel attacks that reveal page-level memory accesses [55, 103, 179, 198, 201, 208], as well as speculative attacks [68, 129] that use the side channels as a way of retrieving information.

DKOM defenses are potentially vulnerable to race conditions within their threat model.

### 2.4.3 Direct Kernel Structure Manipulation

Direct Kernel Structure Manipulation (DKSM) attacks [44] change the interpretation of a data structure between training a VMI tool and its application to classify memory regions into data structures. Simple examples of a DKSM attack include swapping two fields within a data structure or padding the structure with garbage fields so that relative offsets differ from the expectation of the VMI tool.

Because most VMI tools assume a benign kernel, a successful DKSM attack hinges on changing kernel control flow. DKSM attack can be broken down to 2 attack vectors: 1) change in-memory kernel code and 2) compile malicious code into the kernel. In order to change the in-memory kernel code, DKSM needs to manipulate control flow. If KOH attacks cannot be used to bootstrap the DKSM attack, the DKSM attacker needs to employ return-oriented programming [143]. As discussed above, a number of successful countermeasures for KOH attacks have been developed, as have effective countermeasures to return-oriented programming, including G-Free [159], "Return-Less" kernels [137], and STIR [203].

**Trust.** DKSM is somewhat of an oddity in the literature because it is effectively precluded by a generous threat model. However, a realistic threat model might allow an adversarial OS to demonstrate different behavior during the data structure training and classification phases—analogous to "split-personality" malware that behaves differently when it detects that it is under analysis.

DKSM is a reasonable concern obviated by generous threat models.

### 2.4.4 The Semantic Gap is Really Two Problems

Under a stronger threat model, the DKSM attack effectively leverages the semantic gap to thwart security measures. Under DKSM, a malicious OS actively misleads VMI tools in order to violate a security policy.

In the literature on VM introspection, the semantic gap problem evolved to refer to two distinct issues: (1) the engineering challenges of generating introspection tools, possibly without source code [87, 96, 174], and (2) the ability of a malicious or compromised OS to exploit fragile assumptions underlying many introspection designs in order to evade a security measure [113, 140, 169, 181, 202]. These assumptions include:

- Trusting that the guest OS is benign during the training phase, and will not behave differently under monitoring.
- All security-sensitive invariants and hooks can be automatically learned.
- Attacks will persist long enough to be detected by periodic searches.
- Administrators can allow-list trustworthy kernel modules.

Most papers on introspection focus on the first problem, which has arguably been solved [87, 96, 174], yet interesting attacks leverage the second issue, which is still an open problem, as is reliable introspection under stronger threat models.

Unfortunately, before this work was published, the literature do not clearly distinguish these problem variations, and only a close reading indicate which one a given paper is addressing. This confusion is only exacerbated when one attempts to place these papers next to each other in the context of attacks and defenses. That said, we do believe that the overall path of starting with a weak attacker and iteratively strengthening the threat model is a pragmatic approach to research in this area; the issue is ambiguous nomenclature.

We therefore suggest a clearer nomenclature for the two sub-problems: the **weak** and **strong** semantic gap problems. The weak semantic gap is the largely solved engineering challenge of generating VMI tools, and the strong semantic gap refers to the challenge of defending against an adversarial, untrusted guest OS. A solution to the open strong semantic gap problem would not make any assumptions about the guest OS being benign during a training phase or accept inferences from guest source code as reliable without runtime validation. The strong semantic gap problem is, to our knowledge, unsolved, and the ability to review future work in this space relies on clearer delineation of the level of trust placed in the guest OS. A solution to the strong semantic gap problem would also prevent or detect DKSM attacks.

| App | Guest OS | Hypervisor | Challenge | Solutions |
|---|---|---|---|---|
|  | $\surd$ | $\surd$ | Weak Semantic Gap | Layered Security, VMI. Incrementally reduce trust in the guest OS. |
|  |  | $\surd$ | Strong Semantic Gap | Difficult to solve. Need techniques that can learn from untrusted sources and detect inconsistencies during VMI. |
| $\surd$ |  | $\surd$ | Untrusted guest OS | Paraverification. Application trust bridges the semantic gap. |
| $\surd$ | $\surd$ |  | Untrusted cloud hypervisor | Support from trusted hardware like SGX [10, 152]. |
| $\surd$ |  |  | Untrusted guest OS and hypervisor | Fine grained support from trusted hardware needed. |

Table 2.4: Trust Models. ($\surd$ indicates the layers that are trusted.)

The weak semantic gap is a solved engineering problem. The strong semantic gap is an open security problem.

Thenceforward, new VMI literature [83, 133, 166, 210] adopted the nomenclature of weak and strong semantic gap, and new research [84, 115] is making progress to solve the strong semantic gap.

## 2.5 Toward an Untrusted OS

Any solution to the strong semantic gap problem may need to remove assumptions that the guest OS can be trusted to help train an introspection tool. As illustrated in Section 2.2, most existing introspection tools rely on the assumption that the guest OS begins in a benign state and its source code or initial state can be trusted. Over time, several designs have reduced the degree to which they rely on the guest OS. It is not clear, however, that continued iterative refinement will converge on techniques that eliminate trust in the guest.

Table 2.4 illustrates the space of reasonable trust models in virtualization-based security. Although a lot of effort in VMI has gone into the first row (the weak semantic gap), the community should focus on new directions likely to bridge the strong semantic gap (second row), as well as adopt useful techniques from research into the other rows.

This section identifies promising approaches to the strong semantic gap, based on insights from the literature.

### 2.5.1 Paraverification

Many VMI systems have the implicit design goal of working with an unmodified OS, or limiting modifications to the module loader and hooks. The goal of introspecting on an unmodified guest OS often induces trust in the guest OS to simplify this difficult problem. Specifically, most VMI tools assume the guest OS is not actively malicious and adheres to the behavior exhibited during the learning phase.

This subsection observes that, rather than relax the threat model for VMI, relaxing the requirement of an unmodified OS may be a more useful stepping stone toward an untrusted OS. By analogy, although initial hypervisors went through heroic efforts to virtualize unmodified legacy OSes on an ISA very unsuitable for virtualization [57], most modern OSes now implement paravirtualization support [46]. Essentially, paravirtualization makes small modifications to the guest OS that eliminate the most onerous features to emulate. For instance, Xen allowed the guest OS to observe that there were inaccessible physical pages, substantially reducing the overheads of virtualizing physical memory. The reason paravirtualization was a success is that it was easy to adopt, introduced little or no overheads when the system executes on bare metal, and dramatically improved performance in a VM.

Thus, we expect that light modifications to a guest OS to aid in introspection could be a promising direction. Specifically, we observe that the VirtualGhost [81] and InkTag [114] system introduced the idea of *paraverification*, in which the guest OS provides the hypervisor with evidence that it is servicing an application's request correctly. The evidence offered by the guest OS is easily checked by the hypervisor without trusting the guest OS. For instance, a trusted application may request a memory mapping of a file, and, in addition to issuing an `mmap` system call, also reports the request to the hypervisor. When the OS modifies the application's page tables to implement the `mmap` system call, the OS also notifies the hypervisor that this modification is in response to a particular application request. The hypervisor can then do an end-to-end check that (1) the page table changes are applied to an appropriate region of the application's virtual memory, (2) that the CPU register values used to return to the application are sensible, and (3) that the contents of these pages match the expected values read from disk, using additional metadata storing hashes of file contents.

We hasten to note that the goals of InkTag and Virtual Ghost are different from VMI—ensuring a trusted application can safely use functionality from a malicious OS. This problem has also been explored in a number of other papers [72, 139]. Moreover, InkTag leverages the trusted application to bridge the semantic gap—a strategy that would not be suitable for the types of problems VMI aims to solve. Nonetheless, forcing an untrusted OS to aid in its own introspection could be fruitful if the techniques were simple enough to adopt.

Rather than relaxing the threat model for VMI, relax strict limits on guest modifications.

### 2.5.2 Hardware Support for Security

As we observe in §2.3.3, memory protection or other synchronous notification mechanisms appear to be a requirement to move from detection to prevention. Unfortunately, the coarseness of mechanisms in commodity hardware introduce substantial overheads. §2.3.2 summarizes recent work on memory monitoring at cache line granularity—a valuable approach meriting further research.

An interesting direction taken by Intel is developing a mutual distrust model for hardware memory protection, called Software Guard Extensions (SGX) [40, 112, 117, 152]. SGX allows an OS or hypervisor to manage virtual-to-physical OS mappings for an application, but the lower-level software cannot access memory contents. SGX provides memory isolation of a trusted application from an untrustworthy software stack. Similar memory isolation has been provided by several software-only systems [72, 114], but at a substantial performance cost attributable to frequent traps to a trusted hypervisor. Finally, we note that in order for an application to safely *use* system calls on an untrusted OS, a number of other problems must be addressed [66, 114].

In the context of introspection or the strong semantic gap, hardware like SGX can also be useful for creating a finer-grained protection domain for code implanted in the guest OS 2.2.2. However, we also note that SGX introduces the problem of malware getting in the trusted execution environment and hiding from the OS [178, 179, 180]. More fine-grained memory protection and monitoring tools are needed from hardware manufacturers to help solve the strong semantic gap.

Fine-grained memory protection and monitoring hardware can reduce overheads and trust.

### 2.5.3 Reconstruction from Untrusted Sources

Current tools that automatically learn data structure signatures assume the OS will behave similarly during training and classification (§2.4.2). Among the assumptions made in current VMI tools, this is one that potentially has the best chance of being incrementally removed. For example, one approach might train the VMI classifiers on the live OS, and continue incrementally training as the guest OS runs.

Another approach would be to detect inconsistencies between the training and classification stages of VMI. By analogy, distributed fault tolerance systems are often built around the abstraction of a **proof of misbehavior**, where a faulty participant in the protocol generates signed messages to different participants that contradict one another [38, 138]. Similarly, one approach to assuring learning-based systems is to look for proof of misbehavior in the guest OS. For instance, Lycosid detected inconsistencies between the `cr3` register and the purported process descriptor's `cr3` value [121]. A proof of misbehavior may also include inconsistencies in code paths or data access patterns between the training and classification phases of introspection.

| VMI should detect inconsistent behavior over the life of an OS, not just between training and classification. |
|---|

## 2.6 Under-explored Issues

Based on our survey of the literature on VMI, we identify a few issues that deserve more consideration in future work.

### 2.6.1 Scalability

Many VMI designs are fairly expensive, especially designs that run a sibling VM on a dedicated core for analysis. For example, one state-of-the-art system reports overheads ranging from 9.3—500$\times$ [96]. There is a reasonable argument why high VMI overheads might be acceptable: the average desktop has idle cores anyway, which could be fruitfully employed to improve system security. However, this argument does not hold in a cloud environment, where all cores can be utilized to service additional clients. In a cloud, customers will not be pleased with doubling their bill, nor would a provider be pleased with halving revenue.

It is reasonable to expect that VMI would be particularly useful on a cloud or other multi-VM system. Thus, future work on VMI must focus not only on novel techniques or threat models, but also on managing overheads and scalability with increasing numbers of VMs.

| VMI research must measure multi-tenant scalability. |
|---|

Another strategy to mitigate the costs of asynchronous scanning is to adjust the frequency of the scans—trading risk for performance. For instance, a recent system measured scanning time at 50ms, and could keep overheads at 1% by only scanning every 5s [113]. Similarly, one may cache and reuse introspection results to trade risk of stale data for better scalability [174]. An interesting direction for future work is identifying techniques that minimize both overheads and risk.

### 2.6.2 Privacy

VMI has the potential to create new side-channels in cloud systems. For instance, after reading application binaries, Patagonix [142] queries the NSRL database with the binary hash to determine the type of binary that is running on the system. This effectively leaks information about the programs run within a VM to an outside observer, undermining user privacy.

More generally, VMI has the potential for one guest to observe different cache timings based on the behavior of another guest. Consider a VMI tool that does periodic memory scans of multiple VMs on a cloud system, one after another. The memory scan or snapshot will disrupt cache timings of the guest under observation by forcing exclusive cache lines to transition back to a shared, read-only mode §2.4.2. Based on its own cache timings, the VM can observe the frequency of its periodic scans. Because the length of a scan of another VM can also be a function of what the VM is doing, changes in time between scans of one VM can indicate what is happening in another VM on the same system.

Although it is unclear whether this example side channel is exploitable in practice, the example raises the larger issue that VMI projects should be cognizant of potential side channels in a multi-VM system. Richter et al. [170] present initial work on privacy-preserving introspection, but more work is needed. An ideal system would not force the user to choose between integrity or privacy risks.

| VMI designs should evaluate risks of new side channels. |
|---|

## 2.7 Summary

Virtual machine introspection is a relatively mature research topic that has made substantial advances since the semantic gap problem was posed. However, efforts in this space should be refocused on removing trust from the guest OS in service of the larger goal of reducing the system's TCB. Moreover, future VMI solutions should balance innovative techniques and security properties with scalability and privacy concerns. We have observed that the lessons from previous work is guiding new efforts to adapt existing techniques and develop new techniques to bridge the strong semantic gap. The main takeaway from this chapter is that encapsulating the threat using a VM, and then relying on the VMI to neutralize that threat is an incomplete solution due to the strong semantic gap. Finally, this chapter observes that all VMI solutions that prevent an attack are based on the secure design principle of complete mediation and the technique of memory protection.

# CHAPTER 3: RETROFITTING LEAST PRIVILEGE PRINCIPLE ONTO SETUID-ROOT BINARIES

Unprivileged users of modern Unix systems access safe subsets of otherwise privileged system functionality through trusted, setuid-to-root binaries. For instance, the `mount` utility executes with administrative privilege, allowing unprivileged users to mount a CD-ROM or USB Flash device without involving a human administrator. Because the `mount` binary must issue the `mount()` system call, which requires root privilege (or the `CAP_SYS_ADMIN` capability), the `mount` binary is inadvertently empowered to issue many more privileged system calls. For instance, if an attacker can exploit an input parsing bug in `mount`, she might be able to change the root user's password, install a rootkit, or replace the contents of the `/etc` directory. Chen et al. [70] show that many privilege escalation attacks go through setuid-to-root binaries, even on SELinux [145] or AppArmor [28]. This attack surface is ubiquitous; e.g., 99.99% of surveyed systems install `mount` (§3.2.3).

The problem with setuid-to-root binaries is that they violate the Least Privilege Principle (LPP) [175], and create opportunities for privilege escalation attacks. As a result, most major Linux distributions had ongoing, but incomplete efforts to prune unnecessary setuid-to-root binaries [93, 197] (§3.2.1). Although these efforts have made substantial progress in reducing the number of setuid-to-root binaries, they leave a small core of "unavoidable" trusted binaries that continue to violate least privilege.

Previous research efforts [52, 101, 108, 130, 199, 205], as well as hardened Linux configurations, such as SELinux and AppArmor, have only considered least privilege on these utilities from the perspective of the administrator, not the untrusted user. In the case of mount utilities, systems like AppArmor attempt to limit the effects of a compromised `mount` to arbitrarily changing the file system tree. When the administrator executes `mount` with least privilege, she can only corrupt the file system tree, not change passwords (directly) or configure the network device. In contrast, least privilege for an unprivileged user should restrict that user to only mounting allow-listed devices and directories (e.g., `/cdrom`); even on AppArmor, a bug in `mount` could allow an unprivileged user to make arbitrary changes to the file system tree. SELinux further restricts `mount` to specific users and mountpoints, but still trusts `mount` to correctly map devices and options to these mountpoints. In this example, least privilege on these utilities can only be enforced by the OS kernel, and policies must be expressed not just in terms of the user requesting a system call, but also in terms of the objects of the requested call.

This chapter presents a simple, efficient framework for migrating policies from setuid-to-root binaries into the kernel, obviating the need for these privileged binaries in nearly all situations. We study the 28 most commonly installed binaries on Debian and Ubuntu Linux, which account for all setuid-to-root binaries on roughly 89.5% of systems surveyed (§3.2.3).

One essential insight from the study is that only eight system calls and a few other system interfaces underlie the vast majority of root privilege requirements. These system calls export functionality that is required by unprivileged users, yet required administrative privilege. For these system calls, this chapter proposes enforcing more expressive policies that more closely match the policies encoded in setuid binaries and configured by administrators. We present a prototype, called **Protego**, which extends the AppArmor [28] Linux Security Module (LSM) [206]. Protego changes only 715 lines of Linux kernel code, and adds additional trusted utilities to keep the kernel policy synchronized with legacy, policy-relevant configuration files, such as `/etc/sudoers`. In addition to these 8 system calls, a few additional system abstractions must be adjusted to remove privilege. Although Protego extends AppArmor, any security-hardened Linux variant could adopt these techniques.

An underlying motivation for setuid-to-root binaries is flexibility. When the kernel adopts a new abstraction, system developers may not understand precisely what the safe subsets of functionality are, or which subsets any real application will require. The underlying assumptions are that the setuid binary can be patched faster than the OS kernel, and that some experience may be required before a sufficient and minimal policy language can be defined. This chapter observes, however, that almost all setuid binaries are using abstractions that are decades old with very well-understood policies, and that modern kernels have flexible infrastructures for security policy enforcement [206]. Thus, there is no compelling reason to prefer setuid binaries to dynamically configurable policies enforced by a kernel security module.

The contributions of this work are as follows:

- A study of the policies encoded in setuid-to-root binaries on current Linux systems, considering privilege from the perspective of the non-administrative user.

| | |
|---|---|
| Net lines of code de-privileged. | 12,717 |
| Percentage of deployed Ubuntu and Debian systems that can eliminate the setuid bit. | 89.5% |
| Historical exploits that would be unprivileged on Protego. | 40/40 |
| Performance overheads. | $\leq$7.4% |
| System calls changed. | 8 |

Table 3.1: Summary of results.

- Identifying a set of straightforward changes to Linux which would obviate the need to violate the least privilege principle on most systems. Our prototype reduces the trusted computing base of Ubuntu Linux by 12,717 lines.
- Evaluating these changes on Linux, demonstrating that setuid-to-root binaries can be deprivileged with minimal overheads, minimal changes to trusted code, and no loss of functionality for users. For instance, a Linux kernel compilation on Protego is less than 2% slower than on unmodified Linux.

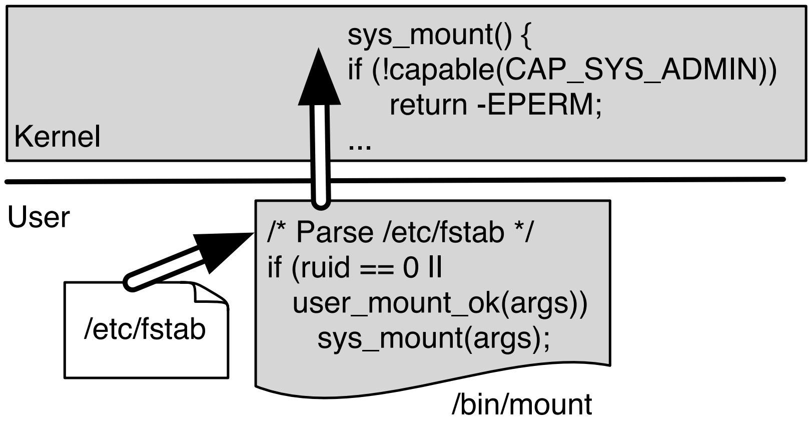
Thus, this chapter demonstrates that most deployed systems can uphold the Least Privilege Principle at minimal costs. Section 3.4.4 surveys the expected effort to remove setuid-to-root altogether.

## 3.1 Overview

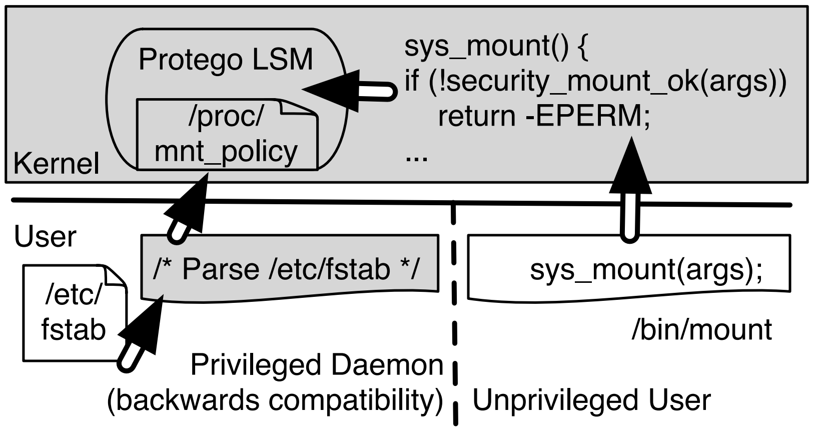
Protego executes setuid-to-root binaries without privilege. A fairly wide range of packages (currently 82) containing setuid-to-root binaries use only a small number of system calls (8), system files, and devices that require root privilege. Protego centralizes the policies currently encoded in this disparate collection of trusted binaries by instead adding Linux Security Module (LSM) hooks (§3.2.2) that apply equivalent policies in the kernel. Protego is an extension of AppArmor on Linux 3.6.0.

