Kernel Self-Protection through Quantified Attack Surface Reduction

Von der
Carl-Friedrich-Gauß-Fakultät
der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades eines
Doktoringenieurs (Dr.-Ing.)

genehmigte Dissertation

von
Anil Kurmus
geboren am 9. Januar 1986
in Ankara (Turkei)

Eingereicht am: 28. Mai 2014
Disputation am: 16. Juli 2014
1. Referent: Prof. Dr. Rüdiger Kapitza
2. Referent: Prof. Dr. Herbert Bos

2014

This `NULL` *page intentionally mapped.*

Abstract

An operating system kernel orchestrates the use of the hardware by programs. Much of the overall security of a system depends on the kernel, yet commodity OS kernels count millions of lines of code. Inevitably, some of this code contains vulnerabilities that can be exploited by attackers.

However, attackers need more than finding a single vulnerability in the source code. They need to realize the conditions under which the vulnerability is triggered. The amount of the kernel code that is reachable to an attacker, regardless of it being exploitable is defined here as the *attack surface*.

This thesis provides an in-depth analysis of what the kernel attack surface is and how it can be measured, how it can be reduced, and an evaluation of the advantages and disadvantages of doing so.

The quantification is based on a reachability analysis over the kernel call graph, taking into account assumptions on the attacker's privileges. For example, precise measurements show that, out of the 10 million lines of code in the kernel sources, only about 2.5 million lines are reachable to a local unprivileged attacker on a popular Linux distribution.

The attack surface reduction techniques are twofold. The first makes use of the Linux kernel's configurability to generate a workload-tailored Linux kernel, at compile time. The second achieves finer granularity by instrumenting individual kernel functions at run time, without requiring any kernel recompilation.

Using the quantification techniques of this thesis, I show the effectiveness of both approaches. They can both reduce drastically (more than four-fold) the kernel attack surface. I conclude that attack surface reduction, or the principle of *economy of mechanism*, has been rather neglected on commodity OS kernels, and that the security benefits that would be achieved outweigh their drawbacks in many practical use cases.

This `NULL` *page intentionally mapped.*

Table of Contents

List of Figures

List of Tables

1 Introduction

> “It was a pleasure to burn.”
>
> Ray Bradbury, *Farenheit 451*

Security vulnerabilities allow attackers to cross trust boundaries. Often, it is an operating system kernel which is relied upon to establish and maintain those boundaries. For instance, in a platform-as-a-service cloud designed with security in mind, each customer can be served by a process within a distinct security domain. Similarly, the security of an internal corporate network is dependent on the security of each of the endpoints in the network, which often includes embedded systems such as printers and phones, and personal devices such as smartphones and tablets. In each of those cases, the kernel can enforce security policies (such as preventing low-level network access or the installation of stealth rootkits on the device) for applications running on each device.

In other words, the OS kernel, due to it’s traditional *reference monitor* role [And72], is part of the Trusted Computing Base (TCB) of the overall system. Because limiting the amount of trusted components, i.e., reducing TCB size, is generally accepted to improve security, this means smaller kernels result in better overall system security as well. Indeed, less code means a lower likelihood of defects existing (about 5 defects per 1,000 source lines of code (SLOC) on average [BP84; Lu+13; OWB04]).

Traditionally, TCB size is measured in terms of SLOC of the TCB components. For instance, part of the operating system research community focuses on building small kernels (e.g., micro-kernels [Ber+94; Her+06b; Lie95] or exo-kernels [EKO95]), in an effort to improve dependability in general. However, the choice of an operating system in today’s cloud services or end-user devices is rarely dictated by security considerations — to the extent that this choice still exists. Rather, ease-of-development and compatibility

with existing middleware and applications is of primary importance.

From the above, we can derive one of the primary assumptions motivating much of the systems security research work, including this dissertation:

A1: *Compatibility* with existing systems should be ensured for novel security tools to be used in current production systems.