To explain the system design, we use the `mount` system call as a representative example. The `mount` system call grafts a new file system onto the file system directory tree at a given location. Similarly, the `umount` system call removes a file system from the directory tree. In general, changing the file system directory tree requires root privilege, as a malicious user might mount bad configuration files over `/etc` or even replace the system binaries in `/bin` or `/sbin`. Thus, the Linux kernel currently rejects any `mount` or `umount` call from processes without administrative privilege—namely the `CAP_SYS_ADMIN` capability, explained in §3.2.2.

Some mount-related binaries are `setuid` to root. The reason is to permit certain privileged operations without requiring administrator privilege, such as mounting a CD-ROM at `/dev/cdrom`. Operational



Linux `mount`.



Protego `mount`.

Figure 3.1: Comparison of the `mount` system call on Linux and Protego. Trusted components are in gray. Linux places trust in the `/bin/mount` binary to enforce policies specified in `/etc/fstab`; this binary is called by an untrusted user. The `mount` system call fails if the caller doesn't have the `CAP_SYS_ADMIN` capability (i.e., is not root). In Protego, a trusted daemon reads the policies from `/etc/fstab` and configures the Protego LSM through a file in `/proc`. Separately, an untrusted user can use `/bin/mount`, *or any other binary*, to issue a `mount` system call. The `mount` system call then calls an LSM hook to check this change against system policy.

constraints for these privileged functions are set by the administrator in `/etc/fstab` with the "user" or "users" option. The mount utilities, e.g., `mount` or `umount`, are then responsible for checking the real UID with which they were invoked: they are required to fail unless the file system and mount point match a "user" entry in `/etc/fstab`.

Protego is based on the observation that the kernel can just as easily perform these checks in an LSM. Both approaches are compared in Figure 3.1. For the `mount` system call, Protego keeps an allow-list of allowed user mountpoints in the kernel. If a process without `CAP_SYS_ADMIN` calls `mount`, the system

| Component | Description | Lines |
|---|---|---|
| **Kernel** | | |
| Linux | Additional LSM hooks, /proc filesystem interface. | 415 |
| Protego LSM module | Implement security policies, called by additional LSM hooks in Linux. | 200 |
| Netfilter | Extensions for raw sockets. | 100 |
| **Trusted Services** | | |
| Monitoring daemon | Trusted process that monitors changes in policy-relevant configuration files. Required only for backwards compatibility. | 400 |
| Authentication utility | Trusted binary launched by the kernel to authenticate user sessions, password protected groups. Code refactored from `login` and `newgrp`. | 1200 |
| **Utilities** | | |
| iptables | Extension for raw sockets. | 175 |
| vipw | Modified to edit per-user files instead of a shared database file. | +40 |
| dmcrypt-get-device | Switch to /sys to read underlying device information. | 43 |
| mount/umount, sudo, pppd | Disable hard-coded root uid checks. | -25 |
| | **Grand Total Changed** | 2,598 |

Table 3.2: Lines of code written or changed in Protego, including kernel, trusted services, and command-line utilities.

call will only succeed if the arguments match the allow-list. Mount utilities no longer require administrative privilege when invoked by unprivileged users, as the policies are migrated to the kernel.

This mount allow-list can be created by either the administrator by directly adding entries to a file in `/proc`, or, for convenience, we also provide a trusted daemon that reads and monitors `/etc/fstab`, propagating changes to the kernel via the `/proc` file. Protego provides two other files in `/proc` for configuration inputs using a simple grammar: a mapping of privileged ports to allowed application paths, and an `/etc/sudoers`-like syntax for delegation. Policies that do not take configuration parameters are simply hard-coded in the LSM. The underlying policy abstractions, concerns, and defaults derive from our study of current setuid-to-root binaries (Section 3.3).

Table 3.2 details the added or modified lines of code in Protego. The only code changes we made to most setuid utilities was removing checks that cause the binary to exit if the effective user id is not root—all policy checks are moved into the OS kernel and the utility runs without changing its effective user id to root. In a few cases, such as `vipw`, we modified these utilities to use different configuration files.

**Backward compatibility.** Protego modifies a few system configuration file formats. For example, Protego fragments the password database to better match policy and file system permissions; the monitoring database can keep the original database file and the new files synchronized for backwards compatibility. Although some previously-privileged applications must make changes to adopt the newer file formats, Protego maintains copies of the old formats for compatibility with other applications. Applications that did not previously require privilege are unmodified on Protego.

The monitoring daemon is written with a Python library [26] based on Linux's `inotify` file monitoring framework [184]. The monitoring daemon is only required for backward compatibility.

**Threat model.** We assume that setuid binaries may have programming errors that are exploitable through inputs carefully crafted by an adversary. We assume the adversary is an unprivileged user of a system who aims to acquire enhanced privileges (one or more administrative capabilities, or root access, depending on the system).

Protego's goal is to minimize the privilege held by binaries executed by a non-administrative user. Even if one of these binaries is compromised, the user acquires no more privilege than she already had before executing the binary. Other privilege escalation attack vectors, such as vulnerable system daemons running as root or bugs in system calls, are beyond the scope of this chapter.

## 3.2 Background

This section explains background on the setuid bit, related efforts to improve Linux security, and current setuid installation statistics.

### 3.2.1 The Setuid Bit

**Setuid bit vs. system call.** The namespace collision between the setuid permission bit and setuid system call can lead to some confusion. The primary mechanism to raise privilege in Linux is the setuid permission *bit* (04000) in an inode's `stat` field. When a setuid binary is executed, the process executes as the binary's owner, regardless of which user `exec`-ed the binary. Some privileged daemons, such as `ssh`, may initially execute as root and then drop to an unprivileged user by using the setuid *system call*. Similarly, setuid-to-root binaries often bound the risk of privilege escalation by dropping root privilege after completing the last privileged system call, using the setuid system call. Unless otherwise specified, setuid in this chapter refers to the bit.

Papers including “Setuid Demystified” explain how an application should use the `setuid` *system call* to drop its privileges [69, 195]. Our primary interest is complementary: eliminating privilege altogether in current setuid-to-root binaries, and thereby eliminating the need to securely drop privilege with the setuid system call.

**Why is setuid-to-root needed?** Administrators use setuid binaries to relax hard-coded policy decisions in the kernel, which are inconsistent with the desired system policy. Hecht et al. [108] observe that there are three categories of system calls: (1) unprivileged calls, (2) privileged calls, and (3) calls with privileged options. Modern Linux kernels generally enforce a stricter policy on calls with privileged options than administrators want, limiting application functionality. For example, the `mount` system call fails if the caller doesn't have administrative privilege—*even if the caller is only requesting options considered safe by system policy*. In the cases of `bind` and `open`, setuid is often used to allow an application to access a single port or file, but trusts the binary with access to *all* other ports or files. These point solutions implement the desired policy, but violate least privilege.

**setgid.** This chapter focuses on the `setuid`-to-root binaries, although similar issues could arise with the `setgid` bit. We note that no Debian or Ubuntu packages currently install binaries that are `setgid` to root. The delegation framework we describe in §3.3.3 provides equivalent functionality to setuid and setgid-nonroot on Protego.

**Eliminating setuid-to-root binaries.** Several Linux distributions have hardening efforts that have reduced the number of setuid-to-root binaries [93, 197]. For instance, Ubuntu has eliminated roughly 30 setuid-to-root packages between 2008-2014. These efforts have used the following major techniques:

- **Consolidation.** When several different packages perform similar tasks, developers create a shared setuid helper utility, such as the `sensible-mda` mail server utility.

- **File system permissions.** Utilities such as `at` write to logs and other protected system files, generally under the `/var` directory. Root privilege can be replaced with setuid non-root or setgid non-root by changing permissions on these files to an unprivileged user or group.

- **Capabilities.** Linux has a coarse capability model, which we explain next. Several utilities have replaced setuid with the similar `setcap` mechanism, which launches the binary with specific capabilities. Although setcap can replace setuid, several setuid-to-root binaries require capabilities tantamount to root.

These techniques are insufficient to enforce least privilege on all categories of current setuid-root binaries.

### 3.2.2 Capabilities, LSMs, and SELinux.

Capabilities in Linux are not pointers with fine-grained access control information, as commonly defined in capability-based operating systems [136, 182]. Linux divides root privilege into roughly 36 capabilities, called file system capabilities [77], which are roughly based on Trusted IRIX capabilities [119] and the POSIX.1e draft specification [165]. All uses of the term "capability" in this chapter refer to Linux file system capabilities.

By default, Linux gives all capabilities to a process running as root. A hardened Linux variant can reduce the capabilities granted to the administrator for a given task, as well as limit the capabilities given to a setuid-to-root binary.

Capabilities are designed to enforce least privilege *on the administrator*, but are generally too coarse to enforce least privilege on unprivileged users. For instance, if the administrator is configuring a network interface, the configuration utility may only run with the `CAP_NET_ADMIN` capability. If the configuration utility is buggy and makes errant privileged system calls, the damage is limited to the network.

Continuing our example, `CAP_NET_ADMIN` is required by the setuid-to-root `pppd` binary so that unprivileged users may make very restricted changes to the system's routing tables, such as creating a route if it does not conflict with previously existing routes. Even if `pppd` executes only with the `CAP_NET_ADMIN` capability, if `pppd` is compromised, the unprivileged user has substantially escalated her privileges, gaining the ability to arbitrarily change routes, disable devices, set privileged socket options, enable multicasting, etc. A potential solution to this problem is to bestow finer-grained capabilities on trusted binaries.

Unfortunately, developers have failed to effectively manage 36 coarse capabilities in the Linux kernel. Most Linux developers are not security experts, but nonetheless are required to place capability checks throughout the kernel. When in doubt, developers use the `CAP_SYS_ADMIN` capability. As a result, over 38% of all capability checks in Linux require this capability. Even our initial `mount` example requires

| **Package** | **Ubuntu(%)** | **Debian(%)** | **Wt.Avg.(%)** |
|---|---:|---:|---:|
| mount | 100.00 | 99.75 | 99.99 |
| login | 99.99 | 99.82 | 99.98 |
| passwd | 99.97 | 99.84 | 99.97 |
| iputils-ping | 99.87 | 99.60 | 99.85 |
| openssh-client | 99.54 | 99.48 | 99.53 |
| eject | 99.68 | 90.95 | 99.24 |
| sudo | 99.48 | 74.34 | 98.21 |
| ppp | 99.54 | 45.65 | 96.81 |
| iputils-tracepath | 99.78 | 13.06 | 95.39 |
| mtr-tiny | 99.54 | 11.79 | 95.10 |
| iputils-arping | 99.60 | 3.55 | 94.74 |
| libc-bin | 50.14 | 86.15 | 51.96 |
| fping | 27.70 | 12.42 | 26.92 |
| nfs-common | 9.76 | 82.89 | 13.46 |
| ecryptfs-utils | 11.64 | 0.72 | 11.08 |
| virtualbox | 10.56 | 7.78 | 10.41 |
| kppp | 10.11 | 4.97 | 9.85 |
| cifs-utils | 2.59 | 19.23 | 3.43 |
| tcptraceroute | 0.33 | 23.38 | 1.50 |
| chromium-browser | 0.48 | 8.49 | 0.89 |

Table 3.3: Percent of systems that install packages containing setuid-to-root binaries, as reported by the Debian and Ubuntu 'popularity contest' surveys. Average is weighted by the total number of systems reporting in each survey.

CAP_SYS_ADMIN. Moreover, this capability has become so permissive that it can acquire all other capabilities and is described as "the new root" [122]. Finally, the mapping of capabilities to privileged tasks is many-to-many. For instance, the X server requires 4 capabilities to set the video mode (CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_SYS_RAWIO, and CAP_SYS_ADMIN). Changing passwords requires 6 capabilities: CAP_SYS_ADMIN, CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_SETUID, CAP_DAC_READ_SEARCH, and CAP_FOWNER. Linux capabilities do not enforce least privilege on the administrator, much less limit the risk of privilege escalation by unprivileged users.

The deeper problem with Linux capabilities is that they provide an insufficient language to express policies for unprivileged users. Linux capabilities require a subject-based policy ("allow if requester is root"), yet most setuid-root binaries export select, safe operations on a kernel abstraction to all users—an object-based policy. As a result, system security hinges on a cumbersome, error-prone translation of an object-based policy onto a strictly subject-based language.

**Linux Security Modules (LSMs).** LSMs encapsulate sophisticated security policies from the rest of the kernel development process by placing suitable access control hooks throughout the kernel [206].

Security experts can then implement more advanced policies, such as mandatory access control (MAC), by using these hooks to override the default, discretionary access control policies. These hooks are intended to be sufficient to implement any policy without making additional changes outside of the module. The exact number of hooks varies (184 in Linux 3.13.5); Protego adds additional LSM hooks for system calls that are currently hard-coded to check specific capabilities.

**Security-hardened Linux variants.** SELinux [145] and AppArmor [28] are implemented as LSMs, as is Protego. SELinux provides a powerful mandatory access control (MAC) and multi-level security (MLS) model for Linux, complete with role-based access control [94] and security type enforcement. AppArmor is also an MLS implementation, and the default on Ubuntu. AppArmor tends to enforce coarser policies than SELinux.

LSMs can carefully control when a process is given a capability, and can use an LSM hook to obviate a capability check for some, but not all system calls. For example, SELinux associates capabilities with roles, which can prevent a process from accumulating capabilities. For instance, in order to use capabilities associated with another role, the process has to change roles, and thus relinquish the capabilities associated with the previous role. SELinux can also replace the coarse capability checks in `bind` with an allocation of low-numbered ports to types.

SELinux requires considerable policy effort to expose safe functionality to non-administrator users. For instance, SELinux must carefully manage the `CAP_NET_RAW` capability and assign the capability to trusted binaries, such as `ping`. This does not enforce least privilege, as `CAP_NET_RAW` is coarser than `ping`'s safe functionality (§3.3.1). SELinux could enforce simpler and more precise policies by adopting Protego's strategy of considering safe functionality separately from fragmenting administrator privilege.

Tools such as VulSAN [70] analyze system attack surface, generating the path for an attacker to install a rootkit. In many cases, the path goes through a setuid or capability-enhanced program, even on SELinux or AppArmor.

### 3.2.3 Setuid Installation Statistics

To focus our efforts on removing privilege from the most commonly-installed setuid-to-root binaries, we studied installation statistics collected by the Debian and Ubuntu distributions. The first step was to identify all potentially installable setuid binaries in all Debian and Ubuntu 12.10 APT repositories using the Lintian reports [32]. Ubuntu adopts and repackages stable versions of Debian packages, and the differences between the distributions tend to be minor. 82 packages contain setuid-to-root binaries.

We obtained the rough frequency of installation from the popularity contest results for all the monitored Ubuntu [33] and Debian [31] systems, based on over 2.5 million systems (2,502,647 Ubuntu and 134,020 Debian).

Table 3.3 lists the 20 most frequently installed packages, with per-distribution percentages and an average weighted by number of installations of each distribution. We have completely investigated all popular packages through ecryptfs-utils—indicating that roughly 89.5% of sample systems could adopt Protego with no loss of functionality. The 62 packages not listed are installed by fewer than .89% of systems sampled; Section 3.4.4 summarizes the additional work we foresee in deprivileging the remaining packages.

## 3.3 Setuid Policy Study

This section presents a detailed study of the policies encoded in the 28 most commonly installed setuid-to-root binaries. This study identifies the system-level policy goal encoded in the binary, how the kernel policy for a specific system call or other interface is mismatched to the policy goal, how low-level mechanisms can be easily modified to efficiently and comprehensively enforce these policies in the kernel, and how Protego enforces these policies.

In the interest of brevity, we do not discuss each binary in detail, but rather organize our discussion around each interface that requires administrative privilege. Table 3.4 summarizes these results. The Protego design is guided by two major themes from this study:

- **Express and enforce object-based policies.** The majority of these binaries encapsulate sensible policies that do not map onto subject-based checks (e.g., “is this user x?”), but that can be easily expressed as object-based policies (e.g., “may any user take this action on this object?”). For instance, several utilities enforce fine-grained access control on network ports, and enforce policies for raw sockets based on the protocol type (§3.3.1).

- **Interface designs can thwart least privilege.** In several cases, poor interface design can require applications to have more privilege than necessary. For instance, the `dmcrypt-get- device` utility uses a privileged `ioctl` to report the physical device underneath an encrypted block device; additional privilege is required because this ioctl also discloses the private key. A more subtle example of this

| Interface | Used by | Kernel policy | System policy | Security concerns | Our approach |
|---|---|---|---|---|---|
| `socket` | ping, ping6, arping, mtr, traceroute6.-iputils | Creating raw or packet sockets requires `CAP_NET_RAW`. | Users may send and receive safe, non TCP/UDP packets, such as ICMP. | Raw sockets allow one to send both benign packets (e.g., ICMP) and packets that appear to come from socket owned by another process. | Allow any user to create a raw or packet socket, but outgoing packets are subject to firewall rules that filter unsafe packets. |
| `ioctl` | pppd | Only the administrator my configure modem hardware or modify routing tables. | A user may configure a modem (if not in use) and add routes that don't conflict with existing routes. | Protect the integrity of routes for unrelated applications. | Add LSM hooks that verify routes do not conflict with old rules when requested by non-root users. |
|  | dmcrypt-get-device | Require `CAP_-SYS_ADMIN` to read dmcrypt metadata. | Any user may read public portion of dmcrypt metadata (e.g., device set). | The same `ioctl` discloses both the physical devices and the encryption keys. | Abandon this `ioctl` for a `/sys` file that only discloses the physical devices. |
| `bind` | procmail, sensible-mda, exim4 | Require `CAP_-NET_BIND_-SERVICE` to bind to ports $<$ 1024. | Mail server should generally run without root privilege. | Prevent untrustworthy applications from running on "well-known" ports. | System policies allocating low-numbered ports to specific (binary, userid) pairs. |
| `mount`, `umount` | fusermount, mount, umount | Mounting or unmounting a file system requires `CAP_SYS_ADMIN`. | Any user may mount or unmount entries in `/etc/fstab` with the "user(s)" option. | Protect the integrity of trusted directories (e.g., `/etc`, `/lib`); | Add LSM hooks that permit anyone to mount an allow-listed file system with safe locations and options. |

Table 3.4: Part I: System abstractions used by commonly installed setuid utilities on recent Debian and Ubuntu systems. The table identifies the common thread of inconsistent kernel policies and system policies for these abstractions, discusses the underlying security concerns, and how Protego unifies system and kernel policies.

| Interface | Used by | Kernel policy | System policy | Security concerns | Our approach |
|---|---|---|---|---|---|
| `setuid`, `setgid` | polkit-agent-helper-1, sudo, pkexec, dbus-daemon-launch-helper, su, sudoedit, newgrp | Only allowed with `CAP_SETUID`. | Permit delegation of commands as configured by administrator, in some cases require recent reauthentication. | Require authentication and authorization to execute as another user. | Add LSM hooks that check delegation rules encoded in files like `/etc/sudoers`, and a kernel abstraction for recency. |
| Credential databases | chfn, chsh, gpasswd, lppasswd, passwd | Only root can modify these files (or read `/etc/shadow`). | A user may change her own entry to update password, shell, etc. | Prevent users from accessing or modifying each other's accounts. | Fragment the database to per-user or per-group configuration files, matching DAC granularity. |
| Host private ssh key | ssh-keysign | Only root may read the key (FS permissions) | Allow non-root users to sign their public key with the host key, disabled by default. | A user should be able to acquire a host key signature without copying the host key. | Restrict file access to specific binaries instead of, or in addition to, user IDs. |
| Video driver control state | X | Root must set the video card control state, required by older drivers. | Any user may start an X server. | An untrustworthy application could misconfigure another application's video state. | Linux now context switches video devices in the kernel, called KMS. |
| /dev/pts* terminal slaves | pt_chown | Root must allocate pts slaves on pre-2.1 kernels. | Users may create terminal sessions. | This utility has been obviated for 17 years, but is still shipped. | Ignore. |

Table 3.5: Part II: System abstractions used by commonly installed setuid utilities on recent Debian and Ubuntu systems. The table identifies the common thread of inconsistent kernel policies and system policies for these abstractions, discusses the underlying security concerns, and how Protego unifies system and kernel policies.

point is the division of labor between user and kernel in the X window server (§3.3.5). Where needed, Protego adjusts these interfaces.

Several of the issues raised in this study could be addressed in more than one way, and some already have point solutions in another OS. This section strives to delineate the essential requirements of any solution from the particular solution implemented in the Protego prototype, as well as cite existing alternative point solutions. To the best of our knowledge, however, no system has comprehensively addressed all of the issues requiring setuid-to-root binaries.

### 3.3.1 Network

Three networking tasks require privilege: creating a raw or packet socket, the point-to-point protocol (PPP), and binding a socket to a TCP or UDP port less than 1024.

#### 3.3.1.1 Raw and Packet Sockets

When applications program with TCP or UDP sockets, the kernel encapsulates the application-level payload with the appropriate headers. In the rare case that an application needs to send messages with a protocol the kernel doesn't implement, the application must create most of the packet headers itself, using a *raw* or *packet socket* [27]. The difference between raw and packet sockets is that raw sockets provide the IP layer and MAC layer headers, whereas a packet socket only implements the MAC layer headers.

Linux requires the `CAP_NET_RAW` capability to create a raw or packet socket, because an application can also create packets that appear to come from another application. However, a number of setuid-to-root binaries, such as `ping`, allow unprivileged users to send safe packets over raw sockets.

A more precise, object-based policy, would specify which types of outgoing packets are acceptable from unprivileged users on raw sockets. We observe that the set of all safe packets exported by setuid-to-root binaries can be easily encoded as an allow-list in any packet filtering framework, such as Linux's netfilter/iptables [24], or BSD's Berkeley Packet Filters (BPF) [149]. For instance, some BSD variants use BPFs to sandbox binaries in a similar manner, such as ensuring the DHCP server sends DHCP packets only to limited addresses [194].

The Protego prototype allows unprivileged programs to create raw sockets, with the caveat that these unprivileged raw sockets are subject to additional netfilter rules. The default rules are based on the studied setuid-to-root binaries, but the rules may be changed by the administrator through the `iptables` utility. Protego implements this with modest extensions to the Linux netfilter framework, as well as adding an LSM hook to the `socket` system call.

In Protego, a compromised network utility cannot spoof packets from a TCP or UDP socket, unlike all current Linux variants. Also in contrast to Linux, Protego allows any unprivileged user to create her own enhanced `ping` utility, as long as it conforms to system security policy.

#### 3.3.1.2 Point-to-point Protocol (PPP)

PPP is a protocol used to establish connections over modems, including dial-up and cellular networks. In most Linux systems, the service that manages a PPP connection (`pppd`) requires root privilege for two tasks: configuring modem hardware and, potentially setting system routing tables to relay IP traffic through a PPP link. The `pppd` binary is setuid root so that it can be launched on demand, because, unlike ethernet, most PPP connections are not constantly active.

When `pppd` is launched as a user other than root, only certain safe configuration options are accepted, such as compression and congestion control session parameters. An administrator can also configure `pppd` to allow unprivileged users to add system routes over a ppp connection, but only if the new address ranges were not previously reachable. If a PPP connection duplicates an existing route, a user may only create a `tty` that communicates with the remote point.

We add LSM hooks to the appropriate `ioctl` system calls for configuring routing tables and modem options. Specific policies are mined from the ppp configuration file `/etc/ppp/options`. We also changed the default file system permissions on `/dev/ppp` to be more permissive, replacing a capability check with device file permissions.

We verified that `pppd` works without root privilege by connecting two machines over a crossover serial cable, such that one serves as an internet gateway to the other. Both machines ran `pppd` without root privilege, both were able to create routing table entries, and the non-gateway machine was able to connect to remote websites.

#### 3.3.1.3 Bind

Creating a socket that listens to a TCP or UDP port less than 1024 requires root or the `CAP_NET_BIND_SERVICE` capability. For this reason, many network services run with root privilege, at least temporarily, to set up a listening socket.

This policy reflects the notion that an administrator should be involved in setting up a service that listens on a well-known port. For instance, one expects that a web server listening on port 80 is endorsed by the administrator, but not if it is listening on port 8080. However, any privileged application can open any privileged port; for instance, a malicious web server can also act as an email or DNS server.

The system policy goal is that specific ports should be allocated to specific application instances. Protego uses a tuple of (binary path name, user ID) to represent an application instance, and a simple policy configuration file, `/etc/bind`, which maps each TCP or UDP port less than 1024 to an application instance. Each port may map to only one application instance.

Our strategy is a simplified version of SELinux's approach, which allocates ports to types. Using types to specify an application instance is sufficient, but not necessary; eliminating privilege escalation should not hinge on adopting SELinux. Similarly, Berkeley Packet Filters can also be used to delegate access to a privileged port; a centralized configuration has the advantage of easier auditing.

### 3.3.2 Mount

§3.1 explains our approach to the mount utilities as a motivating example; we will not repeat this for brevity. We validated that unprivileged users can mount `user` entries in `/etc/fstab` but not other file systems.

An alternative point solution used by many Linux and Unix systems is a trusted daemon (e.g., `automounter` [25]) that monitors accesses to allow-listed mount points, and automatically mounts them in response to attempts to access the device. Any design that allows unprivileged users to access allow-listed mount points is sufficient; the Protego design adds only 258 lines of trusted code to the system, whereas the Linux automounter implementation increases the TCB by 21,674 lines, including a 79 line kernel patch.

### 3.3.3 UID Switching and Delegation

A process changes its user and group IDs using a family of system calls including `setuid` and `setgid`, which require the `CAP_SETUID` or `CAP_SETGID` capabilities. These capabilities give a process the ability to assume any user's or group's id. Utilities such as `sudo` are setuid root so that they start with administrator privilege, validate that the invoking user is allowed to switch to the new user, and then call `setuid` to drop privileges.

Because our interest is in enforcing least privilege on non-administrator users, we focus on *lateral* moves, where one unprivileged user (Alice) acts for another (Bob). Most delegation utilities, such as `sudo`, violate the principle of least authority by giving Alice the privilege to run as *any* user via a setuid root binary, and then dropping back down to Bob with the setuid system call.

Ideally, Alice's `sudo` process should never be able to switch to any uid other than Bob's, and only in ways Bob has authorized her to act for him. If sudo enforced least privilege on Alice, even if Alice compromises her sudo process, she should never be able to switch to an unrelated user, say Charlie. Obviously, in many cases, `sudo` is used to transition to root; even in this case, root privilege should only be granted to the process *after* all checks have succeeded, not before. These policies are only possible if the kernel, not a user application, validates `setuid` system call transitions.

Taking `sudo` as a representative example, there are two checks that require a trusted agent:

- **Authentication:** Either the current or target user's password should be entered and checked. Utilities like `sudo` check the calling user's password to ensure that another person hasn't sat down at an unattended terminal. `sudo` only checks the password if a password has not been entered on the terminal in the last 5 minutes. In contrast, utilities like `su` ask for the target user's password as both authentication and authorization to act as the target user.
- **Authorization:** The administrator may configure `sudo` to only allow a user to issue specific commands as another user. For instance, Alice may allow Bob to issue the `lpr` command to print with her credentials, but `sudo` will not allow Bob to directly execute any other binaries as Alice. Specifying a particular command also requires limiting inheritance of environment variables or open file descriptors, to ensure integrity of the delegated command.

An untrusted binary should not be charged with either.

Rather than check for administrator privilege on a `setuid` system call, Protego enforces the policies collected from the `/etc/sudoers` configuration file and the configuration files in the `/etc/sudoers.d/` directory. Policies currently encoded in setuid binaries are explicated in additional `/etc/sudoers` rules. Similar to `/etc/fstab`, the monitoring daemon parses these files, watches for changes, and sets the kernel policies accordingly. Protego also requires that any user issuing a `setuid` system call be recently authenticated, unless specifically countermanded by a `sudoers` policy with the `NOPASSWD` directive. We shift the validation of command-line arguments to the kernel.

The Protego kernel tracks the last authentication time in the `task_struct` of each process. If a `setuid` system call is issued without a recent authentication of the current user, a trusted authentication service temporarily takes over the terminal and asks for the user's password. This authentication service is refactored from the `login` source code. This service can also request the password of another user or group, according to system policy.

A challenge in restricting `sudo` privilege to a given binary is that policy enforcement must span two system calls: `setuid` and `exec`. To achieve least privilege, the process privileges must not change before the `exec` system call. To address this, we slightly change the behavior of `setuid` for restricted UID transitions. If a process could make an `exec` call with at least one permissible binary, the `setuid` system call's return code (0) will indicate success to the application, but the system call will set a field in the task's security metadata indicating a pending setuid-on-exec and storing the pending user. When a process with the setuid-on-exec field set issues an `exec` call, Protego hooks the `exec` system call and checks whether the requested binary is allowed as the pending user. The authentication service may also ask for the target user's password at this point. If the user is not authorized to `exec` the requested binary as the target user, the `exec` will fail with a permission denied error. This change in error behavior is difficult to avoid when enforcement must span two system calls; in practice, our target utilities have worked correctly despite this change. We also add appropriate restrictions on inheritance through setuid transitions, except where explicitly permitted in the sudoers policy.

In our example of Alice acting for Bob, a setuid-to-Bob binary can provide the desired user transition. In practice, tools like `sudo` are preferred because `sudo` is more flexible, centralizes system policy in an easy-to-audit location, and sanitizes inputs and environment variables to reduce the application's attack surface. An alternative approach to implementing a least-privilege `sudo` might generate a set of setuid-nonroot binaries that encode the system policies, similar to capability amplification in Hydra [207].

Protego encodes the policies of a wide range of delegation utilities as extended sudoers rules, including su, sudoedit, dbus, policykit, and newgrp. For instance, `newgrp` exports password-protected groups; this functionality can be encoded by requiring the user authenticate on a setgid system call that requests certain groups. We omit full details of these utilities for brevity.

### 3.3.4 File System Permissions

Several setuid-to-root binaries require root in order to work around inconvenient file system permissions. For instance, mail servers use root privilege to access configuration files, such as `.forward`, even if the user's permissions otherwise block access to the file. Similarly, most of the software for managing one's local account, such as `passwd` and `chsh`, requires access to a single record in a database file, but the kernel only enforces access control at the file granularity.

In order for a user to change her password or default shell, she should be able to modify her own entry of the database files. However, if Alice can modify Bob's database entry, Alice can access or modify Bob's account—a clear security problem. Thus, the shared database files can only be written by root. In terms of capabilities, a process must acquire `CAP_SYS_ADMIN`, `CAP_CHOWN`, `CAP_DAC_OVERRIDE`, `CAP_SETUID`, `CAP_DAC_READ_SEARCH`, and `CAP_FOWNER` to complete this operation—undermining the goal of substantially limiting administrative privilege with capabilities. To permit users to modify their credentials and preferences, `passwd`, `chsh`, and similar utilities are setuid and validate that the update does not corrupt the entire database.

To modify one's account with least privilege, the system must enforce access control at the granularity of a user's record, not the entire database. Protego splits the shared database files into per-account files. For instance, we split `/etc/passwd` into one file per-user under `/etc/passwds/`. Each file has permissions `rw-------`, owned by the user it defines and the parent directory `/etc/passwds/` has permissions `rwxr-xr-x` owned by user root and group root so that unprivileged users cannot add new users to the system. For backward compatibility, a trusted daemon monitors the per-user files and updates the legacy `/etc/passwd` file (§3.1). A similar approach is taken with `/etc/group` and `/etc/shadow`.

Network directory servers, such as OSX's Directory Server [41] or OpenLDAP [23], already enforce record-level access control. However, such a mechanism is generally not available for the local system accounts, and adds substantially more code to the system TCB (Protego changes 240 LoC, whereas OpenLDAP Version 2.8 is 175,368 LoC).

In the case of mail delivery problems, we advocate sufficient warnings in the log to diagnose delivery problems resulting from incorrect file permissions.

We note that users may desire finer-grained access control to prevent their password hash from leaking from their `/etc/shadows` file. Protego mitigates this risk by requiring the user to reauthenticate before reading the shadow file using the mechanisms described above (§3.3.3). The shadow file handle may not be inherited (`CLOSE_ON_EXEC`).

### 3.3.5 Interface Design

A small number of utilities require root privilege because the kernel's interface design forces otherwise unnecessary trust on the utility. In many cases, non-security reasons naturally lead to more sensible interfaces and obviate the need for trusted binaries. For instance, very old versions (before 2.1, or 1996) of the Linux pseudo-terminal implementation required applications to allocate slave [sic] terminal devices, and thus required a trusted binary to prevent applications from interfering with each other. Ultimately, allocating slaves in the kernel was simpler all around, and obviated the need for a trusted `pt_chown` utility.

Similarly, the X window server is setuid to root because the X server may need to both *manage* the graphics hardware as well as *display* the user's composite desktop. In practice, simply drawing a screen image to a frame buffer does not require any administrator privileges. Root privilege is required to configure and context switch the video hardware, e.g., setting the refresh rate, screen resolution, and, most importantly, configuring which application's frame buffer should be displayed on the screen [29].

In practice, video card management has been migrating from the X server into the kernel. When the X server manages the video card, developers find writing context switching code for a range of video hardware cumbersome. The video configuration changes when a user switches from one X server to another, or from the X server to a text console (via Ctrl-Alt-F1). The X server must correctly save and restore all state for all supported video cards. Thus, X and video driver developers have decided that a more sensible division of labor is for the OS kernel to manage context switching the video card. Linux now has a standard interface for device drivers to save and restore device state, called Kernel Mode Setting (KMS) [29], introduced in Linux 2.6.29.

KMS eliminates the need for X to run as root [78], and is ubiquitous among all major device manufacturers. Device drivers with KMS support have been written for chipsets manufactured by Intel, ATI (radeon), Nvidia (nouveau driver), and others. Nvidia's closed-source drivers have also adopted KMS [131]. We have verified this on our own machines using the Nouveau driver and running an X server binary that is not setuid to root.

An interesting contrast is that, unlike many setuid binaries, X and the Linux video drivers are still under very active development. The same decision that led to a needless attack surface has caused enough practical problems that developers are rectifying the problem for non-security reasons.

### 3.3.6 Limitations and Discussion

Our study covers a representative sample of setuid-to-root binaries, and we expect that Protego can easily support most of the remaining setuid-to-root binaries. We note that the Protego prototype allows an administrator to reenable the setuid bit if necessary. If the setuid bit is actually removed from all other binaries when the bit is reenabled on the system, the only marginal difference will be that the one unsupported binary is added to the system's trusted computing base.

The most likely situations where the setuid bit may be required are new kernel interfaces where the desired policy is not well understood. For instance, Linux has been gradually adding support for sandboxing with restricted namespaces [123], beginning with version 2.6.23. Until version 3.8, the security implications were not sufficiently understood, and sandboxing utilities, such as chromium-sandbox, had to run setuid-to-root. The security implications are now better understood, and newer kernels allow unprivileged users to use safe namespace configurations.

The main weakness of the Protego design is that its tools for mitigating information leaks are limited compared to SELinux, but more powerful than unmodified Linux. Two setuid binaries need to read sensitive data: `ssh-keysign` and password changing utilities. In these cases, a measure of trust is unavoidable. Protego develops fine-grained delegation rules (§3.3.3) that allow only these trusted binaries to read specific sensitive files. These restrictions are not as strong as SELinux roles [94, 145], but they can be combined. For example, a keysigning role might take away network access and write permission to any file handle other than a pipe to the parent. In comparison, Linux allows *any* program with root privilege to read these files. Protego's contributions are orthogonal to SELinux's, and could be adopted by SELinux to mutual benefit.

## 3.4 Evaluation

This section aims to answer the following questions:

- What is the cost (in execution time and network throughput) of moving setuid policies into the kernel? In particular, what cost is imposed on applications that do not use any privileged functionality?
- How does Protego affect overall system security?
- Are Protego functionality and policies identical to unmodified Linux?

| Test | Linux | +/- | Protego | +/- | % OH |
|---|---|---|---|---|---|
| **lmbench** | | | | | |
| syscall (us) | 0.04 | 0.00 | 0.04 | 0.00 | 0.00 |
| read | 0.09 | 0.00 | 0.09 | 0.00 | 0.00 |
| write | 0.09 | 0.00 | 0.09 | 0.00 | 0.00 |
| stat | 0.34 | 0.00 | 0.33 | 0.00 | -2.94 |
| open/close | 1.17 | 0.09 | 1.17 | 0.09 | 0.00 |
| mount/umnt | 525.15 | 18.82 | 531.13 | 19.44 | 1.13 |
| setuid | 0.82 | 0.09 | 0.83 | 0.06 | 1.22 |
| setgid | 0.82 | 0.09 | 0.83 | 0.09 | 1.22 |
| ioctl | 2.76 | 0.45 | 2.78 | 0.48 | 0.72 |
| bind | 1.77 | 0.10 | 1.81 | 0.11 | 2.25 |
| sig install | 0.10 | 0.00 | 0.10 | 0.00 | 0.00 |
| sig overhead | 0.70 | 0.00 | 0.70 | 0.00 | 0.00 |
| prot. Fault | 0.19 | 0.00 | 0.19 | 0.00 | 0.00 |
| fork+exit | 159.00 | 0.99 | 158.00 | 0.40 | -0.63 |
| fork+execve | 554.00 | 1.86 | 573.00 | 1.81 | 3.43 |
| fork+/bin/sh | 1360.00 | 1.33 | 1413.00 | 1.72 | 3.90 |
| 0KB create | 5.57 | 0.39 | 5.43 | 0.37 | -2.51 |
| 0KB delete | 3.93 | 0.59 | 3.79 | 0.73 | -3.56 |
| 10KB create | 11.00 | 0.13 | 10.80 | 0.10 | -1.82 |
| 10KB delete | 5.90 | 0.36 | 5.85 | 0.57 | -0.85 |
| AF_UNIX | 9.30 | 0.00 | 9.69 | 0.00 | 4.19 |
| Pipe | 6.73 | 0.00 | 6.88 | 0.00 | 2.23 |
| TCP connect | 18.00 | 0.00 | 18.55 | 0.00 | 3.05 |
| Local TCP lat | 19.63 | 0.00 | 20.87 | 0.00 | 6.32 |
| Local UDP lat | 16.70 | 0.00 | 17.90 | 0.00 | 7.19 |
| Rem. UDP lat | 543.60 | 0.00 | 578.30 | 0.00 | 6.38 |
| Rem. TCP lat | 588.10 | 0.00 | 631.50 | 0.00 | 7.38 |
| BW (MB/s) | 5316.60 | 0.00 | 5170.69 | 0.00 | 2.74 |
| **Postal Benchmark for Exim server** | | | | | |
| Messages/min | 258.64 | 3.33 | 258.75 | 3.09 | 0.04 |
| **Kernel Compile** | | | | | |
| time(s) | 764.41 | 4.36 | 775.39 | 5.28 | 1.44 |
| **Apache Bench** | | | | | |
| Time per request (ms, lower is better) | | | | | |
| 25 conc. reqs | 0.28 | 0.01 | 0.29 | 0.01 | 3.57 |
| 50 conc. reqs | 0.26 | 0.00 | 0.27 | 0.00 | 3.85 |
| 100 conc. reqs | 0.25 | 0.01 | 0.26 | 0.01 | 4.00 |
| 200 conc. reqs | 1.13 | 0.02 | 1.16 | 0.02 | 2.65 |
| Transfer rate (Kbps, higher is better) | | | | | |
| 25 conc. reqs | 6781.04 | 134.14 | 6506.29 | 157.34 | 4.05 |
| 50 conc. reqs | 7375.21 | 253.56 | 7083.63 | 248.34 | 3.95 |
| 100 conc. reqs | 7342.15 | 206.45 | 7051.54 | 236.78 | 3.96 |
| 200 conc. reqs | 1642.90 | 106.46 | 1599.55 | 113.92 | 2.64 |

Table 3.6: Protego overheads compared to Linux with AppArmor. Unless otherwise noted, tests measure execution time in microseconds (lower is better). A few tests measure bandwidth in MBps or Kbps, where higher is better.

- What effort would be required to deprivilege the remaining 67 packages containing setuid-to-root binaries?

Our baseline is an unmodified Linux 3.6.0 kernel configured to use AppArmor and iptables with no firewall rules on Ubuntu 12.04. Protego was refactored from Linux 3.6.0. All measurements were collected on a Dell Optiplex 790 with a 4-core 3.40 GHz Intel Core i7 CPU. The test machine has 4 GB of memory and a 250GB, 7200 RPM SATA disk.

### 3.4.1 Performance Overheads

We measure the performance cost of Protego policy enforcement on applications that do not require any privilege on Linux with both application benchmarks and the lmbench microbenchmark suite, version 3.0-a9. We note that within a run, we use the standard lmbench configuration, which includes a number of iterations scaled appropriately to the test case. We report the mean and 95% confidence intervals. We also measure Linux kernel 3.6.6 compilation time and network performance using the ApacheBench benchmark, version 2.3, exercising an Apache web server, version 2.2.16.

We note that most applications which use the `setuid` bit are interactive, and thus difficult to meaningfully benchmark. The most interesting exception to this are the mail servers, so we include measurements of exim4 server using the Postal [75] benchmark. We also extend lmbench with 5 additional tests that exercise the system calls we modified.

In general, the overheads of Protego are low, ranging from 0–7.4%. Table 3.6 lists the measurements of Protego, as well as overhead relative to Linux. For many system calls, Protego has no effect on the behavior and performance is comparable. In the case of `exec`, for instance, Protego adds 5.78% overhead to enforce setuid policies. A few microbenchmarks show small performance improvements commensurate with the confidence intervals, which we believe are noise. At the macro-benchmark level, such as a Linux kernel compile, Protego overhead is only 1.44%.