A powerful and widespread current approach to improve application security is via *sandboxing*, i.e., limiting the interactions of a process with the rest of the system. We can categorize some of the existing application-level "sandboxes" as follows: resource access control [SVS01; Tom; Wri+02], software-based fault isolation [Wah+93; Yee+09; Zha+11], language-based sandboxing [Gos00; IFF96; WG92], OS-level virtualization [KW00; Lxc; PT04].

All these mechanisms extend the security services provided to the applications by the kernel, such as isolation of processes and access control to resources. For example, SELinux [SVS01] enables fine-granularity access control compared to traditional Discretionary Access Control (DAC), as well as central security policy control, while Linux Containers (LXC) [Lxc] enables PID- and network- namespaces.

Mainly, these techniques help in building secure applications such as those that employ sandboxes when parsing untrusted user input. However, it is worth noting that all those approaches assume a trusted the kernel. Once the kernel is compromised, the attacker has full control over each application running on the system.

To what extent do these mechanisms protect the kernel itself? Or in other words, do these mechanism provide any *kernel protection*, in particular by limiting the amount of kernel code reachable to attackers (i.e., by limiting the *kernel attack surface*)? These previously mentioned "sandboxes" were not designed with this goal in mind, and experience indicates that, at least under some conditions, such mechanisms are not effective in prevent kernel exploits. For instance, much of the past Linux kernel exploits have been known to work regardless of resource access control mechanisms such as SELinux being used [Spe].

This leads to the second assumption underlying this work:

A2: Existing security mechanisms are not sufficient for kernel attack surface reduction as they were not designed with such a goal in mind.

Although some kernel protection mechanisms, such as seccomp [Goo09], do reduce attack surface, the reduction has neither been quantified, nor has the mechanism been designed in a quantifiable way.

Therefore, I design mechanisms that focus on quantifiable attack surface reduction as a primary goal. More importantly, it becomes clear that we now need to precisely define and quantify what the kernel attack surface is in order to develop new techniques that reduce kernel attack surface and compare them objectively.

Hence, the *research questions* that this thesis aims to answer are:

Q1: *Is it possible to precisely define the kernel attack surface? Can it be measured?*

Past kernel exploits show that attackers will leverage obscure functionality present in the kernel to achieve their goals, hence it is intuitively understood that such additional functionality significantly increases the chances of an attacker finding such vulnerabilities. Can we define and quantify the kernel attack surface to confirm or refute this established wisdom?

Q2: *Can we develop kernel protection mechanisms whose attack surface reduction is quantifiable? To what extent can these mechanisms be applied to commodity OSes in practice?*

No quantification of the attack surface reduction achieved by existing kernel protection mechanisms exists. Can we design and implement mechanisms that do? If so, what are their limitations? How much attack surface reduction can be achieved in practice?

1.1 Main Contributions

This dissertation contains three main contributions: one related to the formalization and quantification of kernel attack surface, and two more related to the design, implementation, and evaluation of novel kernel attack surface reduction techniques.

First, most operating systems security research focuses on measuring TCB size in SLOC to quantify of the amount of code that, if defects were to exist in, could result in security guarantees not being met. Because this approach is too coarse and imprecise for quantifying kernel self-protection methods, I develop new metrics to fill this gap.

C1: Kernel Attack Surface Quantification. For a given set of assumptions on the operation of the kernel (e.g., use of loadable kernel modules (LKMs)), and assumptions on the attacker's interaction with the kernel (e.g., the attacker control an unprivileged process), which is referred to as the *security model*, I formally define what the attack surface represents. In turn, I use this to derive *attack surface metrics* that are used to measure and compare attack surfaces. I perform attack surface measurements not only on current distribution kernels, but also for *tailored* and *trimmed* kernels.