In order to enforce policies on raw network packets, we add additional netfilter rules on all outgoing packets. To measure the impact on unprivileged applications, we compare to netfilter with no rules configured, and the overhead ranges from 2–4% for the standard ApacheBench web service benchmark. These results indicate that the performance overheads are acceptable.

| Utilities | Total CVEs | Priv. Esc. | CVE Identifiers |
|---|---|---|---|
| ping | 84 | 4 | 1999-1208, 2000-1213, 2000-1214, 2001-0499 |
| traceroute | 26 | 2 | 2005-2071, 2011-0765 |
| mount,umount | 114 | 2 | 2006-2183, 2007-5191 |
| mtr | 4 | 3 | 2000-0172, 2002-0497, 2004-1224 |
| sendmail | 84 | 2 | 1999-0130, 1999-0203 |
| exim | 21 | 2 | 2010-2023, 2010-2024 |
| sudo | 61 | 5 | 2001-0279, 2002-0043, 2002-0184, 2009-0034, 2010-2956 |
| sudoedit | 3 | 1 | 2004-1689 |
| newgrp | 7 | 6 | 1999-0050, 2000-0730, 2000-0755, 2001-0379, 2004-1328, 2005-0816 |
| passwd | 87 | 1 | 2006-3378 |
| passwd, su | - | 1 | 2003-0784 |
| su | 31 | 2 | 2000-0996, 2002-0816 |
| chsh, chfn, su, passwd | - | 1 | 2002-1616 |
| chsh, chfn | 10 | 2 | 2005-1335, 2011-0721 |
| dbus | 22 | 1 | 2012-3524 |
| pkexec, policykit | 24 | 2 | 2011-1485, 2011-4945 |
| X | 33 | 2 | 2002-0517, 2006-4447 |
| capabilities | 7 | 1 | 2000-0506 |

Table 3.7: Historical vulnerabilities in setuid-to-root binaries (total, and those that lead to privilege escalation). In Protego, these utilities and the vulnerable code would be deprivileged and would not lead to a privilege escalation. Dashes are placed in the "Total" column for a CVE that spans multiple packages; the package totals are reported in other rows.

### 3.4.2 Security Evaluation

It is difficult to quantify the change in risk of any change to a system interface and enforcement mechanism. To evaluate Protego we consider two rough indicators: the net change in the trusted computing base and whether historical vulnerabilities in `setuid` binaries would occur in code that is now de-privileged, or in code that moved into the OS kernel.

**Trusted Computing Base.** We first carefully measure the change in privileged lines of code in the previously `setuid` binaries. At a high level, Protego adds 715 lines of policy checking code to the Linux kernel, 400 lines of code in Python which monitors certain configuration files for changes, and a 1,200 line authentication utility. The authentication utility is refactored from existing trusted code in `login` and `newgrp`. To balance this number, we also measure the lines of trusted binary code that no longer execute with privilege—a total of 15,047. We are careful to report a conservative estimate—ignoring whitespace, comments, standard libraries, and any code that would have executed after dropping privilege in the original code. Thus, Protego decreases the lines of trusted code by at least 12,732.

One potential concern is that Protego (conceptually) migrates policy enforcement code from applications to the kernel. We hasten to note that the policy enforcement code in the kernel is only 200 lines of straightforward C code. Protego also expands the purview of policies enforced in the kernel; these concerns are localized to our LSM, and generally make existing kernel policies more precise.

Similarly, one might be concerned about adding the user-level authentication service and monitoring daemon to the trusted computing base. We note that authentication code is trusted in both systems, the only difference is how it is invoked. Parsing configuration files can introduce new vulnerabilities, but the risk is small, as the files are simple. This daemon is written in a managed language with regular expressions (Python) to minimize the risks commonly associated with input parsing. The monitoring service could be eliminated by changing additional legacy code. The code we add to the TCB is short, simple, and easily audited.

A more subtle issue in selecting a configuration format is the risk of misconfiguration. Whenever possible, we chose to use legacy configuration files administrators would find familiar. This choice is debatable and largely incidental to the Protego design; one could just as easily have a single policy file, as SELinux does, and perhaps manage this with some application to assist the administrator, all using the same underlying `/proc` interfaces to the Protego LSM.

**Historical Vulnerabilities.** We surveyed the Common Vulnerabilities Database [156] entries for the 28 binaries this chapter studies. We found 618 total vulnerabilities over the lifespan of these utilities—40 of which led to acquisition of root privilege, summarized in Table 3.7.

We manually analyzed these 40 privilege escalation vulnerabilities and determined that these executed in code that now runs without privilege in Protego. Most of these vulnerabilities exploit buffer overflows, format strings, or environment variables. Moreover, these vulnerabilities are not in code substantially similar to the parsing, policy checking, or authentication code that we have added to Protego's TCB.

This data indicates that the probability of new privilege escalation vulnerabilities is relatively low, but the overall risk is still substantial. Most of the credit for the low probability of privilege escalation belongs to efforts to drop privilege after the privileged system calls have executed. Nonetheless, our analysis indicates that kernel support in Protego can further reduce the risk of privilege escalation.

As code matures the probability of exploitable bugs decreases. Although the most popular packages are generally quite old, Ubuntu has added 21 setuid-to-root binaries based on new code between 2011-2014,

| Binary | Coverage % | Binary | Coverage % |
|---|---|---|---|
| chfn | 94.4 | sudo | 90.1 |
| chsh | 92.7 | sudoedit | 90.9 |
| gpasswd | 91.3 | mount | 94.1 |
| newgrp | 93.5 | umount | 92.5 |
| passwd | 91.0 | ping | 96.2 |
| su | 92.2 | | |

Table 3.8: Gcov coverage of command-line setuid binaries

despite a net reduction in setuid binaries. The long-term goal of this project is to completely obviate the need for new setuid-to-root binaries, and their considerably higher risk of exploitable bugs.

### 3.4.3 Functional Testing

In addition to manual functionality tests, we validate that Protego behaves equivalently to unmodified Linux with exhaustive test scripts for setuid command line utilities. We validate that the utilities have the same output and effects on both systems. We also use `gcov` [105] to measure the test coverage for the command-line utilities in Table 3.8, which is always above 90%. Although exercising nearly all code paths does not necessarily mean all inputs are handled equivalently on both systems, one can infer that the functionality and policy enforcement is very likely to be equivalent.

### 3.4.4 Toward Zero Setuid-To-Root Binaries

This subsection surveys the remaining 67 packages (91 binaries) in Ubuntu Linux to assess how many can likely be deprivileged on Protego and how many will require different approaches. We note that this survey is preliminary, based on the documentation, and we have not tested these on Protego.

Table 3.9 groups the remaining binaries by the underlying interfaces that require privilege. We observe that 77 use interfaces already addressed by Protego, but may require refinement to the policies currently enforced. The remaining 14 binaries require privilege for:

- **Namespaces** (6). §3.3.6 explains that namespaces no longer require privilege in Linux kernel versions 3.8 and higher.
- **System administration** (3). Three binaries use privilege to reboot the system, load kernel modules, or configure the network. Some may be able to use PolicyKit or sudo (§3.3.3), but others may require additional consideration.

| Interface | No. of Setuid Binaries |
|---|---:|
| socket | 14 |
| bind | 23 |
| mount | 3 |
| setuid,setgid | 24 |
| Video driver control state | 13 |
| chroot/namespace | 6 |
| miscellaneous | 8 |

Table 3.9: System abstractions used by setuid binaries in packages not included in the Section 3.3 study. Abstractions below the double line need to be addressed in future work.

- **Open a custom device** (5). Virtualbox includes a kernel-level virtual machine monitor, which exports a custom device to 5 setuid binaries. These applications and the kernel module are tightly coupled, and additional work will be required to identify a sensible policy for this device.

Thus, we are optimistic that a few additional policy abstractions can complete our ongoing effort to obviate the need for all setuid-to-root binaries.

### 3.4.5 Design Principles for Protection Mechanisms in Protego

Protego applies design principles to setuid-to-root binaries in userspace, but enforcement is done in kernel space. This thesis will now discuss how the secure design principles are followed in Protego.

**Economy of Mechanism:** Protego changes only 715 LoC in the kernel to enforce the least privilege policies. Almost all the changes are made in the LSM piece of code. Thus, Protego demonstrates the use of economy of mechanism in the Linux kernel.

**Fail-Safe Defaults:** Protego uses allow-list policies that explicitly allow certain functions. All other functions not explicitly allowed by the administrator are denied by default for the untrusted user.

**Complete Mediation:** The LSM code is based on hooks appropriately placed throughout the kernel to check permission before taking any action. Protego leverages this design principle followed by LSM to ensure complete mediation for the setuid-to-root binaries.

**Open Design:** Protego does not have any secret algorithms. The complete code is open source.

**Least Privilege:** The least privilege principle states that each entity should gain exactly the right privileges necessary to perform the function they are allowed to do, and nothing more. This principle was violated by the setuid-to-root binaries as they gained the root privilege or all-encompassing capabilities. The Least Privilege principle is at the core of the Protego design. Protego determines the exact privileges needed to use one of the privileged abstractions based on the administrator policies, and enforces them in the kernel.

**Psychological Acceptability:** As the table 3.2 shows, Protego adds a monitoring daemon and the authentication utility to maintain backward compatibility, and not change the untrusted user's interface to these setuid-to-root binaries.

## 3.5 Related Work

This section surveys related efforts to reduce the privilege of setuid binaries, which generally speaking, either enforce least privilege on the *administrator*, but not regular user, or simply remove functionality from users.

Much related work on setuid binaries attempts to mitigate the risk of privilege escalation attacks by allocating privilege only to portions of a program (also known as privilege bracketing or privilege separation). Systrace [167] mitigates privilege escalation attacks by localizing privilege in setuid binaries to a single system call (e.g., the `socket` call in `ping`). Executable based access control [56] specifies which files a trusted binary may open; this design cannot prevent an errant `passwd` utility from changing other users' passwords, nor does it restrict privileged system calls unrelated to files. Protection domains within a setuid binary have also been proposed to preventing exploits in unprivileged operations from leaking into privileged code [186].

Secure Xenix [101, 108], and other secure Unix variants [52, 130, 199, 205] developed modern best practices for enforcing least privilege on the administrator: fragmenting administrative privilege into roles or capabilities, restricting the ability to create more setuid binaries, and removing the setuid bit if a binary is overwritten. Limiting the risk of exploiting a setuid binary is complimentary to Protego's goal of eliminating the need for setuid binaries.

Plan 9 eliminates setuid binaries by making every OS resource a file or file system, and by prolific use of fine-grained capabilities [79]. For instance, Plan 9 represents all networking abstractions as files with capabilities, whereas Protego enforces finer-grained network security policies.

Although Windows has a richer access control model than Linux users and groups [191], Windows adopts similar practices for some resources, such as requiring Administrator privilege to create a raw socket [155].

Finally, Bastille [47] simply removes the setuid flag and supported functionality from many utilities. For instance, non-privileged users cannot mount a USB drive, ping another computer, or use the `traceroute` command—functionality that could be safely reinstated on Protego.

**Namespaces.** Linux 3.8 allows unprivileged sandboxing applications, such as `chromium`, to create sandboxed network, mount, user, and process namespaces [123].

Namespaces are designed for isolating untrusted binaries, and are simply the wrong tool for enforcing least privilege when accessing shared system abstractions. Inside of a sandbox, a process can appear to have any capability, but any externally visible operations are subject to the original user's privilege. For instance, network namespaces allow an unprivileged process to access a fake network interface, and send ICMP packets within a fake network with no routes to the outside world. The major caveat is that any connection to the outside world requires an agent outside of the sandbox with the appropriate capabilities. In our network example, the agent would still need the `CAP_NET_RAW` capability to send ICMP packets out of the sandbox. In contrast, namespaces cannot safely allow access to shared system resources, such as `passwd` updating the password database.

## 3.6 Summary

This chapter presents a study of setuid-to-root binaries on modern systems, yielding insights into the nature of least privilege, especially that one should consider the administrator and user separately. There is an interplay between the division of labor between the kernel and userspace that directly impacts the need for trust; trusted binaries are often compensating for a design flaw in the system interface. Protego demonstrates techniques that can eliminate this long-standing attack surface without sacrificing functionality.

# CHAPTER 4: RETROFITTING COMPLETE MEDIATION AND MEMORY PROTECTION TO ISOLATE LINUX CONTAINERS

This chapter leverages virtualization hardware, such as Extended Page Tables (EPT) [51], to isolate the memory of OS containers by redesigning the OS abstractions to match the hardware requirements.

OS virtualization systems like Docker and Linux Containers (LXC) have been gaining popularity for cloud computing because of their low resource consumption. Profitability of operating a cloud is tied to how many applications the cloud provider can pack on the same hardware. Thus, the most common cloud use-case for containers is a multi-tenant environment, where containers of competitors may be running on the same physical machine. The security guarantees given by the OS virtualization rely on creating container-specific copies of selected kernel data structures and redirecting the use of these objects to an appropriate container-specific copy. Vulnerabilities in the Linux namespace code that implements this indirection can be exploited to gain unauthorized access to other containers' objects. Currently, there are 15 reported vulnerabilities in the Linux namespace code, which cause privilege escalation [1]. This is a security concern for the widespread adoption of OS virtualization for multi-tenant cloud computing.

The multi-tenancy security concern for containers exists because the container-specific copies of kernel objects are not isolated from each other. Containers reduce their footprint by sharing the OS kernel between the host and the container. As a result, all containers share the vulnerability of the OS kernel. A malicious application in a container can exploit the shared kernel to access or manipulate other containers' kernel objects, making multi-tenancy security difficult to implement for containers. The multi-tenancy security for containers needs a way to control access and isolate the container-specific kernel objects.

Virtual machines provide better security isolation for the applications within the VM compared to containers. VMs do not face the problem of isolating kernel objects because the hypervisor isolates each VM's memory for kernel data structures using the second set of page tables like shadow page tables or Extended Page Tables (EPT) [51] as described in §4.2. Moreover, the interface between the hypervisor and the guest is much narrower than the wide system call interface for containers. However, in the cloud environment, virtual machines run a complete legacy OS for one application and thus lead to substantial overheads in terms of memory and disk space compared to lightweight OS virtualization. Moreover, the

| Technique | time(sec) |
|---|---|
| KVM Start-Up | 10.342 |
| LXC Start-Up | 0.200 |
| FreeBSD Jail Start-Up | 0.090 |

Table 4.1: Start-Up times for a VM and a container

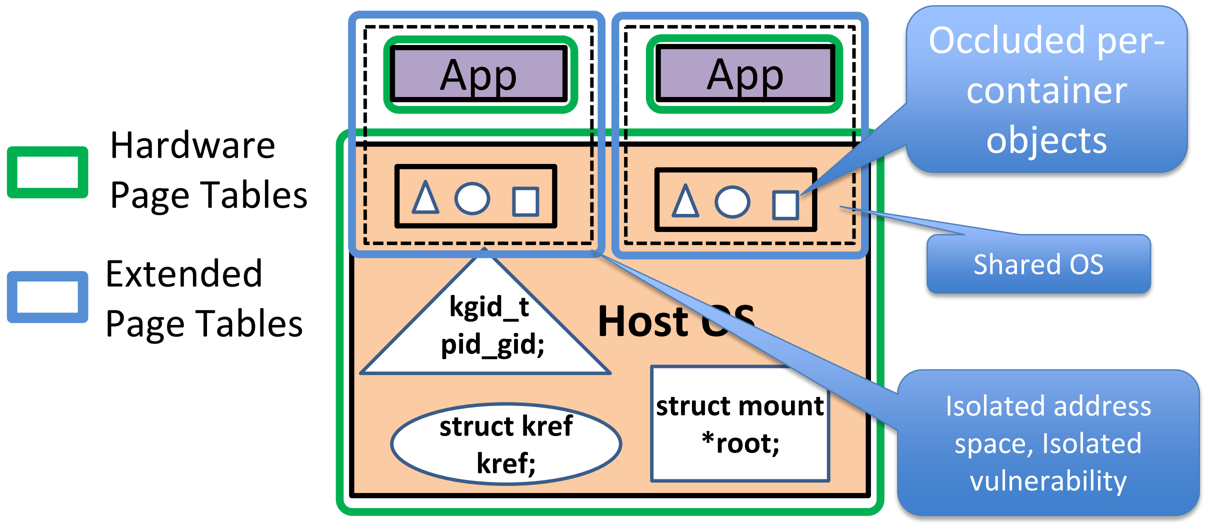


Figure 4.1: Hardware-Assisted OS Virtualization (Containers)

boot time for VM is orders of magnitude higher than containers as shown in Table 4.1. This overhead of booting a complete legacy OS for one application is why the cloud computing community is moving towards containers instead of VMs.

The key insight is that memory isolation for just the container-specific kernel objects can solve the problem of multi-tenant security for containers. Our solution repurposes the virtualization hardware to achieve this isolation. Our solution maintains the lighter weight of containers by sharing the OS kernel between containers and the host while providing memory isolation using a second set of page tables like EPT.

## 4.1 Overview

In the current Linux kernel design, containers are created by passing special flags to the `clone()` system call. The flags indicate that the kernel should create a new namespace, and the new process should use this new namespace. Namespaces, like file handles in Unix, can be selectively inherited from a parent.

Figure 4.1 shows the overview of the new design. This thesis makes the container a first-class object in the Linux kernel by creating all new namespaces for the new process, and protecting those namespaces using a *new* type of hardware-supported **memory namespace**. Hardware virtualization features like EPT help prevent access to container-specific kernel objects outside the memory namespace. As the hardware protection is at page granularity, this thesis redesigns the OS to be EPT page protection-friendly for containers. The memory allocators allocate container-specific objects at page granularity, so that the protection boundary can naturally map to the page translation and the page protection provided by EPT. The hardware delivers all the safe interrupts and exceptions directly to the guest, except a few unsafe interrupts like EPT fault, or triple fault. The unsafe interrupts cause a VMEXIT, so that they can be handled by the host. The host scheduler schedules which guest to run, but a second level of process scheduler inside the guest schedules the guest processes.

The contributions of this work are as follows:

- A redesign of the memory allocation of guest kernel objects to be EPT page protection friendly.
- Isolate containers with VM-like EPT-based protection.
- Leverage known techniques to lower overhead and improve performance in the presence of virtualization hardware.

**Threat Model.** This work follows the same threat model as a virtual machine. The host kernel is fully trusted, however, the code running in the guest mode is not trusted. An attacker can modify the kernel code running in the guest mode. Even in the presence of a compromised kernel in the guest mode, the guest cannot access information about any of the other co-located guests. The guest can only access data if the host has given explicit permission. A malicious guest cannot cause a denial of service for the co-located guests or the host. Any attempt made by a malicious guest to access other guest data will lead to the malicious guest's termination.

This thesis assumes that the code to set up the EPT permissions, which is part of the host kernel, does not have any vulnerabilities. As the code size is small (about 3K lines-of-code), the code can be formally or manually verified. Furthermore, this thesis assumes that the hardware does not have any vulnerabilities and our solution do not protect against hardware attacks. Side-channel attacks are out of the threat model.

Section 4.2 explains how the virtualization hardware provides memory isolation. Section 4.3 describes the changes in the kernel memory layout to use the virtualization hardware for isolation. Section 4.4 explains how to schedule different guests, and the different processes within those guests. Section 4.6 evaluates the memory footprint, startup time, developer effort, performance, and isolation property of our solution. Section 4.7 explains how our solution can be extended to support file I/O and network access for each guest in future. Section 4.9 discusses the related work, and Section 4.10 summarizes this chapter.

## 4.2 Hardware Support for Memory Isolation

In a native system, the hardware converts logical memory page numbers to physical memory page numbers, and caches the translation in a buffer called the TLB. Virtualization uses a second level of the page table called the Extended Page Table (EPT), that allows the host to set permissions that restrict guest access to pages. This thesis uses this feature to occlude parts of the host kernel from being accessible in the guest mode. This subsection will discuss in detail the need for EPT, and how the hardware address translation works.

The operating system maps logical memory page numbers (LPN) to physical memory page numbers (PPN) by storing the mappings in the page table. When a process of any application accesses the logical address of the memory, the hardware goes through the page table to determine the corresponding physical address location of where the data is actually stored. Frequently access translations are cached by the hardware system in the translation lookaside buffer also called the TLB. The TLB is a small cache on the processor, which accelerates the memory management process by providing faster LPN to PPN mappings of frequently accessed memory locations.

Virtual machines require 2 levels of page tables. As a virtual machine runs on a hypervisor, the guest OS cannot access the hardware page tables like the bare-metal OS. The hypervisor gives the guest OS an illusion that the guest OS is accessing actual physical page numbers by emulating the page tables for the guest OS. The hypervisor actually creates its own page table called Shadow Page table, which is visible to the system hardware. So whenever the guest OS requests for virtual address translation to physical memory address, that request is trapped by the hypervisor, which in turn uses its shadow page tables to provide the address of the physical memory location.

Hardware-assisted memory virtualization using EPT hardware walks the guest and the host page tables, and reduces the number of expensive VMEXITs. In presence of EPT, the TLB cache also keeps track of virtual memory and physical memory from the perspective of the guest OS. The TLB assigns each individual virtual machine an address space identifier for tracking the virtual machine address space without the need to flush the TLB cache on context switches.

The permissions in the EPT takes precedence over the access permissions declared in the guest page table managed by the guest kernel. Essentially, the hardware passes the read, write, and execute permission bits in the guest page table and the EPT through an AND gate, and uses the result to determine whether to allow the operation or not. So, if the guest page table mapping sets write permission on a particular page, but the host EPT disables the write permission on that page, the write access request from a guest to that page is denied.

The EPT mechanism lets the host set superseding permissions on pages accessible to guests. Our solution leverages superseding permissions to enable EPT tables for container processes, and only allows access to the kernel objects belonging to that particular guest container process.

## 4.3 Memory Layout Redesign to Leverage Hardware Support

There is a mismatch in granularity between the page-based EPT isolation mechanism, and the subjects of isolation, which are instances of kernel data structures smaller than a page. The Linux kernel uses a cache allocator to allocate memory for different kernel objects. A malicious guest container that has compromised the host kernel can use simple pointer arithmetic to easily locate and access kernel objects belonging to other co-located guests. As the hardware memory protection used to isolate guests works on a page boundary, to correctly isolate kernel objects and the memory of each guest, our solution modifies the cache allocator to allocate memory for objects of different guests from different pages.

Allocating and freeing data structures is a common operation inside any kernel. The Linux kernel uses the cache allocator to allocate memory for different kernel objects. The cache allocator works on certain observations:

- It is better to pool frequently used data structures, because these data structures are usually allocated and freed often.
- Frequent allocation and deallocation can cause memory fragmentation.
- The next allocation can immediately use a freed object.
- The allocator can make more intelligent decisions, if it is aware of concepts such as object size, page size, and total cache size.

The cache allocator divides different objects into groups called caches. Each cache stores a different type of object. Thus, there is one cache per object type. For instance, one cache is for inode objects (`struct` inode), whereas another cache is for process descriptors (a free list of task_struct structures). The kmalloc() interface uses a set of global caches for different sized objects called the memcache array.

The caches are further divided into slabs. The slabs are composed of one or more physically contiguous pages. Typically, slabs are composed of only a single page. Each cache may consist of multiple slabs. Each slab contains a number of objects, which are instances of the data structures. When some part of the kernel requests a new object, the request is serviced from a partially full, or else an empty slab. This strategy reduces fragmentation.

A malicious guest container that has compromised the host kernel can use simple pointer arithmetic to access objects of the same type belonging to other containers. The problem with using caches for allocating container objects is that all objects of the same type are allocated on the same page. This violates the complete mediation security principle.

Our solution redesigns the cache allocator so that each guest object is allocated from its own separate cache, called *memory namespace*. As a result, the object locations for each guest are separated on the page boundary. However, the kmalloc kernel interface allocates objects using the global memcache array for objects of different sizes. To isolate these dynamically allocated objects, our solution creates a separate copy of the memcache array for each guest in its memory namespace.

In addition to the dynamically allocated objects, our solution also needs to isolate the statically allocated data for each container located in the data section. In order to use the page-level protection, our solution needs to consolidate all the data of a specific container on the same set of pages. Our solution uses a per-CPU-like macro code to place per-container data on a separate page. Our solution introduces a new macro called per-CON. The kernel developers can declare and define per-container objects using the macro DEFINE_PERCON, and access these per-container objects using the macro PERCON. The host allocates separate per-container page(s) for each container and map these new pages for each new container in the EPT at the same per-CON page address.

The host cannot trust the kernel mapped in the guest address space to correctly allocate memory only from its designated area. So, our solution developed a two-layered memory allocator such that when the guest needs to allocate memory, the guest memory allocator checks if it has space in the pages assigned to it. If not, the guest memory allocator requests more pages from the host memory allocator, which in turn allocates pages from the memory assigned to the specific guest.

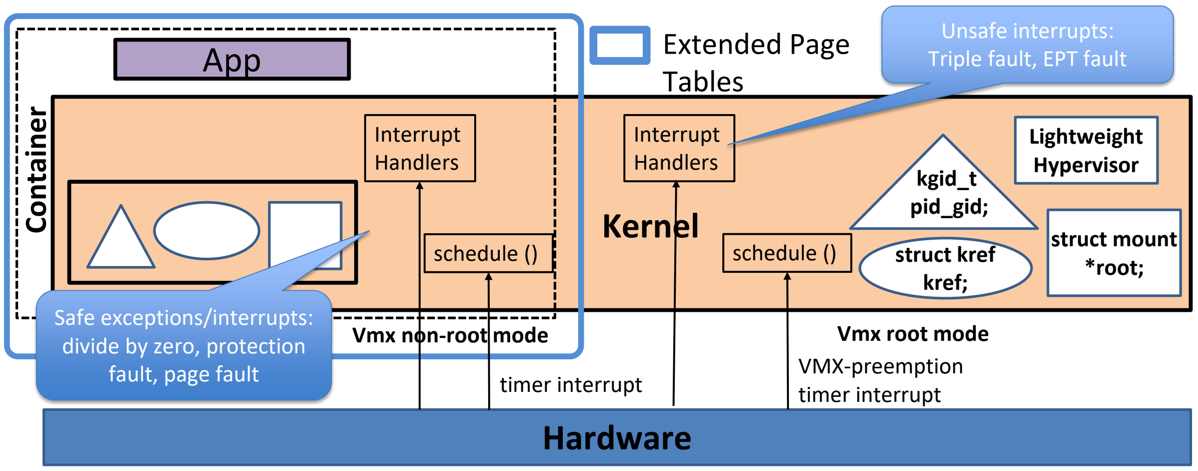


Figure 4.2: Interrupt delivery to host and guest

Thus, in our solution, all of a container's dynamically created kernel objects are densely packed into pages, shared only with other objects for that container. For static structures, the solution leverages link-time directives, such as macros, to make the implementation effort trivial. This ease of retrofitting comes from following existing coding conventions and design patterns, and any coding mistakes will be made apparent by a hardware fault, thus achieving simple design and fail-safe defaults principles.

## 4.4 Scheduling and Interrupts

Linux, like most OSes, uses the timer interrupt to implement the preemption needed for preemptive multitasking. To improve performance and avoid VMEXITs, our solution delivers safe interrupts and expections directly to the guest. In order to ensure that a malicious guest cannot affect denial of service for co-located guests and the host, our solution needs a way to periodically preempt the guest. In our solution the host and the guest modes are separated by the virtualization hardware. So, when a guest is running, the hardware delivers the timer interrupt and other safe interrupts and exceptions directly to the kernel in the guest mode using the APIC Virtualization extension. The guest kernel uses this timer interrupt to switch between different processes of the same guest. To maintain the fairness of execution time and avoid denial of service attacks, our solution uses a special VMX-preemption timer interrupt, which causes a VMEXIT and is delivered to the host. The host can then schedule the next guest or a host process.

In the current Linux design, the host schedules all processes including the containerized guest processes. Containers isolate processes in the guest using the PID namespace. The process in the guest gets a different

PID when inside the container and another PID when outside the container. The PID namespace maps the 2 PIDs inside and outside the guest. The host gets to see all the guest and host processes. The host schedules the guest as well as host processes using the same scheduler. If the selected process is a containerized process, the host runs the corresponding guest process.

The maximum timeslice of each host, as well as the guest process, is determined by the timer interrupt. For containers, as both the guest and the host share the same kernel irrespective of whether the guest or the host process is running, the interrupts are handled by the host. In case of the timer interrupt, the host preempts the running process and selects a new process to be run next based on its scheduling policy.

In our solution, even though the guest and the host share the same kernel, they are separated by the second-level page tables. The *task_struct* of the guest processes in its PID namespace is allocated from a separate slab, which is not mapped or directly accessible in the host. For the host to understand and control what exactly is going on in the guest, the host will have to use expensive techniques such as virtual machine introspection. This would defeat the purpose of using containers.

To solve this issue, our solution adds a second-level scheduler in the guest. Even though the host can't schedule individual guest processes, the host can still schedule the host process corresponding to a container. Once the container process starts running, the first thing it does is check for interrupts or exceptions to be handled, and then the guest scheduler chooses one of the guest processes to be run. However, now the problem is, if the timer interrupt arrives, whether the guest or the host gets to handle that interrupt. If the guest handles this interrupt, a malicious guest may never give control back to the host. On the other hand, if the host handles the timer interrupt and preempts the guest, the guest can only run one process in its time slice, thus affecting the performance.

Our solution trades off the performance and fairness using a special VMX-preemption timer interrupt that traps out of the guest after a set amount of time. The standard timer interrupt is delivered and handled in either guest or host, whichever is running when the timer interrupt arrives. The VMX-preemption timer is initialized to a value that is a multiple of the timer interrupt. So, if the guest is running a few CPU-bound processes, the guest scheduler can switch to other guest processes on receiving the standard timer interrupt. When the preemption interrupt arrives, it gets delivered to the host, which preempts the guest, and runs another guest or host process. The preemption timer interrupt ensures that the guest doesn't monopolize all the CPU time, but can run multiple processes before having to give control back to the host, thus not sacrificing performance. As the preemption timer interrupt is a feature of the virtualization hardware, the interrupt is only as expensive as a VMEXIT, which is the cost of context switching between the guest and the host.

As our solution shares the same kernel in the host and the guest, the guest does not need a separate interrupt descriptor table (IDT), which maps interrupt numbers to their respective handling functions. The same code handles the host, as well as guest, interrupts in respective modes. But, the guest cannot be allowed to handle all the interrupts, especially the hardware interrupts, and highly important ones such as triple fault and EPT fault. Intel's virtualization extensions allow the host to control whether interrupts are routed to the guest or host during guest execution, using a structure called the VM control structure (VMCS).

Intel's Advanced Programmable Interrupt Controller (APIC) is used for sophisticated interrupt redirection, and to send interrupts between the processors. Intel has introduced APIC Virtualization extension (APICv) to improve performance by speeding up the interrupt delivery to the guest. APICv is built on top of x2APIC, which is available in modern CPUs. This virtualization extension allows inter-processor interrupts to be delivered to the guest without causing VMEXIT. APICv creates a Virtual-APIC Page to enable virtualized APIC read and write accesses without causing VMEXIT. The hardware checks for notification of interrupts. If the notification is present, the hardware updates the guest interrupt status in VMCS. On the next VMENTRY, the guest will know about the interrupt, and the guest will jump to the handler while in the guest mode.

Thus, in our solution, the hardware is configured to deliver the timer interrupt and other safe interrupts and exceptions directly to the kernel in the vmx non-root mode, and deliver the VMX-preemption timer interrupt, and unsafe interrupts to the vmx root mode for security and fairness. Adding a guest-level scheduler increases the memory footprint slightly, but the guest scheduler is small and light-weight. Our solution leverages performance optimization features of the virtual hardware interrupt delivery to efficiently deliver interrupts to the vmx root and non-root mode.

## 4.5 Implementation Details

Our solution modifies the Linux kernel version 4.1.6 and supports running on Intel x86 processors in 64-bit long mode. The engineering needed to support AMD CPUs is straightforward. Our solution builds on the virtualization hardware configuration code from Dune [48]. However, Dune uses a library OS to run guest userspace code in the kernel in vmx non-root mode. In contrast, our solution only runs the kernel code in kernel mode in vmx non-root mode and runs the guest userspace code in the userspace in vmx non-root mode.

Our solution also configures the virtualization hardware to deliver safe interrupts and exceptions directly to the kernel in vmx non-root mode to avoid unnecessary VMEXITs as discussed in Section 4.4.

In our solution, guest containers are created in the same way as before — by making a `clone()` system call — except for the additional `CLONE_NEWMEM` flag. If the `CLONE_NEWMEM` flag is passed to the `clone()` system call, our solution creates a new memory namespace for the child process, which initializes new kmem_cache pointers for processes in this new memory namespace. All the subsequent objects for the child process are allocated using these new cache pointers during the `clone()` processing.

Once all the data structures are created, if the new process is isolated in a new memory namespace, the `ret_from_fork` routine, which is the address used by any newly created process, calls a special routine to setup the VMCS data and the EPT tables for the new process, and then transitions to the vmx non-root mode, and starts running the userspace code in the vmx non-root mode. The EPT mappings are made with read and write permissions for the per-CON pages, the new kernel stack page for the guest, and all the memory allocated in the new memory namespace. The EPT mappings are read-only for the kernel text and follow the well-known virtual memory map of a process as explained in the kernel documentation [7].

If anything causes a VMEXIT in the vmx non-root mode, like a hypercall, the VMX-preemption timer interrupt, an unsafe interrupt, a bug, or an attack, the control comes back to the special routine, which then determines the reason for the VMEXIT and deals with it appropriately. The invariant enforced is that if any change is required to shared data or objects, the vmx non-root mode kernel does a VMEXIT to ask the kernel in the vmx-root mode to make the change, which will succeed only if the change requested is safe and fair.

The current implementation can create new memory isolated containers using the `clone()` system call, create multiple guest processes within that memory namespace, switch contexts among these guest processes, and handle system calls that don't need VMEXITs such as `getpid()`. The implementation of filesystem and network support is left as future work, which is discussed in Section 4.7. As this is not a complete solution, some of the evaluation results may change with a complete solution as discussed in Section 4.8.

## 4.6 Evaluation

The goal of our solution is to keep the memory footprint overhead and startup time orders of magnitude lower than a VM. To show that our solution is an economical solution to protect complex, legacy operating systems, this section measures the developer effort in terms of lines of code changed. This section also demonstrates the claim that the execution time will be proportional to the number of VMEXITs. And finally, this section demonstrates that an attacker running in vmx non-root mode cannot access the data of another co-located guest. Our solution is compared with the native `clone()` system call as the lower bound. Our solution offers security similar to a VM at lower costs. So our solution will be worse than native `clone()`; the goal is to be significantly closer to `clone()` than the best case VM. Our solution is also compared with a Firecracker MicroVM as the upper bound, because Firecracker MicroVM represents the state of the art in building a lightweight VM of a full Linux guest as demonstrated in Figure 4.6.

It is important to note that the results in this section are preliminary, because our solution is an incomplete prototype, which is compared to full-featured, production systems of Firecracker MicroVM. It is possible that the overheads of our solution will increase as new features are added, but these results are still encouraging because the differences are so pronounced, and there are known techniques to avoid VMEXITs to improve performance. So, the results of a full-featured prototype are not expected to skew a lot.

### 4.6.1 Memory Footprint Overhead

This thesis defines the memory footprint overhead of a platform as the number of new pages allocated by the kernel to start a guest using that particular platform. For containers, this is the number of pages allocated to make the clone system call. For VMs, the memory footprint overhead is the number of pages allocated by the host to start a new VM. This section uses the lightweight MicroVM on the Firecracker platform as a representative example for virtual machines. For our solution, the memory footprint overhead is the number of pages allocated to make the clone system call, including the pages allocated for the EPT table. For this experiment, the native cloned process as well as the process cloned using our solution, exits without running any instructions.

To measure the memory footprint overhead, the host initializes a counter in `task_struct` and increments the counter for every page allocated till `do_exit()` is called. The value of this counter at `do_exit()` is the memory footprint overhead. Table 4.2 shows the number of pages allocated for a container, VM, and our solution. Although our solution allocates 161x pages compared to a native `clone()` system call, our solution allocates about 40x fewer pages compared to a Firecracker MicroVM.

| | Native Clone | MicroVM | This Solution |
|---|---|---|---|
| Pages Allocated | 5 | 32731 | 809 |
| Normalized | 1x | 6546.2x | 161.8x |

Table 4.2: Memory footprint overhead for native clone, MicroVM, and our solution.

| | Native Clone | +/- | MicroVM | +/- | This Solution | +/- |
|---|---|---|---|---|---|---|
| Startup Time (ms) | 107 | 0 | 3597 | 30 | 1180 | 50 |
| Normalized | 1x | - | 33.61x | - | 11.02x | - |

Table 4.3: Startup time for native clone, MicroVM, and our solution.

| | Linux kernel | KVM | MicroVM | This Solution |
|---|---|---|---|---|
| Lines of Code | 17.3 M | 0.15 M | 65 K (0.065 M) | +/- 2845 (0.002 M) |

Table 4.4: Developer effort for Linux kernel, MicroVM, and our solution.

### 4.6.2 Startup Time

In addition to the low memory footprint, another benefit of using containers is the fast startup time. This subsection compares the startup time of our solution with the native `clone()` system call and the Firecracker MicroVM. For the native clone() system call and our solution, startup time is the time to run the first line of guest userspace code. For a Firecracker MicroVM, the startup time is the time to boot the VM. Table 4.3 shows results with respective standard deviation. Our solution takes 11x more time than a native `clone()` system call, but is 3x faster than the startup time of the Firecracker VM. The higher startup time of our solution is due to the time required for configuration of the EPT table and the virtualization hardware.

### 4.6.3 Developer Effort

One of the goals of this design is to keep the developer effort minimal so that it is economical to apply our solution to complex, legacy systems. Additionally, the fewer the lines of code added or changed, the easier it is for auditors to audit and verify the design and code changes. The redesign of the memory subsystem caused propagation of some mechanical changes throughout the code, such as using the memory namespace for kernel object allocation (§4.3). Table 4.4 shows the lines of code needed to implement our solution compared to the Firecracker MicroVM, KVM, and the Linux kernel. Our solution is orders of magnitude smaller than MicroVM and KVM.

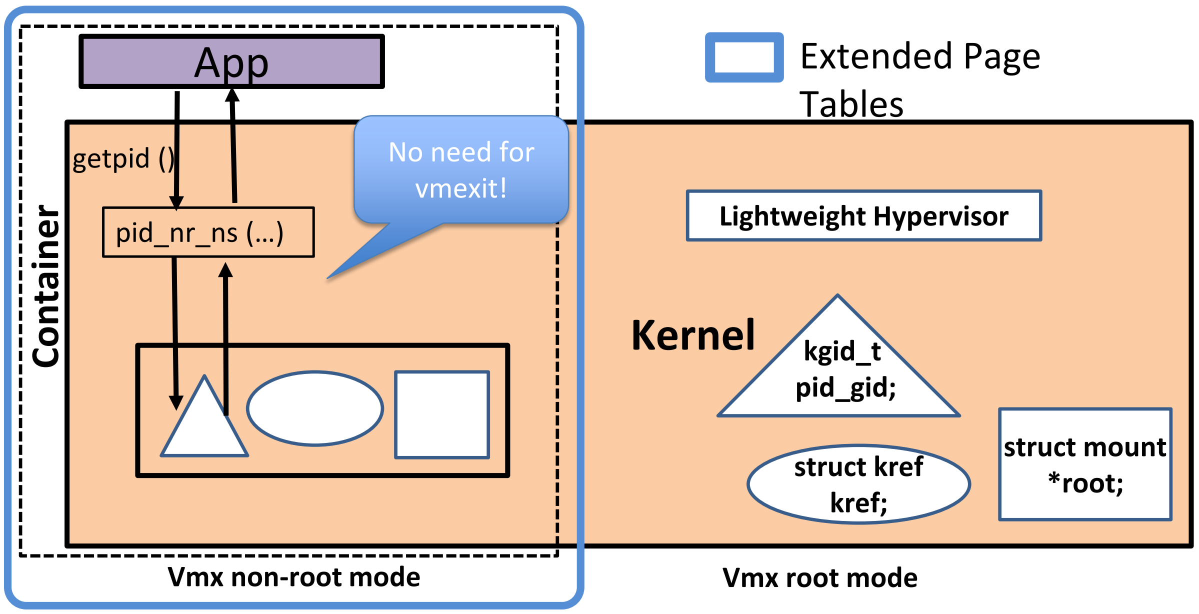


Figure 4.3: Reduce VMEXITs by servicing system calls in vmx non-root mode

### 4.6.4 Effect of VMEXIT on Execution Time

The cost of using the virtualization hardware is that every context switch between each guest and the host is the cost of handling a VMEXIT in the host. In order to improve performance, the goal is to avoid VMEXITs as much as possible, and only pay the price of VMEXIT if absolutely necessary. Our solution reduces VMEXITs by servicing most system calls in vmx non-root mode as shown in Figure 4.3. This subsection compares the performance of the `getpid()` system call in native Linux with our solution. The execution time for `getpid()` system call is the same in our solution as well as the native Linux because our solution services the `getpid()` system call in the vmx non-root mode, without the need to cause a VMEXIT.

According to the conventional wisdom, the fewer the VMEXITs, the better the performance efficiency. To verify this conventional wisdom, this work modified the implementation of the `getpid()` system call to make null hypercalls to the vmx root mode that returns without doing any computation. The plot of the execution time for `getpid()` against the number of null hypercalls made demonstrates the effect of VMEXITs on execution time as shown in Figure 4.4. The linear increase in the execution time shows that the execution time is directly proportional to the number of VMEXITs. Thus, the goal of this design is to reduce the number of VMEXITs needed, so that the performance is as close to the native performance as possible.