As we can measure kernel attack surface, it becomes easier to understand how to design mechanisms that focus on reducing kernel attack surface. I design and implement new techniques that quantifiably reduce kernel attack surface, while keeping in mind *compatibility* and *performance*:

C2: Compile-time Kernel Tailoring. (joint work)[1] The Linux kernel is highly configurable, and most kernel distributions attempt to maximize user satisfaction (and support) by shipping with as many kernel features enabled as possible. However, this is in turn offers attackers a large code base to exploit. A simple approach to produce a smaller kernel is to manually configure a *tailored* Linux kernel, by only enabling features necessary for the use case the kernel will be used for. However, the more than 11,000 configuration options available in recent Linux versions make this a time-consuming and non-trivial task. We design and implement an automated approach to produce a kernel configuration that is adapted to a particular workload and hardware. Results show that, for real-world server use cases, the attack surface reduction obtained by tailoring the kernel ranges from about 50% to 85%. Therefore, kernel tailoring is an attractive approach to improve the security of the Linux kernel in practice.

C3: Run-time Kernel Trimming. In this contribution, I show it's possible to reduce the kernel's attack surface by tracing individual kernel functions used by an application, and, after using the kernel call graph to infer a greater set of permissible functions,

[1] This contribution is joint work with another PhD student, Reinhard Tartler, who contributed his domain-specific knowledge of the Linux kernel build system, and of a tool he developed in his thesis to extract the configuration dependencies therein [Tar+11]. I contributed the idea of leveraging configuration dependencies to generate workload-tailored kernels, and the usage of kernel tracing as previously performed in [KSK11].

it's possible to detect uses of unnecessary kernel functions by a process. Unlike previous work, this results in quantifiable, non-bypassable, per-application kernel attack surface reduction at run-time. I implement this approach as a kernel module, KTRIM, and evaluate results under real-world workloads for four typical server applications. Results show that the performance overhead and false positives remain low, while the attack surface reduction can be as high as 80%.

1.2 Related Publications

Parts of this thesis have been published in peer-reviewed conferences, namely:

1. Anil Kurmus, Sergej Dechand, and Rüdiger Kapitza. "Quantifiable Run-time Kernel Attack Surface Reduction". In: *Proceedings of the 10th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'14)*. London/Egham, UK, 2014 (to appear) [KDK14]

2. Anil Kurmus et al. "Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring". In: *Proceedings of the 20th Network and Distributed System Security Symposium*. San Diego, CA United States, 2013 [Kur+13]

3. Anil Kurmus, Alessandro Sorniotti, and Rüdiger Kapitza. "Attack Surface Reduction For Commodity OS Kernels". In: *Proceedings of the Fourth European Workshop on System Security*. Salzburg,Austria, 2011 [KSK11]

4. Anil Kurmus et al. "A Comparison of Secure Multi-Tenancy Architectures for Filesystem Storage Clouds". In: *Middleware*. Lisbon, Portugal, 2011, pages 471–490 [Kur+11]

1.3 Organization

Chapter 2 presents an overview of existing work related to kernel attack surface reduction: examples of past kernel vulnerabilities, state-of-the-art of kernel self-defense mechanisms, and finally a survey of existing security metrics that are related to attack surface quantification. Chapter 3 describes the methods I developed to quantify kernel attack surface reduction: this includes formal definitions and assumptions, examples of

attack surface metrics. Chapter 4 focuses on compile-time kernel attack surface reduction (KTAILOR), presenting both the implementation and evaluation results. Similarly, Chapter 5 focuses on run-time kernel attack surface reduction (KTRIM). Chapter 6 compares KTAILOR and KTRIM, and presents a study showcasing a cloud-service case study where KTAILOR and KTRIM can be applied. Finally, I conclude in Chapter 7.

2

State-of-the-art Kernel Protection, Exploitation and Security Metrics

> “The best books, he perceived, are those that tell you what you know already.”
>
> George Orwell, *1984*

This chapter takes a bottom-up approach to present the state-of-the-art in kernel protection. I start with a few examples of past kernel vulnerabilities and an overview of the typical three phases of writing an exploit. Then, I describe kernel protection mechanisms, especially in contrast to kernel security services: the distinction is central in understanding the goals of kernel attack surface reduction. Next, I categorize existing kernel protection mechanisms. As a by-product, I also propose a (non-exhaustive) taxonomy of kernel protection mechanism where kernel attack surface reduction can be situated. Finally, I briefly review existing ways of quantifying the effectiveness of kernel protection mechanisms.