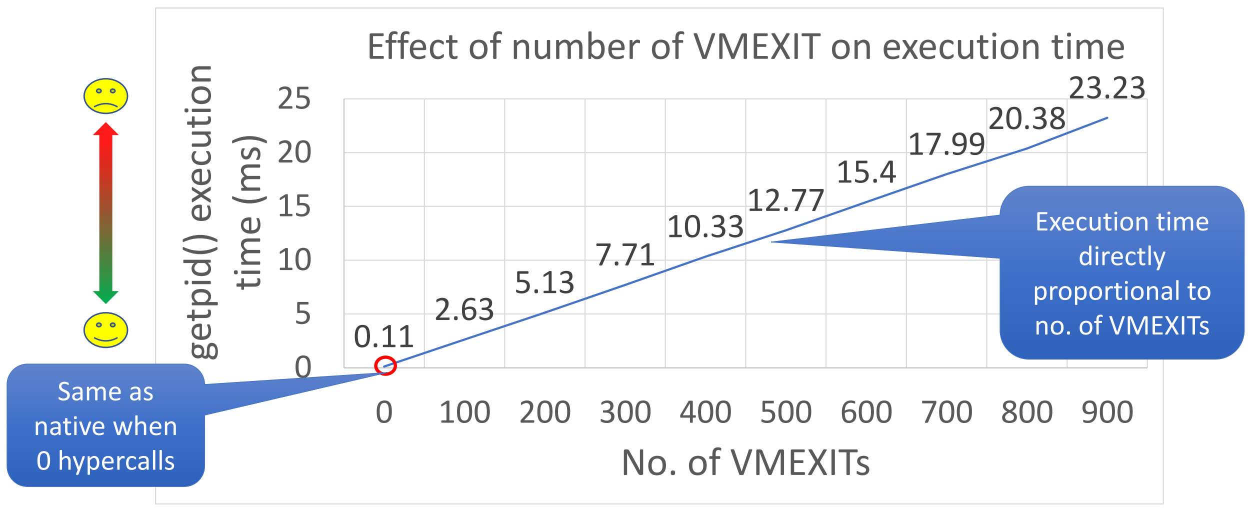


Figure 4.4: Effect of VMEXIT on `getpid()` execution time

### 4.6.5 Prevent Illegal Memory Access

One of the most important security invariants, based on the threat model, is that even if an attacker controls the guest, and leverages kernel vulnerabilities to compromise the kernel, the attacker in the guest still cannot compromise host or other co-located guests. Our solution achieves this property by only mapping the kernel data specific to a particular guest in its EPT table, and setting correct permission values in the EPT entries.

In order to verify that this protection isolates the host memory and co-located guest memory from a malicious guest, this work introduces a new system call `attack()` that takes an address and an operation flag to indicate read, write or execute operation to be performed at the given address. Effectively, just for verification purpose, a rootkit is added in the guest, that allows the attacker to read, write or execute arbitrary address. Whatever the goal of an attacker may be, the attacker can achieve it using this interface in the absence of our solution.

When the attacker passed an address to the `task_struct` page containing host data structures, as well as another guest data structure, any attempt to access these memory pages in the vmx non-root mode caused an EPT fault, and the control was transferred to the kernel in vmx root mode, which realized that the guest is trying to access a page that is not mapped in its EPT tables. As the page is not mapped in the guest EPT table, it meant that the guest was not supposed to have access to this page. Clearly, this was an attempt at

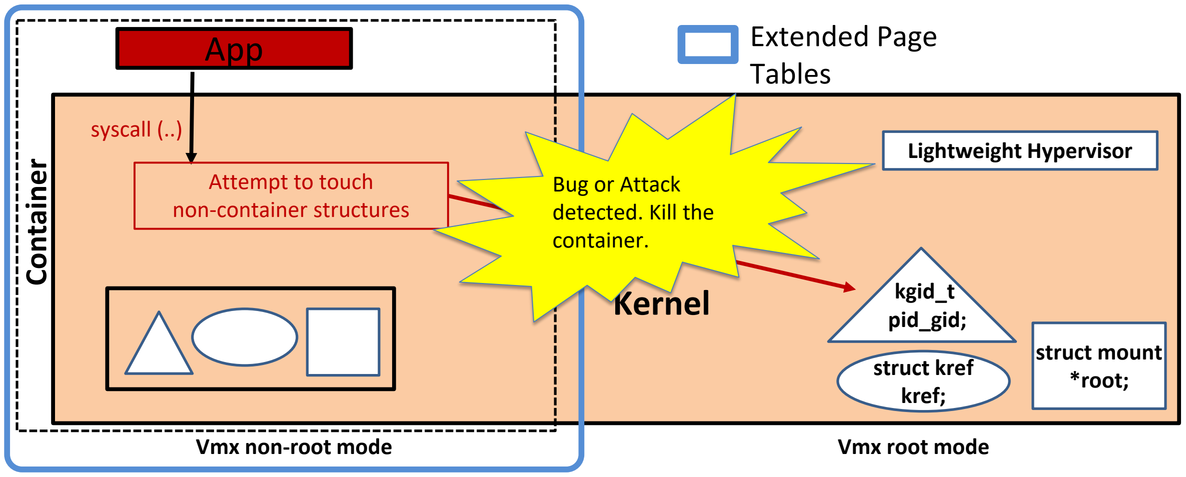


Figure 4.5: Our solution prevents illegal memory access by an attacker

breaking out of the memory isolation, and as a result, on recognizing this attempt, the host terminated the guest without having to even resume the execution of the guest as shown in Figure 4.5.

This thesis notes that, in our solution, the attack surface of the host consists of both memory mappings and the host API. Iago and semantic attacks [66] against this host API, and hardening that API in general are a potential concern, but out of scope for now, since this is of a smaller size than KVM's hypercall table.

### 4.6.6 Design Principles for Protection Mechanisms in This Solution

**Least Privilege:** In our solution, the kernel in the guest mode runs at a higher ring level than the host kernel, which means it has a lower privilege than the host kernel. Even though our solution shares the same kernel between the host and the guest, the guest can only read and execute the code. The guest kernel does not have the privilege to write to data objects belonging to the host or other guests. Our solution reduces the privilege of the kernel in the guest mode so that it can read or write data that belongs only to that guest. Our solution assumes that the small piece of code added to the host kernel doesn't have vulnerabilities.

**Economy of Mechanism:** This principle calls for a simple and small design. To that end, our solution only add about 5K lines of code, and use well-known techniques such as virtio and exitless interrupt delivery to reduce duplicating code. This design is already sharing the same kernel in the host and the guest.

The modifications that our solution did to the kernel are very modest. The layer of code that sets up the environment to use and manage EPT, slab allocator changes, and the ring buffer for I/O and network packets is very minimal.

**Complete Mediation:** This principle requires that every access to every object must be checked for authority. Our solution only maps the relevant pages of the guest in its EPT. As a result, the hardware automatically does the complete mediation.

**Open Design:** This principle requires the design to not be secret. The code, as well as design, is open-sourced. Beyond this, none of the security properties depend on secret information remaining secret.

**Separation of Privilege:** When applied to computer systems, this principle requires that two or more conditions must be met before access is permitted. Our solution has 2 levels of page tables. The userspace program running in the guest cannot access any memory unless it is explicitly allowed by the guest kernel page table and the host EPT for that guest. Thus even if the guest userspace program compromises the guest kernel, it cannot compromise the host and other guest memory due to the EPT mappings.

**Least Common Mechanism:** The least common mechanism requires minimizing the number of shared mechanisms to more than one entity. Virtual machines follow this principle well as the guest and the host have their own separate copies of the kernel. Containers on the other hand break this principle by design by sharing the same kernel code and data with all the guests and the host. This design allows sharing of code between the host and guest, but not the data objects. This reduces the number of shared mechanisms between the guest and the host, which may unintentionally compromise security. However, the small amount of new code that our solution added to the kernel is in the shared TCB for all the guests and the host.

**Psychological Acceptability:** This principle suggests that the human interface for the system should be a well-known specification. Our solution runs unmodified apps, so that the software developers do not have to change their interactions with the system. Our solution supports the creation of containers with the standard clone system call, and allow the guest programs to make system calls as in the case of containers. Our solution only changes the internal workings of the system so as to be backward compatible with existing mechanisms.

**Fail-safe Defaults:** This suggests basing the access decisions on permission rather than exclusion. The way EPT is configured, the host can set read, write and execute permissions for the guest at the page granularity. By default, each EPT entry is initialized so that the page is not accessible to the guest. Thus, any type of access by the guest is allowed only if the host has given explicit permission for that access.

## 4.7 Future Work

Our solution doesn't yet support filesystem and network. However, this section will describe the design for filesystem and network subsystem. The implementation of these subsystems is left as future work.

### 4.7.1 Filesystem Design

Containers use the FS namespace and a unioning file system to isolate files on the host file system, while the host for VMs emulate a separate virtual disk for each guest, which is accessed as a block virtio disk device by the guest kernel. Our solution supports the standard mechanism for file system in containers of unioning file systems and FS namespace, but to avoid giving the guest mode kernel full access to the host disk, our solution creates a separate ring buffer for each guest, and allow guests to access the file system using file-based abstraction and the POSIX interface over the virtio protocol. This subsection describes in detail how our solution plans to support file I/O in the guests.

Containers isolate the guest file systems using FS namespace and a unioning file system such as AUFS. AUFS projects to the guest a unified view of the files, which may be scattered over the host. As the entire kernel is mapped in each container guest, all guests have access to AUFS driver, which in turn forwards the requests to the corresponding underlying file system, such as ext4 or btrfs. Thus, guest containers can directly access the disk contents through this indirection. The abstraction used by the guests is file-based and POSIX interface.

Virtual machines, on the other hand, have a separate kernel in the guest, but the disk is managed by the host kernel. The guest cannot directly access the files on the host's disk. VMs solve this problem by having the host emulate a virtual disk device to the guest, which is managed by a virtio block driver in the guest. The guest finds this device during discovery at boot time, and registers the device for block io, which is used by the guest file system. The virtio driver uses a mechanism called vring to communicate with the host emulator. The virtio driver forwards the block io requests to the host, and the host forwards those requests to the disk device driver. Once the disk driver returns the data to the host emulator, the emulator forwards the data back to the virtio driver, and the virtio driver forwards it to the guest file system. Thus, the abstraction used by the guests is block-based and the interface is block requests over a vring.

Our solution can't just use the virtio system of virtual machines as is, because a) like containers, guests in our solution do not go through the boot process and device discovery, and b) the virtual machine virtio system is block-based, whereas containers use file-based abstraction and the POSIX interface to access files. In order to solve this problem, our solution plans to modify the virtio infrastructure to fit the design. At the launch of the container, our solution will create 2 vrings for each guest to keep track of the requests and responses from the guest. This design will maintain the container's file-based abstraction and the POSIX interface, but over a vring like in the case of virtual machines. However, since our solution does not have a virtual device and its driver, this design will need to create the wrappers around the vring interface in the guest and the host.

The guest side wrapper will register itself as a file system in the guest. When the guest userspace makes file system calls, those calls get passed on to the VFS layer, which forwards it to the guest side-wrapper. The guest-side wrapper then creates a request message, adds it to the vring, and makes a hypercall to the host for it to service this request.

The host-side wrapper handles this hypercall, checks to see if there is a pending request, and if there is a request, forwards it to the VFS layer of the host. Just like containers, this design plans to use the file system namespace and AUFS to project a unified restricted view of the file system. So the VFS layer then forwards the request to the AUFS driver in the host, which in turn services the request and returns the response back through the call-chain to the host-side wrapper. The host-side wrapper creates the response message, adds it to the response vring, and injects an interrupt in the guest. This interrupt is handled in the guest by the guest-side wrapper. The guest-side wrapper collects the response and returns it back to the guest userspace through the call chain.

The level of abstraction for our solution is files for efficiency, so that there is no need to duplicate the block device drivers in the guests. All interactions between the guest and the disk are mediated for security, but can be asynchronous for efficiency.

### 4.7.2 Network Design

Network support in VMs as well as containers is a solved problem. This subsection plans to leverage known solutions, and adapt the solutions to fit the security requirements of the threat model. For any guest to communicate with the host and the outside world, the host needs to provide special networking support. Containers, as well as Virtual machines, have two mechanisms for this communication: a) single-root input/output virtualization (SR-IOV) and b) virtual network. SR-IOV is hardware-assisted virtualization, whereas virtual network is software-only virtualization. In the case of SR-IOV, each guest gets its own virtual function, which acts as an ethernet device. In absence of SR-IOV, the host passes packets to guests using another ring buffer and virt-net protocol, similar to the file system.

The SR-IOV is a specification that allows a PCIe device to appear as multiple separate physical PCIe devices. A PCIe device, such as an Ethernet port, that is SR-IOV-enabled with appropriate hardware and OS support can appear as multiple, separate physical devices, each with its own PCIe configuration space. The SR-IOV enabled PCIe device advertises physical functions (PFs) and virtual functions (VFs) to configure and access these network devices. PFs have the ability to configure or control the PCIe device via the PF, and the ability to move data in and out of the device. VFs are similar to PFs but lack the configuration ability; VFs only have the ability to move data in and out. Each SR-IOV device has a PF and each PF can have multiple VFs associated with it. The VFs are created by the PF. After SR-IOV is enabled in the PF, the PCI configuration space of each VF can be accessed by the bus, device, and function number of the PF. Each VF has a PCI memory space, which is used to map its register set. The VF device drivers operate on the register set to enable its functionality and the VF appears as an actual PCI device. So, to use SR-IOV in containers or virtual machines, the host needs to configure the SR-IOV enabled device and set the number of virtual network adapters, which are accessed using respective VFs. The guest needs to configure its network stack against these VFs.

In case SR-IOV is not supported by the PCIe device, the host connects the guest and the host using a virtual network. For virtual machines, the host exposes a virtual PCIe device to the guest. As the guest and the host have a separate copy of the OS, they instantiate and manage their own network stacks. At boot time, after discovering the PCIe device, the guest configures its network stack against this device, and initializes the virt-net driver to access this device. The virt-net driver functions similarly to the virt-io driver. It uses a ring buffer to pass network packets to and from the host. Thus when a packet arrives at the host, the host forwards that packet to the guest using the ring buffer and raises an interrupt in the guest to handle the packet. Similarly, when the guest wants to send a packet, the virt-net driver forwards the packet to the host using the ring buffer, and makes a hypercall for the host to handle the packets.

In the case of containers, in absence of SR-IOV support, the host creates a virtual ethernet (veth) to connect the host and the guest. veth can be considered as a virtual network cable with connectors at both ends. The guest network stack can be isolated from the host and the other guests using the network namespace. The host moves one end of the veth to the guest network namespace, and the other end is connected to the host network stack. Often the host end is connected to a bridge created in the host so that all the guests can communicate with each other and the host. Unlike VMs, the container guests share the kernel, but the network namespace instantiates a separate network stack for the container configured with its end of the veth.

Our solution plans to support the standard mechanisms for networking in containers. If a SR-IOV enabled network adapter is present, the guest network stack is configured with the virtual functions exposed by the device. Our solution also plans to change the driver for these devices so that the objects for a particular guest are allocated from its memory namespace, which is isolated from the host and other guests by the second-level page table. If SR-IOV support is missing, our solution plans to use the veth to connect the host and the guest. However, our solution plans to change the veth driver to create the objects for each container endpoint in a separate page, which are then mapped into that guest's address space using EPT. Also, as the guest and host are isolated by memory namespace, our solution plans to add a transmit and receive routine in the veth driver code. Our solution creates another ring buffer for network packets, just like the file system, and then plans to use the same mechanism described above to transfer the network packets between the host and the guest.

Thus, our solution plans to adapt well known solutions for network support in VMs and containers. The networking pass-through for guests has rapidly evolved over the last 10 years in a way storage has not, especially as hardware like SR-IOV, and user-space solutions like Arrakis [161], IX [49], Snap [148]. Our solution plans to use these well known techniques to isolate the packets sent and received by the guest, and enforce complete mediation either in the hardware or the vmx root mode kernel.

## 4.8 Discussion

VMs can use a technique called kernel Same-page Merging (KSM) to increase the density of guests on a particular host, and use snapshots to speed up the startup costs of VMs [37]. Although these are similar goals to our solution, our solution's approach can reduce the memory footprint of a guest more than the gains from using KSM because our solution requires more fine-tuning compared to the coarse KSM approach.

As an incomplete prototype is used for evaluation in Section 4.6, some of the results may change in the future after the implementation of the filesystem and network support. This thesis expects an increase in the number of lines of code, commensurate with the order of magnitude. Adding support for the filesystem and the network may cause more VMEXITs, and slow down the performance of the system. In such a case, more engineering effort will be needed to fine-tune the system to avoid VMEXITs as much as possible. Changing the memory layout of some of the objects specific to the guest may have idiosyncratic effects on cache hit ratios, which needs to be further analyzed, but is also left as future work. As the prototype is incomplete, this thesis cannot definitively declare the problem as solved. Instead, this thesis claims this work is a significant step in the direction of achieving the best of both the container and the VM world.

## 4.9 Related Work

This section discusses related work divided into 4 major groups. The first group discusses the techniques to isolate drivers from the core kernel. Next, this section describes different options to virtualize the supervisor mode or the ring 0 in the case of Intel x86. Then this section discusses other research exploring the design points on the spectrum between VMs and containers. The section concludes with a collection of other research works related to the solution described in this chapter.

### 4.9.1 Isolating Drivers from the Core Kernel

There is an area of research focused on isolating the faults in the drivers from the core kernel. This isolation is achieved by using one of the following five techniques as also discussed by Herder et al. [111].

#### 4.9.1.1 Kernel Wrapping

First, wrapping and interposition are used to safely run untrusted drivers inside the OS kernel. In this technique, the idea is to interpose and verify all parameters on calls between the core kernel and device drivers. For example, SafeDrive [213] uses wrappers to enforce type safety constraints and system invariants for extensions written in C. Nooks [190] combines in-kernel wrapping and hardware-enforced protection domains to trap common faults and permit recovery. Similarly, the solution in this chapter plans to interpose and verify the I/O and network syscalls, and only allow access to the files or packets that belong to the container.

#### 4.9.1.2 Virtualization

Data corruption problems, which are one of the most common driver faults, can be isolated using Virtual memory protection. However, this technique can't catch deadlock errors caused by improper disabling of interrupts. Virtualization can be used to run services in separate hardware-enforced protection domains.

Examples of virtual machine (VM) approaches are VMware [189] and Xen [95]. However, the driver faults can still propagate and crash the core OS. Instead, each driver needs to run in a para-virtualized OS in a dedicated VM [134]. The client OS running in a separate VM accesses its devices by issuing virtual interrupts to the driver OS. The solution in this chapter leverages virtualization hardware to provide memory isolation for guests.

#### 4.9.1.3 User-mode Drivers

One can prevent drivers from accessing privileged address space, executing privileged instructions, and corrupting the kernel, by lowering the privilege level of drivers to user level. However, this lowering of privilege level causes an additional trap and return to change privilege level for every call into the driver, thus incurring a huge performance penalty. MINIX 3 [110] encapsulates untrusted drivers in a private address space of user-mode process. Mach [102] experimented with user-mode drivers directly linked into the application. L4Linux [107] runs drivers in a para-virtualized Linux server. SawMill Linux [99] focuses on performance rather than driver isolation. User-mode drivers were also used in commodity systems such as Linux and Windows. By using the virtualization hardware, the solution in this chapter effectively runs the guest kernel at a lower privilege, but not in the userspace.

#### 4.9.1.4 Software Fault Isolation (SFI)

Software fault isolation (SFI) such as VINO [64] provides similar the benefits to a privilege level change, but is difficult to implement for non-contiguous range of addresses. It is cheaper to call into and out of SFI code as compared to lowering the privilege level, but SFI code executes slower. Additionally, SFI does not easily support recovery using copy-on-write, which is supported by hardware memory protection. XFI [91] combines static verification with run-time guards for system state integrity and memory access control. SFI technique is orthogonal to our solution.

#### 4.9.1.5 Language-based Protection

Some solutions use language-based protection and formal verification to isolate drivers. OKE [54] instruments an extension's object code according to a policy corresponding to the user's privileges using a customized Cyclone compiler. Singularity [116] combines type-safe languages with protocol verification and seals processes after loading. Redleaf [158] uses only type and memory safety of the Rust language for isolation instead of the hardware address space. Tock [135] relies on Rust's language safety for isolation of a collection of device drivers from the kernel. The seL4 kernel [128] formally verified the microkernel by mapping the design onto a provably correct implementation. Devil [153] enables low-level code generation and consistency checking. Dingo [173] reduces concurrency and formalizes protocols to simplify the communication between drivers and the OS. Although the solution in this chapter has not been formally verified, this thesis posits that it will be easier to do so because of its small size.

### 4.9.2 Virtualizing Ring 0

Some related works isolate a small piece of a Linux kernel from the rest of the kernel, and rely on different hardware protection mechanisms to enforce the isolation. These techniques effectively de-privilege part of the Linux kernel.