The references given here will, when applicable, focus on Linux. This is mainly because the Linux kernel is taken as the primary example in this thesis. However, many examples of vulnerabilities and security mechanisms have their counterparts in other operating system kernels, and the applicability of the taxonomy presented here, and of this work in general, is broad.

2.1 Kernel Vulnerabilities

This thesis pursues pursues the line of research which consists in creating novel protections based on the observation of how the exploitation of operating systems occurs in practice — an area of systems security research which has proven to be effective in raising the bar for attackers over the past fifteen years [Vee+12].

To motivate the need for kernel attack surface reduction and other kernel protection mechanisms, I show a few examples of kernel vulnerabilities and give some high-level insight into the process of exploiting a vulnerability. This matters mainly for two reasons. Firstly, these vulnerabilities and exploits provide a proof that, as it stands, commodity operating systems such as Linux cannot be considered *secure* (for a given version). Secondly, their understanding provides an intuition of which protection mechanisms can be effective, and to what extent. In particular, it is noteworthy that many vulnerabilities were found in features that rarely need to be used on production systems.

2.1.1 Examples

Among the many Common Vulnerabilities and Exposures (CVE) entries available for the Linux kernel, the vulnerabilities I describe below all had publicly available exploits. This is the main criterion that I use in their selection. In addition to being recent, the vulnerabilities also illustrate distinct types of exploits: the first three provide attackers with privilege escalation capabilities, while the last one allows leakage of kernel stack contents. The privilege escalation exploits are very different in their nature: out-of-bounds array access, insufficient access control checks, and unchecked user-pointer dereference. This helps in having an intuitive understanding of the common saying that "there is no silver bullet in security", i.e., that a single security mechanism is unlikely to prevent all kernel vulnerabilities. Finally, they provide us with the intuition that rarely-used functionality may be responsible for a significant amount of vulnerabilities.

These examples will be referred to later on, in Chapter 4 and Chapter 5, as supplementary anecdotal evidence of the effectiveness of proposed kernel protection mechanisms.

`perf_swevent_init` **(CVE-2013-2094).** This vulnerability concerns the Linux kernel's recently introduced low-level performance monitoring framework. Such frameworks, while useful in development environments, are not necessary in production systems that may be exposed to attackers. It was discovered using the TRINITY fuzzer [Dav],

and, shortly after its discovery, a kernel exploit presumably dated from 2010 was publicly released, suggesting that the vulnerability had been exploited in the wild for the past few years. The vulnerability is an out-of-bounds access (decrement by one) into an array, with a partially-attacker-controlled index. Indeed, the index variable, `event_id` is declared as a 64 bit integer in the kernel structure, but the `perf_swevent_init` function assumes it is of type `int` when checking for its validity: as a consequence, the attacker controls the upper 32 bits of the index freely. In the publicly released exploit, the `sw_perf_event_destroy` kernel function is then leveraged to provoke the arbitrary write, because it makes use of `event_id` as a 64-bit index into the array. This results in arbitrary kernel-mode code execution.

`check_mem_permission` **(CVE-2012-0056).** This vulnerability resides in the rarely-used, alternative debuging interface that is provided through `/proc/self/mem`. A detailed description of this cleverly-discovered vulnerability is provided by its author online[1]. It essentially consists in tricking a set-user-id process into writing to its own memory (through `/proc/self/mem`) attacker-controlled data, resulting in obtaining root access. The vulnerability is in the kernel function responsible for handling permission checks on `/proc/self/mem` writes: `__check_mem_permission`.