The PerspicuOS system [85] virtualizes the ring 0 hardware privilege level in x86 architecture by turning on the Write-Protect Enable (WP) bit in the CR0 register, which enforces read-only policies even on supervisor-mode writes. The OS is divided using a nested kernel architecture that uses a small isolated trusted piece, to remove the privilege from the rest of the kernel by enforcing memory access control. Even though the entire operating system, including untrusted components, operate at the highest hardware privilege level, it is de-privileged by the nested kernel such that the nested kernel manages all the page-tables, and configures all the MMU mappings to page-table pages as read-only. Thus, PerspicuOS de-privileges outer kernel code by replacing instances of writes to CR0 with invocations of nested kernel services and restricting outer kernel code execution to validated, write-protected code.

SKEE [43] is another system that provides an isolated lightweight execution environment at the same privilege level of the kernel for the ARM architecture. The memory regions used by SKEE are carved out of the memory ranges accessible to the kernel. SKEE also instruments the kernel code to remove certain MMU control instructions that may change the memory translation tables. Thus, SKEE prevents the kernel from managing its own memory translation tables, and forces the kernel to switch to SKEE to modify the system's memory layout. The switch to SKEE passes through a carefully designed switch gate so that its execution sequence is atomic and deterministic. SKEE in turn verifies that the requested modification does not compromise the isolation of the protected address space.

DIKernel [147] is another sandbox for kernel extensions by inserting a layer of indirection. DIKernel is implemented by inserting a layer between the base kernel and its extensions. It intercedes all interactions between the extensions and the kernel to enforce isolation and compatibility with existing extensions. This special layer called DI-switcher has three roles. First, it enforces memory access isolation between the base kernel and the extensions. The second role of the DI-switcher is interposition, it supervises all inter-domain control transfers and provides a secure interface between base kernel and extensions. The third role of the DI-switcher is to ensure compatibility with the existing extension code. Redirecting the kernel APIs by hooking the kernel symbols, and changing the stack, heap into the isolated memory region happens transparently without altering any existing codes or logic of the extension.

Another approach to isolating parts of the Linux kernel is to use compiler techniques. Virtual Ghost [81] combines compiler instrumentation and run-time checks on operating system code, to create ghost memory that the operating system cannot read or write. Virtual Ghost then interposes a thin hardware abstraction layer in this ghost memory between the kernel and the hardware, that provides a set of operations that the kernel must use to manipulate hardware. Even though the hardware abstraction layer runs at the same privilege level as the rest of the kernel, unlike other solutions, Virtual Ghost uses compiler techniques rather than hardware page protection to secure its own code and data. The OS is not able to access the secure ghost memory pages of the hardware abstraction layer as all the OS code is first passed through LLVM bitcode form and translated to native code by the Virtual Ghost compiler. During this phase, the compiler instruments memory access instructions in the kernel, and adds CFI checks to prevent by-passing the instrumentation. Virtual Ghost also extends the MMU configuration instructions in the hardware abstraction layer to ensure that a new MMU configuration does not make ghost memory accessible to the kernel.

On the other hand, the solution in this chapter uses the x86 virtualization architecture to isolate the guest-specific parts from the rest of the host kernel. The kernel memory layout redesign techniques used by our solution to isolate the guest and the host kernel are orthogonal to how the isolation is enforced, whether by virtualizing the ring 0 in hardware like x86 vt-d, in software like the PerspicuOS or SKEE, or by using compiler techniques like Virtual Ghost.

### 4.9.3 Design Points on the Spectrum Between VMs and Containers

Recently, researchers have started exploring different design points on the spectrum of VMs and Containers to bridge the gap between VMs and containers in terms of security. New solutions are using library OS, unikernel, or running a lightweight virtual machine. While these virtualization techniques try to reconcile the security and resource efficiency of containers and virtual machines, they have to build new runtimes

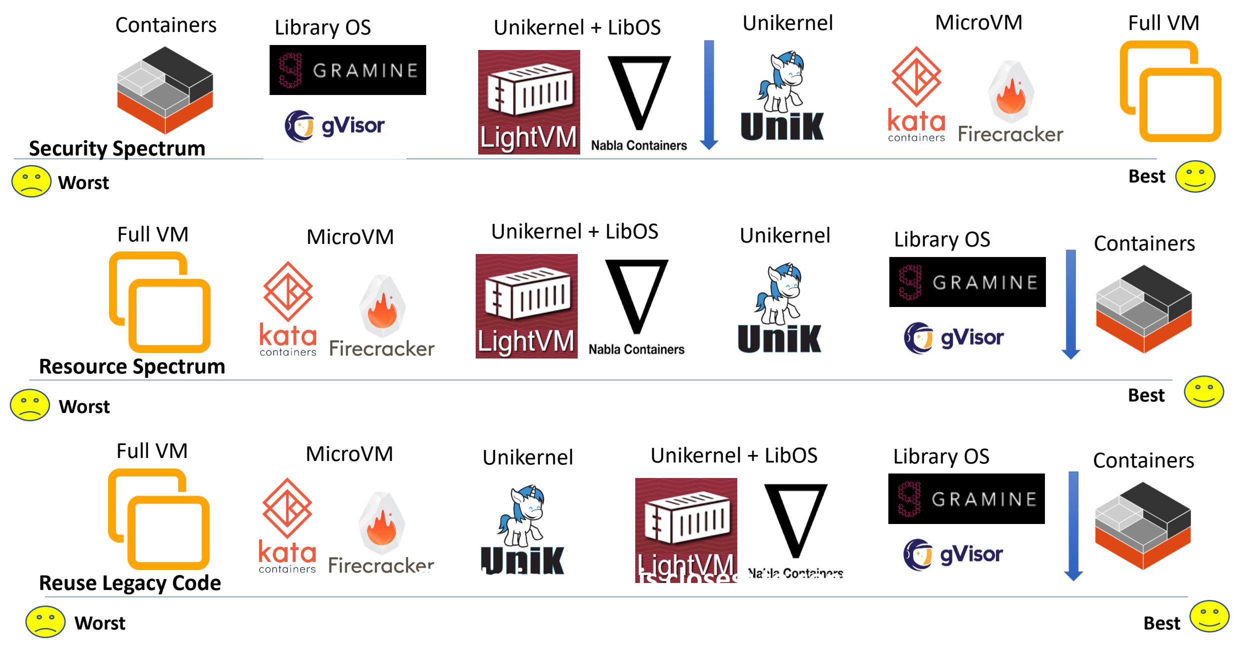


Figure 4.6: Container technologies on a spectrum for security, resource consumption, legacy code reuse. The blue arrow shows the design point on the spectrum explored by our solution.

from scratch, or still use a lighter weight VM, or use the unikernel and Library OS to isolate the guests from each other and the host. The solution in this chapter on the other hand explores a different design point on the spectrum where instead of recreating a new image for every single application, the design leverages the variety of features available in a stock Linux kernel, while solving most of the same problems as these different virtualization techniques. As shown in Figure 4.6, the design point in this chapter is towards the right side for most of the metrics compared to these virtualization techniques.

Manco et al. [146] use compiler techniques to build a lightweight VM targeted for specific applications using unikernels. The LightVM solution provides faster boot-times by redesigning the Xen's control plane. They reduce the interactions with the hypervisor to the minimum by transforming the centralized operation of Xen's control plane to a distributed one. In addition, they use Tinyx, a compiler tool to create tailor-made, trimmed-down Linux virtual machines for a given application. This tool creates unikernels by directly linking a minimalistic operating system such as MiniOS with the target application. The resulting VM is typically only a few megabytes in size and can only run the target application. This approach moves the OS in the userspace, and only links pieces of the OS components needed by that specific application.

IBM’s Nabla containers [204] run a unikernel application as a process on a specialized virtual machine monitor. They replace the general-purpose monitor, like QEMU, with Nabla Tender, which is a unikernel-specific monitor. The idea is to improve security by reducing the number of allowed system calls. Nabla Tender translates the hypercalls made by the unikernel into system calls, and uses a seccomp policy to block other system calls that are not needed. Nabla Tender combined with a unikernel runs as a userspace process on the host, and uses less than seven system calls to interact with the host. Nabla Tender acts as a library OS to run the unikernel.

Google uses gVisor [61] in CloudML, Cloud Functions, and Google Computing Platform’s (GPC) App Engine for sandboxing. gVisor uses Sentry, as a library OS, which is a userspace kernel implementation that intercepts and handles the system calls from applications to the host kernel. gVisor sandboxes an application from the host by creating a software security boundary, and restricting the system calls that an application can use. Sentry implements most of the kernel functionality, and only uses less than 20 Linux syscalls to interact with the host kernel. Both gVisor and Nabla use less than 10% of the Linux system calls to communicate with the host, and sandbox an application using a specialized guest kernel in userspace like a Library OS [196].

On the other end of the spectrum is the idea to use lightweight virtual machines by stripping off the unnecessary code and functionality in the hypervisor. AWS Lambda and AWS Fargate use Amazon Firecracker [36], which is a hypervisor that creates lightweight virtual machines (MicroVMs) specifically for serverless operational models in multi-tenant environment. Similar to the unikernel model, Firecracker provisions only a small subset of the emulated devices and functionality that are absolutely necessary for the guest operations, so that the microVMs have a much smaller attack surface, memory footprint, and boot time compared to traditional VMs. Firecracker provisions a maximum of four emulated devices for each microVM: virtio-block, virtio-net, serial console, and a 1-button keyboard controller used only to stop the microVM. MicroVMs do not share files with the host, instead the VM images are exposed to the guest using File Block Devices, and the network devices are built as tap devices over a network bridge.

Kata container [168] is another approach to build light-weight VMs by creating a highly-customized QEMU-KVM hypervisor, called qemu-lite, to achieve high-perfomance for VM-based container. QEMU-lite is a lightweight version of QEMU with 80% of devices and packages removed. Additionally, Kata container reduces the boot time by using VM-Templating, which snapshots a running Kata VM instance, and uses this snapshot to start new Kata VMs. Kata container uses Intel’s virtualization technology, similar to our solution, but only creates light-weight VMs.

### 4.9.4 Other Related Work

#### 4.9.4.1 Non-uniform Virtual Address Space

In the solution in this chapter, parts of the host Linux kernel are not accessible while running the guest context due to EPT mappings, which keep guests isolated from each other. Some systems isolate parts of a process from the rest of the process. Wedge [53] creates privilege separation and isolation among sthreads, which otherwise share the address space of the same Linux process. Wedge also helps developers refactor existing applications into multiple isolated compartments. Shreds [73] provide isolated compartments of code and data within a process using the architectural support for memory domains in ARM CPUs, a compiler toolchain, and kernel support. SpaceJMP [90] makes address spaces first-class objects separate from processes. In SpaceJMP, applications can use memory larger than the available virtual address bits allow, maintain pointer-based data structures beyond process lifetime, and share large memory objects among processes. SpaceJMP does not support applications that require isolation or privilege separation within a process, because a SpaceJMP context switch is not associated with a mandatory control transfer.

lwCs [141] provide in-process isolated contexts, privilege separation, and snapshots. lwCs are fully independent of threads, require no compiler support, and rely on page-based hardware protection only. Each lwC has its own virtual address space, set of page mappings, file descriptor bindings, and credentials. Within a process, a thread executes within one lwC at a time and can switch between lwCs.

#### 4.9.4.2 Containers as Packaging Instead of a Virtualization Technique

In addition to being a virtualization technique, Containers are also referred to as a packaging technique. First Docker, and more recently Kubernetes introduce toolchains that make managing and orchestrating the containers easier. These tools do not actually run the container, but handle the image building, orchestration, and management of the container instances. The Kubernetes Container Runtime Interface (CRI) defines an API between Kubernetes and the container runtime. The container runtime implements the Open Container Initiative (OCI) specifications for images and containers. The OCI also provides an implementation of the spec called runc, which is a tool for spawning and running containers. Thus, any new virtualization technique for containers just needs to implement their own OCI compliant runtime which can communicate with Kubernets using CRI, and Kubernetes can manage the orchestration of containers on that virtualization technology.

#### 4.9.4.3 Leveraging Virtualization Hardware for More Than Virtual Machines

Dune [48] pioneered the new idea of a kernel applying VT-x to individual processes, rather than a VMM applying VT-x to entire guest operating systems. Dune thereby gives processes direct access to privileged hardware, which is leveraged by IX [50] to improve the I/O performance for network packets. IX separates the data and control plane for the TCP/IP stack, and moves the control plane to the userspace, which manages the network devices, made manageable in the userspace using Dune.

The solution in this chapter also uses the same idea of applying VT-x to individual containers, which are seen as just another process by the host. However, the goals of our solution and Dune or IX differ completely, even though the underlying idea is similar.

## 4.10 Summary

This chapter takes the first steps to demonstrate a new approach to economically isolate containers in a complex, legacy operating system such as the Linux kernel. Our solution isolates each guest's memory into the memory namespace, and protects the memory namespace using second-level hardware page tables. This chapter redesigned the memory subsystem of the Linux kernel to logically isolate the memory of each guest on a separate set of pages. The lighter weight of containers is maintained by sharing the OS between the container and the host, and the memory isolation of VMs is provided by isolating the memory namespace using a second set of page tables like EPT. Finally, this chapter also showed how the design choices are guided by secure design principles.

This chapter also takes the first steps to demonstrate how to retrofit security on existing kernel design, but observes that retrofitting security requires creatively rethinking the design. The implementation of the network and the file system design is left as the future work, but it is not expected to significantly skew the results in section 4.6. Thus, due to incomplete prototype, this chapter is a significant step in the direction of achieving the best of both the container and the VM world, but it is a long way to go to achieve that goal.

# CHAPTER 5: CONCLUSION

This thesis shows that encapsulating the threats in a complex, legacy operating system like Linux using virtual machines is an imperfect solution, because the strong semantic gap is often obviated by generous threat models. This thesis demonstrates two instances of a better approach to protecting a legacy system like Linux by using economical and practical techniques to retrofit security design principles in the existing kernel design. This thesis economically removed a source of privilege escalation in the Linux userspace by changing only 715 lines of code in the kernel to design and implement a simple, efficient framework for migrating policies from setuid-to-root binaries into the kernel. This thesis obviated the need for these setuid-to-root binaries to run with escalated privileges in nearly all situations, and de-privileged about 12,717 lines of trusted code. Finally, this thesis takes the first steps to economically isolate the memory of containers, and protect the host as well as the guest memory to achieve the goal of VM-like memory isolation properties for containers. This thesis first logically isolates the memory of containers, and then physically isolates that memory using virtualization hardware.

This thesis notes that it is not trivial or mechanical to just apply security design principles arbitrarily to any code. There is an art to this process of finding structural pinch points, that one can then leverage and creatively enforce security design principles, but finding them requires some skill and luck.

# BIBLIOGRAPHY

[1] Cve - search results. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=namespace+privilege+escalation. (Accessed on 08/09/2022).

[2] Draugr. Online at https://code.google.com/p/draugr/.

[3] E.W. Dijkstra archive: On the reliability of programs. https://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD303.html.

[4] FatKit. Online at http://4tphi.net/fatkit/.

[5] Foriana. Online at http://hysteria.sk/~niekt0/foriana/.

[6] GREPEXEC: Grepping Executive Objects from Pool Memory). Online at http://uninformed.org/?v=4&a=2&t=pdf.

[7] https://www.kernel.org/doc/documentation/x86/x86_64/mm.txt. https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt. (Accessed on 08/29/2022).

[8] idetect. Online at http://forensic.seccure.net/.

[9] Intel 64 and IA-32 Architectures Developer's Manual: Vol. 3B.

[10] Intel Software Guard Extensions (Intel SGX) Programming Reference.

[11] Kernel object hooking rootkits (koh rootkits). http://my.opera.com/330205811004483jash520/blog/show.dml/314125.

[12] Kntlist. Online at http://www.dfrws.org/2005/challenge/kntlist.shtml.

[13] Linus torvalds: 'i don't trust security people to do sane things' — zdnet. https://www.zdnet.com/article/linus-torvalds-i-dont-trust-security-people-to-do-sane-things/. (Accessed on 07/31/2022).

[14] Linux security, then and now — esecurity planet. https://www.esecurityplanet.com/trends/linux-security-then-and-now/. (Accessed on 07/31/2022).

[15] lsproc. Online at http://windowsir.blogspot.com/2006/04/lsproc-released.html.

[16] Memparser. Online at http://www.dfrws.org/2005/challenge/memparser.shtml.

[17] PROCENUM. Online at http://forensic.seccure.net/.

[18] Red Hat Crash Utility. Online at http://people.redhat.com/anderson/.

[19] The Linux Cross Reference. Online at http://lxr.linux.no/.

[20] The Volatility framework. Online at https://code.google.com/p/volatility/.

[21] Volatilitux. Online at https://code.google.com/p/volatilitux/.

[22] Windows Memory Forensic Toolkit. Online at http://forensic.seccure.net/.

[23] The openldap project. `http://www.openldap.org/project/`, Aug. 1998.

[24] netfilter. `http://www.netfilter.org/`, Dec. 2001.

[25] The automounter autofs. `http://www.autofs.org/`, Sep. 2002.

[26] Py-notify. `http://home.gna.org/py-notify/`, Dec. 2008.

[27] SOCK_RAW Demystified. `http://www.sock-raw.org/papers/sock_raw`, May 2008.

[28] AppArmor. `http://wiki.apparmor.net/index.php/Main_Page`, Jun. 2011.

[29] Kernel mode setting. `https://wiki.archlinux.org/index.php/Kernel_Mode_Setting`, Jan. 2011.

[30] Chrome owned by exploits in hacker contests, but google's $1m purse still safe — wired. March 2012.

[31] Debian Popularity Contest. `http://popcon.debian.org/by_inst`, Feb. 2013.

[32] Lintian Reports : setuid-binary. `http://lintian.debian.org/tags/setuid-binary.html`, Feb. 2013.

[33] Ubuntu Popularity Contest. `http://popcon.ubuntu.com/by_inst`, Feb. 2013.

[34] Pwn2Own 2016: Windows, OS X, Chrome, Edge, Safari all hacked - gHacks tech news. March 2016. (Accessed on 04/25/2017).

[35] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *CCS*, pages 340–353, 2005.

[36] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th usenix symposium on networked systems design and implementation (nsdi 20)*, pages 419–434, 2020.

[37] Kavita Agarwal, Bhushan Jain, and Donald E Porter. Containing the hype. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, pages 1–9, 2015.

[38] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. BAR Fault Tolerance for Cooperative Services. In *SOSP*, pages 45–58, 2005.

[39] AMD. AMD I/O Virtualization Technology (IOMMU) Specification Revision 1.26. White Paper, AMD: `http://support.amd.com/us/Processor_TechDocs/34434-IOMMU-Rev_1.26_2-11-09.pdf`, Nov 2009.

[40] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. HASP '13, 2013.

[41] Apple. Mac OS X Server V10.6 - Open Directory Administration. White Paper, Apple: `http://manuals.info.apple.com/MANUALS/1000/MA1180/en_US/OpenDirAdmin_v10.6.pdf`, August 2009.

[42] Ahmed M. Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C. Skalsky. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *CCS*, pages 38–49, 2010.

[43] Ahmed M Azab, Kirk Swidowski, Rohan Bhutkar, Jia Ma, Wenbo Shen, Ruowen Wang, and Peng Ning. Skee: A lightweight secure kernel-level execution environment for arm. In *NDSS*, volume 16, pages 21–24, 2016.

[44] Sina Bahram, Xuxian Jiang, Zhi Wang, Mike Grace, Jinku Li, Deepa Srinivasan, Junghwan Rhee, and Dongyan Xu. Dksm: Subverting virtual machine introspection for fun and profit. In *SRDS*, pages 82–91. IEEE Computer Society, 2010.

[45] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. Automatic inference and enforcement of kernel data structure invariants. In *ACSAC*, pages 77–86, 2008.

[46] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP*, pages 164–177, 2003.

[47] Securing Debian Manual: Bastille Linux. `http://www.debian.org/doc/manuals/securing-debian-howto/ch-automatic-harden.en.html#s6.2`, Apr. 2012.

[48] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 335–348, 2012.

[49] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, October 2014. USENIX Association.

[50] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, 2014.

[51] Nikhil Bhatia. Performance Evaluation of Intel EPT Hardware Assist. *VMware ESX white paper)*, October 2012.

[52] M. Bishop. Managing Superuser Privileges under UNIX. Technical report, Research Institute for Advanced Computer Science, NASA Ames Research Center, 1986.

[53] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. USENIX Association, 2008.

[54] Herbert Bos and Bart Samwel. Safe kernel programming in the oke. In *2002 IEEE Open Architectures and Network Programming Proceedings. OPENARCH 2002 (Cat. No. 02EX571)*, pages 141–152. IEEE, 2002.

[55] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure:{SGX} cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[56] Bjorn Bringert. Executable based access control. Technical report, Chalmers University of Technology, 2003.

[57] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y. Wang. Bringing virtualization to the x86 architecture with the original vmware workstation. *ACM TOCS*, 30(4):12:1–12:51, November 2012.