`sk_run_filter` **(CVE-2010-4158).** This vulnerability is in the implementation of the Berkeley Packet Filter (BPF) [MJ93] system used to filter network packets directly in the kernel. It is a "classic" stack-based information leak vulnerability: a carefully crafted input allows an attacker to read uninitialized stack memory. Such vulnerabilities can potentially breach confidentiality of important kernel data, or be used in combination with other exploits, especially when kernel hardening features are in use (such as kernel base address randomization).

`rds_page_copy_user` **(CVE-2010-3904).** This vulnerability is in reliable datagram sockets (RDS), a seldom used network protocol. The vulnerability is straightforward: the developer has essentially made use of the `__copy_to_user` function instead of the `copy_to_user` function which checks that the destination address is not within kernel address space. This results in arbitrary writes (and reads) into kernel memory, and therefore kernel-mode code execution. This vulnerability is in an LKM which is not in use on the target system, yet, because of the Linux kernel's on-demand LKM loading feature which will load some kernel modules when they are made use of by user-space

[1] `http://blog.zx2c4.com/749`

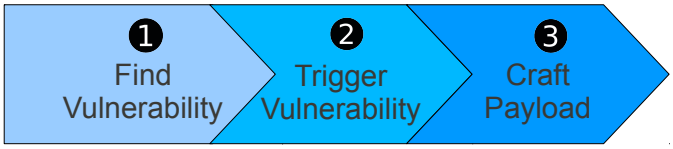


Figure 2.1. High-level exploit development steps

applications, the vulnerability was exploitable on many Linux systems.

2.1.2 Exploitation Process

Exploiting a vulnerability for arbitrary code execution is most often[2] a three step process (see Figure 2.1): ❶ Discovery of the vulnerability. This can be achieved through manual analysis of the source, static analysis, or run-time techniques such as fuzz-testing and selective symbolic execution. ❷ Triggering the vulnerability. Especially in the case where the vulnerability has been found through manual or static analysis, the attacker needs to create a small test program to trigger the vulnerability[3]. In particular, some vulnerabilities can be impossible to reach for the attacker, especially in large software such as the Linux kernel — this could be because it can only be triggered with elevated privileges, or one of the many other reasons we will describe in Chapter 3. ❸ Developing a payload. In the case of Linux kernel vulnerabilities, local attacks will typically aim to elevate privileges by executing arbitrary code under kernel privileges. This is typically quite straightforward in existing kernels once the attacker controls the instruction pointer, because the attacker can simply return into code of his choosing in user-space. Grsecurity [St] kernels are known to have protections against such payloads, and it can be significantly more difficult for attackers to craft payloads in such cases.

As an example, the RDS vulnerability (explained in Section 2.1.1) has been discovered by Rosenberg thanks to a simple text-search over the kernel source files for the callers of the `__copy_from_user` function. Triggering the vulnerability was not a challenge: most Linux distributions compile the RDS module, and, although the kernel module is not loaded at boot-time, it is configured to be automatically loaded at the first use of the socket (by any unprivileged process). Hence the vulnerable code path can be triggered by simple making an RDS socketcall and receiving a network packet. Finally, Rosenberg

[2] See [Bra+12] for a more detailed description

[3] This is usually referred to as a proof-of-concept (PoC) in the exploitation community.

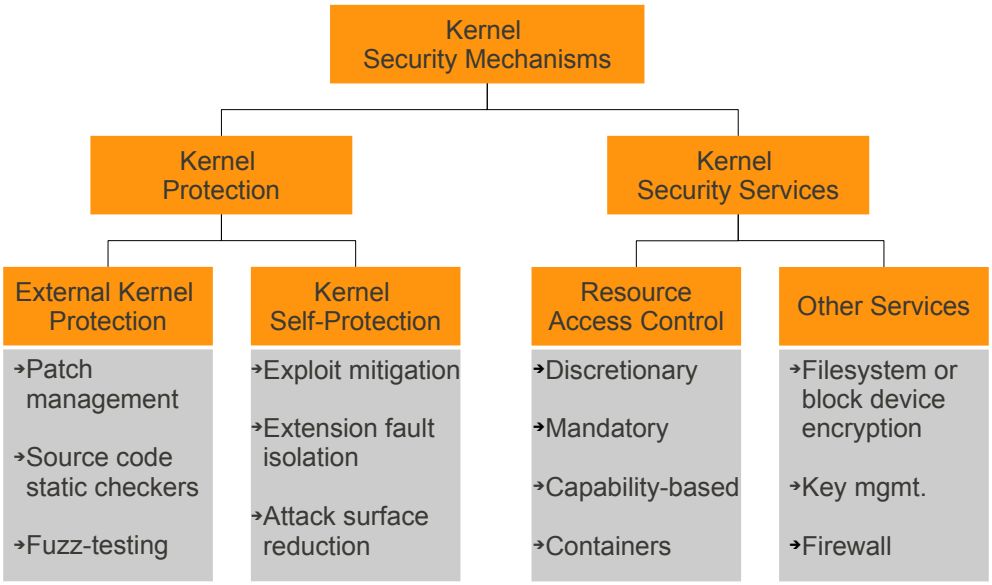


Figure 2.2. A proposed taxonomy of kernel security mechanisms.

created an exploit by using the vulnerability as an arbitrary content, length and location write primitive into kernel memory (by receiving an RDS packet to an attacker-chosen buffer location in the kernel, containing attacker-provided data that was sent by the exploit) — the actual payload was to overwrite an RDS `ioctl` function pointer with the address of a (userland) function in the exploit that gives root privileges to the currently running process (the RDS exploit itself).

As we will see, each of the approaches presented in Section 2.2.2 (or alternatively, under "kernel protection" in Figure 2.2) can be seen as making one of these steps more difficult for the attacker. In a nutshell: patch management, static analysis, fuzz-testing and symbolic execution make vulnerability discovery more difficult; attack surface reduction makes triggering the vulnerability more difficult; extension isolation and exploit mitigation make payload generation more difficult.

2.2 Kernel Protection

We now survey mechanisms that can prevent kernel vulnerabilities from being exploited, or that can remove vulnerabilities altogether. Before this, however, I compare kernel protection and kernel security services. This distinction is important to understand the benefits of kernel attack surface reduction, and how it is different from resource access control mechanisms such as SELinux [SVS01].

2.2.1 Security Services vs. Protection

Many tools aim to improve kernel security. However, *improving kernel security* is a rather vague formulation that would benefit from being defined. Distinguishing between kernel (security) *protection* and kernel security services helps preventing misunderstandings over the main objectives and effectiveness of kernel security mechanisms.

Kernel protection aims to either protect the vulnerabilities that exist in the kernel from being exploited, or to remove those vulnerabilities in the first place. In contrast, kernel security services are mechanisms that are provided by the kernel to user-space to enable secure application design and inter-process isolation (i.e., the creation of trust boundaries between processes).

In particular, I focus mainly on the state-of-the-art of kernel protection mechanisms, simply because kernel attack surface reduction belongs to this category. For this reason, Section 2.2.1.1 briefly reviews existing security mechanisms especially from a historical perspective, while we reserve a larger part in Section 2.2.2 to references on kernel protection. Section 2.2.1.2 provides historical references for the need for kernel protection.

2.2.1.1 Kernel Security Services: History and Examples

The primary goal of this category of kernel security mechanisms is to provide security services to user-space applications. The kernel, being usually a trusted entity which mediates access to security-critical resources, is indeed in a prime position to provide resource access control services to applications. This has been one of the first responsibilities of operating system kernels: one of the innovation of Multics [CV65; Sal74] was to provide ACLs for files, as well as application isolation through the use of virtual memory.