[58] Thomas Wolfgang Burger. Intel Virtualization Technology for Directed I/O (VT-d): Enhancing Intel platforms for efficient virtualization of I/O devices. `http://software.intel.com/en-us/articles/intel-virtualization-technology-for-directed-io-vt-d-enhancing-intel-platforms-for-efficient-virtualization-of-io-devices/`, February 2009.

[59] James Butler and Greg Hoglund. Vice–catch the hookers. *Black Hat USA*, 61:17–35, 2004.

[60] Xiang Cai, Yuwei Gui, and Rob Johnson. Exploiting unix file-system races via algorithmic complexity attacks. In *Oakland*, pages 27–41, 2009.

[61] Tyler Caraza-Harter and Michael M Swift. Blending containers and virtual machines: a study of firecracker and gvisor. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 101–113, 2020.

[62] Martim Carbone, Matthew Conover, Bruce Montague, and Wenke Lee. secure and robust monitoring of virtual machines through guest-assisted introspection. In *RAID*, pages 22–41. Springer-Verlag, 2012.

[63] Martim Carbone, Weidong Cui, Long Lu, Wenke Lee, Marcus Peinado, and Xuxian Jiang. Mapping kernel objects to enable systematic integrity checking. In *CCS*, pages 555–565, 2009.

[64] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 45–58, 2009.

[65] Ramesh Chandra, Taesoo Kim, Meelap Shah, Neha Narula, and Nickolai Zeldovich. Intrusion recovery for database-backed web applications. In *SOSP*, pages 101–114, 2011.

[66] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. In *ASPLOS*, 2013.

[67] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. *ACM SIGARCH Computer Architecture News*, 41(1):253–264, 2013.

[68] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre attacks: Leaking enclave secrets via speculative execution. *arXiv preprint arXiv:1802.09085*, 2018.

[69] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *USENIX Security*, pages 171–190, 2002.

[70] Hong Chen, Ninghui Li, and Ziqing Mao. Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems. In *NDSS*, 2009.

[71] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *HotOS*, pages 133–. IEEE Computer Society, 2001.

[72] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS*, pages 2–13, 2008.

[73] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-grained execution units with private memory. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 56–71. IEEE, 2016.

[74] Andrew Chi, Robert A Cochran, Marie Nesfield, Michael K Reiter, and Cynthia Sturton. A system to verify network behavior of known cryptographic clients. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 177–195, 2017.

[75] Russell Coker. Benchmarking Mail Relays and Forwarders. In *OSDC Conference*, 2006.

[76] Jonathan Corbet. A new adore root kit. *Linux Weekly News*, March 2004. `http://lwn.net/Articles/75990/`.

[77] Jonathan Corbet. File-based capabilities. `http://lwn.net/Articles/211883/`, November 2006.

[78] Jonathan Corbet. Rootless X. `http://lwn.net/Articles/341033/`, July 2009.

[79] Russ Cox, Eric Grosse, Rob Pike, Dave Presotto, and Sean Quinlan. Security in Plan 9. In *USENIX Security*, pages 3–16, 2002.

[80] Andrew Cristina, Lodovico Marziale, Golden G. Richard Iii, and Vassil Roussev. Face: Automated digital evidence discovery and correlation. In *Digital Forensics*, 2005.

[81] John Criswell, Nathan Dautenhahn, and Vikram Adve. Virtual ghost: Protecting applications from hostile operating systems. *ACM SIGARCH Computer Architecture News*, 42(1):81–96, 2014.

[82] Weidong Cui, Marcus Peinado, Zhilei Xu, and Ellick Chan. Tracking rootkit footprints with a practical memory analysis system. In *USENIX Security*, pages 42–42, 2012.

[83] Thomas Dangl, Benjamin Taubmann, and Hans P Reiser. Agent-based file extraction using virtual machine introspection. In *Nordic Conference on Secure IT Systems*, pages 174–191. Springer, 2020.

[84] Thomas Dangl, Benjamin Taubmann, and Hans P Reiser. Rapidvmi: Fast and multi-core aware active virtual machine introspection. In *The 16th International Conference on Availability, Reliability and Security*, pages 1–10, 2021.

[85] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 191–206, 2015.

[86] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *CCS*, pages 51–62, 2008.

[87] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Oakland*, pages 297–312, 2011.

[88] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. Robust signatures for kernel data structures. In *CCS*, pages 566–577. ACM, 2009.

[89] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. In *OSDI*, 2002.

[90] Izzat El Hajj, Alexander Merritt, Gerd Zellweger, Dejan Milojicic, Reto Achermann, Paolo Faraboschi, Wen-mei Hwu, Timothy Roscoe, and Karsten Schwan. Spacejmp: programming with multiple virtual address spaces. *ACM SIGPLAN Notices*, 51(4):353–368, 2016.

[91] Ulfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. Xfi: Software guards for system address spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 75–88, 2006.

[92] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, December 2007.

[93] Fedoraproject Wiki: Features/RemoveSETUID. `https://fedoraproject.org/wiki/Features/RemoveSETUID`, Apr. 2011.

[94] D.F. Ferraiolo and D.R. Kuhn. Role-based access control. In *15th National Computer Security Conference*, pages 554–563, 1992.

[95] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, Mark Williamson, et al. Safe hardware access with the xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, pages 1–1. Boston, USA;, 2004.

[96] Yangchun Fu and Zhiqiang Lin. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Oakland*, pages 586–600. IEEE Computer Society, 2012.

[97] Yangchun Fu and Zhiqiang Lin. Exterior: using a dual-vm based external shell for guest-os introspection, configuration, and recovery. In *VEE*, pages 97–110, 2013.

[98] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, pages 191–206, 2003.

[99] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J Elphinstone, Volkmar Uhlig, Jonathon E Tidswell, Luke Deller, and Lars Reuther. The sawmill multiserver approach. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, pages 109–114, 2000.

[100] Alorie Gilbert. Fixing the sorry state of software - cnet. `https://www.cnet.com/news/fixing-the-sorry-state-of-software/`, October 2002. (Accessed on 08/23/2021).

[101] V.D. Gligor, C.S. Chandersekaran, R.S. Chapman, L.J. Dotterer, M.S. Hetch, Wen-Der Jiang, A. Johri, G.L. Luckenbaugh, and N. Vasudevan. Design and implementation of Secure Xenix. *IEEE Transactions on Software Engineering*, SE-13(2):208 – 221, Feb. 1987.

[102] David B Golub, Guy G Sotomayor, and Freeman L Rawson III. An architecture for device drivers executing as user-level tasks. In *USENIX MACH III Symposium*, pages 153–172, 1993.

[103] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017.

[104] Zhongshu Gu, Zhui Deng, Dongyan Xu, and Xuxian Jiang. Process implanting: A new active introspection framework for virtualization. In *SRDS*, pages 147–156, 2011.

[105] Nicholas Mc Guire. Linux kernel gcov - tool analysis. `http://dslab.lzu.edu.cn:8080/docs/2006summerschool/team1/teama/Documentation/howtos/der_herr_gcov.pdf`, 2006.

[106] Robert T. Hall and Joshua Taylor. A framework for network-wide semantic event correlation, 2013.

[107] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of $\mu$-kernel-based systems. *ACM SIGOPS Operating Systems Review*, 31(5):66–77, 1997.

[108] M.S. Hecht, M.E. Carson, C.S. Chandersekaran, R.S. Chapman, L.J. Dotterrer, V.D. Gilgor, Wen-Der Jiang, A. Johri, G.L. Luckenbaugh, and N. Vasudevan. UNIX without the Superuser. In *USENIX Security*, pages 243 –256, June 1987.

[109] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using cla: a million lines of c code in a second. In *PLDI*, pages 254–263, 2001.

[110] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. Minix 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[111] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. Fault isolation for device drivers. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 33–42. IEEE, 2009.

[112] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *HASP*, 2013.

[113] Owen S. Hofmann, Alan M. Dunn, Sangman Kim, Indrajit Roy, and Emmett Witchel. Ensuring operating system kernel integrity with OSck. In *ASPLOS*, pages 279–290, 2011.

[114] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. Inktag: secure applications on an untrusted operating system. In *ASPLOS*, pages 265–278. ACM, 2013.

[115] Sanghyun Hong, Alina Nicolae, Abhinav Srivastava, and Tudor Dumitraş. Peek-a-boo: Inferring program behaviors in a virtualized infrastructure without introspection. *Computers & Security*, 79:190–207, 2018.

[116] Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber. Sealing os processes to improve dependability and safety. *ACM SIGOPS Operating Systems Review*, 41(3):341–354, 2007.

[117] PLC INTEL. Intel software guard extensions programming reference, 2014.

[118] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *CCS*, pages 128–138, 2007.

[119] Karen Johnson, Jeffrey B. Zurschmeide, John Raithel, Terry Schultz, and Bill Tuthill. IRIX admin: Backup, security, and accounting. Technical report, Silicon Graphics, Inc., 2005.

[120] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Geiger: Monitoring the buffer cache in a virtual machine environment. In *ASPLOS*, ASPLOS XII, pages 14–24, 2006.

[121] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. VMM-based Hidden Process Detection and Identification Using Lycosid. In *VEE*, pages 91–100, 2008.

[122] Michael Kerrisk. CAP_SYS_ADMIN: the new root. `http://lwn.net/Articles/486306/`, March 2012.

[123] Michael Kerrisk. User namespaces progress. `https://lwn.net/Articles/528078/`, Dec. 2012.

[124] D. Kienzle, N. Evans, and M. Elder. NICE: Network Introspection by Collaborating Endpoints. In *Communications and Network Security*, pages 411–412, 2013.

[125] Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich. Recovering from intrusions in distributed systems with DARE. In *APSYS*, pages 10:1–10:7, 2012.

[126] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Intrusion recovery using selective re-execution. In *OSDI*, pages 1–9, 2010.

[127] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *USENIX Security*, pages 191–206, 2002.

[128] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, 2009.

[129] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[130] S. Kramer. LINUS IV — An Experiment in Computer Security. In *Oakland*, pages 24–33, 1984.

[131] Michael Larabel. A NVIDIA Tegra 2 DRM/KMS Driver Tips Up. `http://www.phoronix.com/vr.php?view=MTA4NjA`, Apr. 2012.

[132] Hojoon Lee, Hyungon Moon, Daehee Jang, Kihwan Kim, Jihoon Lee, Yunheung Paek, and Brent ByungHoon Kang. Ki-mon: a hardware-assisted event-triggered monitoring platform for mutable kernel object. In *USENIX Security*, pages 511–526, 2013.

[133] Tamas Lengyel, Thomas Kittel, George Webster, Jacob Torrey, and Claudia Eckert. Pitfalls of virtual machine introspection on modern hardware. In *1st Workshop on Malware Memory Forensics (MMF)*. Citeseer, 2014.

[134] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *OSDI*, volume 4, pages 17–30, 2004.

[135] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 234–251, 2017.

[136] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, Bedford, Massachusetts, 1984.

[137] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. Defeating return-oriented rootkits with ”return-less” kernels. In *EuroSys*, pages 195–208, 2010.

[138] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, pages 9–9, 2004.

[139] David Lie, Chandramohan A. Thekkath, and Mark Horowitz. Implementing an untrusted operating system on trusted hardware. In *SOSP*, pages 178–192, 2003.

[140] Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu, and Xuxian Jiang. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *NDSS*. The Internet Society, 2011.

[141] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-weight contexts: An os abstraction for safety and performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 49–64, 2016.

[142] Lionel Litty, H. Andrés Lagar-Cavilla, and David Lie. Hypervisor support for identifying covertly executing binaries. In *SS*, pages 243–258, 2008.

[143] Limin Liu, Jin Han, Debin Gao, Jiwu Jing, and Daren Zha. Launching return-oriented programming attacks against randomized relocatable executables. In *TRUSTCOM*, pages 37–44, 2011.

[144] Ziyi Liu, JongHyuk Lee, Junyuan Zeng, Yuanfeng Wen, Zhiqiang Lin, and Weidong Shi. Cpu transparent protection of os kernel and hypervisor integrity with programmable dram. In *ISCA*, pages 392–403, 2013.

[145] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *USENIX*, 2001.

[146] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 218–233, 2017.

[147] Valentin JM Manès, Daehee Jang, Chanho Ryu, and Brent Byunghoon Kang. Domain isolated kernel: A lightweight sandbox for untrusted kernel extensions. *computers & security*, 74:130–143, 2018.

[148] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 399–413, 2019.

[149] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX Security*, pages 259–270, 1993.

[150] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Oakland*, pages 143–158, 2010.

[151] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization. In *EuroSys*, pages 315–328, 2008.

[152] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Workshop on Hardware and Architectural Support for Security and Privacy*, pages 10:1–10:1. ACM, 2013.

[153] Fabrice Mérillon, Laurent Réveillere, Charles Consel, Renaud Marlet, and Gilles Muller. Devil: An idl for hardware programming. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*, 2000.

[154] Cade Metz. Google Embraces Docker, the Next Big Thing in Cloud Computing. *WIRED*, June 2014. `http://www.wired.com/2014/06/eric-brewer-google-docker/`.

[155] Microsoft. TCP/IP Raw Sockets. `http://msdn.microsoft.com/en-us/library/windows/desktop/ms740548%28v=vs.85%29.aspx`, 2012.

[156] MITRE. Common Vulnerabilities and Exploits Database. `http://cve.mitre.org/`, Feb. 2013.

[157] Hyungon Moon, Hojoon Lee, Jihoon Lee, Kihwan Kim, Yunheung Paek, and Brent Byunghoon Kang. Vigilare: toward snoop-based kernel integrity monitor. In *CCS*, pages 28–37, 2012.

[158] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. {RedLeaf}: Isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 21–39, 2020.

[159] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In *ACSAC*, pages 49–58, 2010.

[160] Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *Oakland*, pages 233–247, 2008.

[161] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, October 2014. USENIX Association.

[162] Nick L. Petroni, Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *USENIX Security*, pages 13–13, 2004.

[163] Nick L. Petroni, Jr. and Michael Hicks. Automated detection of persistent kernel control-flow attacks. In *CCS*, pages 103–115. ACM, 2007.

[164] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *CACM*, 17(7):412–421, July 1974.

[165] Summary about POSIX 1.e. `http://wt.tuxomania.net/publications/posix.1e/`, Feb. 1999.

[166] Sergej Proskurin, Julian Kirsch, and Apostolis Zarras. Follow the whiterabbit: Towards consolidation of on-the-fly virtualization and virtual machine introspection. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 263–277. Springer, 2018.

[167] Niels Provos. Improving host security with system call policies. In *USENIX Security*, pages 257–272, 2002.

[168] Alessandro Randazzo and Ilenia Tinnirello. Kata containers: An emerging architecture for enabling mec services in fast and secure way. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pages 209–214. IEEE, 2019.

[169] Junghwan Rhee, Ryan Riley, Dongyan Xu, and Xuxian Jiang. Kernel malware analysis with untampered and temporal views of dynamic kernel memory. In *RAID*, pages 178–197, 2010.

[170] Wolfgang Richter, Glenn Ammons, Jan Harkes, Adam Goode, Nilton Bila, Eyal De Lara, Vasanth Bala, and Mahadev Satyanarayanan. Privacy-sensitive VM Retrospection. In *HotCloud*, pages 10–10, 2011.

[171] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In *RAID*, pages 1–20, 2008.

[172] Dennis M Ritchie. ‘on the security of unix. *UNIX Supplementary Documents*, 1979.

[173] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming device drivers. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 275–288, 2009.

[174] Alireza Saberi, Yangchun Fu, and Zhiqiang Lin. HYBRID-BRIDGE: Efficiently Bridging the Semantic Gap in Virtual Machine Introspection via Decoupled Execution and Training Memoization. In *NDSS*, 2014.

[175] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer System. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[176] Andreas Schuster. Pool allocations as an information source in Windows memory forensics. In *IMF*, pages 104–115, 2006.

[177] Andreas Schuster. The impact of Microsoft Windows pool allocation strategies on memory forensics. *Digital Investigation*, 5:S58–S64, 2008.

[178] Michael Schwarz, Samuel Weiser, and Daniel Gruss. Practical enclave malware with intel sgx. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 177–196. Springer, 2019.

[179] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using sgx to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24. Springer, 2017.

[180] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: abusing intel sgx to conceal cache attacks. *Cybersecurity*, 3(1):1–20, 2020.

[181] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *SOSP*, pages 335–350, 2007.

[182] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *SOSP*, pages 170–185, 1999.

[183] Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-VM monitoring using hardware virtualization. In *CCS*, pages 477–487, 2009.

[184] Ian Shields. Monitor linux file system events with inotify. `http://www.ibm.com/developerworks/linux/library/l-inotify/index.html`, Sep. 2010.

[185] Yonghee Shin and Laurie Williams. An Empirical Model to Predict Security Vulnerabilities Using Code Complexity Metrics. In *ESEM*, pages 315–317, 2008.

[186] Takahiro Shinagawa and Kenji Kono. Implementing A Secure Setuid Program. In *Proceedings of the Conference on Parallel and Distributed Computing and Networks*, pages 301–309, 2004.

[187] Jim Slater. The isrg wants to make the linux kernel memory-safe with rust — ars technica. `https://arstechnica.com/gadgets/2021/06/lets-encrypt-parent-org-sponsors-rust-for-linux-kernel-development/`, June 2021. (Accessed on 08/23/2021).

[188] Deepa Srinivasan, Zhi Wang, Xuxian Jiang, and Dongyan Xu. Process out-grafting: An efficient "out-of-VM" approach for fine-grained process execution monitoring. In *CCS*, pages 363–374. ACM, 2011.

[189] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference, General Track*, pages 1–14, 2001.

[190] Michael M Swift, Brian N Bershad, and Henry M Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems (TOCS)*, 23(1):77–110, 2005.

[191] Michael M. Swift, Peter Brundrett, Cliff Van Dyke, Praerit Garg, Anne Hopkins, Shannon Chan, Mario Goertzel, and Gregory Jensenworth. Improving the granularity of access control in Windows NT. In *Proceedings of the ACM Symposium on Access Control Models and Technologies*, pages 87–96, 2001.

[192] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[193] Vasily Tarasov, Deepak Jain, Dean Hildebrand, Renu Tewari, Geoff Kuenning, and Erez Zadok. Improving I/O performance using virtual disk introspection. In *HotStorage*, pages 11–11. USENIX, 2013.

[194] Akihiro Tominaga, Osamu Nakamura, Fumio Teraoka, and Jun Murai. Problems and Solutions of DHCP - Experiences with DHCP implementation and Operation. `http://www.isoc.org/inet95/proceedings/PAPER/127/html/paper.html`, 1995.

[195] Dan Tsafrir, Dilma Da Silva, and David Wagner. The murky issue of changing process identity: revising "setuid demystified". *USENIX ;login*, 33(3):55–66, June 2008.

[196] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. Cooperation and Security Isolation of Library OSes for Multi-Process Applications. In *EuroSys*, pages 9:1–9:14, 2014.

[197] Ubuntu Wiki: Filesystem Capabilities. `https://wiki.ubuntu.com/Security/Features#Filesystem_Capabilities`, March 2014.

[198] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page {Table-Based} attacks on enclaved execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1041–1056, 2017.

[199] Kenneth M. Walker, Daniel F. Sterne, M. Lee Badger, Michael J. Petkac, David L. Sherman, and Karen A. Oostendorp. Confining root programs with domain and type enforcement (DTE). In *USENIX Security*, pages 3–3, 1996.

[200] Jiang Wang, Angelos Stavrou, and Anup Ghosh. Hypercheck: A hardware-assisted integrity monitor. In *RAID*, pages 158–177, 2010.

[201] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2421–2434, 2017.

[202] Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. Countering kernel rootkits with lightweight hook protection. In *CCS*, pages 545–554. ACM, 2009.

[203] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *CCS*, pages 157–168, 2012.

[204] Dan Williams and Ricardo Koller. Unikernel monitors: extending minimalism outside of the box. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[205] R.M. Wong. A comparison of secure UNIX operating systems. In *ACSAC*, pages 322–333, 1990.

[206] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. K. Hartman. Linux security modules: General security support for the Linux kernel. In *USENIX Security Symposium*, 2002.

[207] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *CACM*, 17(6):337–345, June 1974.

[208] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656. IEEE, 2015.

[209] Junfeng Yang, Ang Cui, Sal Stolfo, and Simha Sethumadhavan. Concurrency attacks. In *HotPar*, pages 15–15, 2012.

[210] Chuan Yue. Teaching computer science with cybersecurity education built-in. In *2016 USENIX Workshop on Advances in Security Education (ASE 16)*, 2016.

[211] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *CCS*, pages 305–316, 2012.

[212] Youhui Zhang, Yu Gu, Hongyi Wang, and Dongsheng Wang. Virtual-machine-based intrusion detection on file-aware block level storage. In *SBAC-PAD*, pages 185–192, 2006.

[213] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. Safedrive: Safe and recoverable extensions using language-based techniques. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 45–60, 2006.