Note that, although access control can be seen as a kernel service preventing untrusted application from accessing sensitive resources, it can also be seen as a service restricting information flows and (partially) solving the confinement[4] problem as formulated by Lampson [Lam73]. Over time, existing access control mechanisms were deemed insufficient, e.g., because they were not granular enough, or because they would not apply to some types of resources. In particular, this lead to the development of Mandatory Access Control (MAC) frameworks [Los+98; SVS01; Wat+03] and capability-based access control systems [Har85; Wat+12]. Another trend is the move towards OS-level virtualization solutions such as FreeBSD jails [KW00] or LXC [Lxc]. For instance, LXC provides process-id namespace, network interface namespaces, as well as the possibility of restricting CPU and memory usage per container. Access control is not the only security service that is provided by an OS kernel. Cryptographic services, as instantiated in the encrypting stacked filesystem ecryptfs or kernel-provided key management services[5], are one such example. In addition to data-at-rest protection, cryptography is also used for load-time code-signing, as implemented in technologies such as IMA [Sai+04]. Another example is network firewalls. Note that quotas (e.g., maximum number of processes, maximum storage space usage) would be categorized as resource access control mechanisms.

2.2.1.2 Kernel (Self-) Protection: History

Kernel protection encompasses mechanisms that prevent attackers from taking advantages of design or implementation defects in the kernel itself. Attackers are interested in such attacks: they provide a way of subverting the kernel-provided security services described above, through direct compromise of the kernel. Such attacks are not new: the first publication[6] considering and documenting such attacks is the 1972 report of the Computer Security Planning Study Panel [And72]. In the first Appendix, Anderson identifies and categorizes multiple sources of security threats, among which scavenging and incomplete parameter checking. Interestingly, those attacks remain realistic on modern kernels.

[4] Confinement is a mechanism preventing an untrusted application from communicating (or exfiltrating) information it might have gathered from sensitive resources.

[5] On Linux, see `Documentation/filesystems/ecryptfs.txt` and `Documentation/keys.txt` in the kernel source.

[6] To the best of my knowledge.

For instance, although the specific scavenging attack[7] described does not exist in modern kernels, similar attacks do: such as uninitialized members of a C structure being passed to user by the kernel. Those vulnerabilities are referred to more broadly as *information leakage* vulnerabilities nowadays. Incomplete user-provided parameter checking is also a vulnerability that remains relevant to this day: the Linux kernel RDS exploit [Ros] from 2010 fits Anderson's definition: the kernel does not check the user-provided buffer's location, hence, an attacker can trick the kernel into reading or writing to kernel memory attacker-controlled data[8].

Anderson's report also recommended both the development of secure operating systems from scratch, and retrofitting security into existing systems on the short term. These approaches can be considered as kernel protection mechanism. As of 2013, this trend towards new kernel protection mechanisms continues, e.g., with approaches aiming to develop new secure OS kernels from scratch, such as the formally verified seL4 [Kle+09] kernel, as well as approaches that improve the security of commodity OS kernels with various protection mechanisms that do not necessitate a complete overhaul of the kernel (e.g., PaX kernel hardening [St]). In the remainder of this work, I use the term kernel *self*-protection to refer to those approaches that involve modifying the kernel (either at compile time or at run time) in order to protect the kernel from attacks.

2.2.1.3 Relationship between the two categories

The raison-d'être of operating systems is to provide services to user space. Security-related services allow system administrators and user-space programmers to design systems with enhanced security, e.g., by separation of components according to a least-privilege design [Sal74]. Such security services can only be assured as long as kernel self-protection is effective in preventing attacks directly targeting the kernel: compromising the kernel allows attacker to circumvent all controls. This dependency between kernel self-protection security services is mutual. Indeed, it is only necessary to prevent attacks directed towards the kernel if an attacker's actions are restricted by the kernel in the first

[7] Anderson gives the example of uninitialized memory being assigned to a program.

[8] "The code performing this function does not check the source and destination addresses properly, permitting portions of the monitor to be overlaid by the user" [And72]. This same paragraph is sometimes used as a reference to buffer overflows: e.g., see the Wikipedia page [Wik] or [Hal+]. In my interpretation of the paragraph, the explanation does not correspond to a buffer overflow but rather to missing user address access checks.

place.

As an example, non-multi-user and non-connected commodity operating systems such as Disk Operating System (DOS) provided no security services to user space[9], and, unsurprisingly, no kernel self-protection mechanism existed. Nowadays, many processes are sandboxed: browser renderers, OpenSSH server connection handling, as well as individual application on mobile platforms. To circumvent those sandboxes, attackers are increasingly targeting the kernel [Ben+12; Eva12], which motivates the development of improved kernel self-protection techniques.

2.2.2 State-of-the-art in Kernel Protection

Over the years, both industry and academia have strived to improve the state-of-the-art in kernel protection. I categorize and list some of those efforts below, with a particular emphasis on techniques that are applicable to the Linux kernel. The *proactiveness*, i.e., the ability of preventing unknown vulnerabilities or classes of vulnerabilities from being exploited, is also mentioned for each technique. The more proactive a protection mechanism is, the less likely it is that newly discovered vulnerability impacts a system. Thus it is a valuable parameter when informally assessing the usefulness of a protection mechanism.

Figure 2.2 presents a proposed taxonomy that categorizes the approaches described below. I first start with external kernel protection mechanisms — as opposed to kernel self-protection, where kernel protection is ensured "within" the kernel (i.e., by making changes to the kernel, its configuration, or its build system).

2.2.2.1 Patch Management

Large software projects tend to suffer from a large number of defects, and in particular, security-related defects which need to be patched with higher urgency (vulnerabilities). Commodity OS kernels are no exception to this rule. Security vulnerabilities are commonly tracked in the form of CVE entries that contain information on each vulnerability (e.g., affected product versions, description, severity scores)[10].

[9] In MS-DOS, every process had access to all files, including the kernel image.

[10] Note that CVE entries merely form a subset of all known vulnerabilities that exist in a particular program, because only vulnerabilities that were reported to the CVE database maintainers can be considered for inclusion. For instance, the vendor can fix vulnerabilities in a program without applying for a CVE,

The Linux kernel had on average 118 CVE entries each year for the past 5 years (2009 to 2013 inclusive)[11]. Linux kernel distributors, such as Red Hat, examine each new CVE (and sometimes report CVEs) and evaluate whether they are applicable to the distribution kernels they are maintaining, issuing advisories accordingly. In case a vulnerability exists in one of the maintained distribution kernels, they will patch and compile a new kernel to create an updated kernel package (the speed at which this is done, and whether multiple patches are bundled together depends mostly on the perceived severity of the vulnerability). In turn, sysadmins using the distribution kernels are tasked with upgrading the kernel (usually after testing, in large enterprises). This process is a part of what is generally referred to as *patch management*.

The goal of patch management is essentially to reduce as much as possible the window of vulnerability[12] by applying patches as soon as they are available, while taking into account cost constraints, system availability and reliability [CCZ06; Iso]. It is therefore a technique for improving kernel self-protection, albeit a very reactive one.

Indeed, patch management can only "fix" vulnerabilities that are known (and for which a patch has been prepared): in that sense, it is the least proactive technique. Besides the obvious problem of attackers exploiting vulnerabilities that are not currently known, it can be very costly to create an infrastructure to apply patches as soon as they appear (as this requires continuous monitoring for the availability of such patches, and thorough and quick testing for regressions). For those reasons, more proactive techniques are necessary: by preventing or mitigating a portion of vulnerabilities before they are known to attackers, fewer patches will have to be applied in a timely manner, thus one can reduce the pressure on patch management.

2.2.2.2 Source Code Static Checkers

A simple approach to reduce the number of vulnerabilities in a given code base is to manually review the source code for known patterns of certain vulnerabilities: e.g., the improper bounds-checking on buffers in C programs. Of course, as soon as code bases become large, it becomes interesting to analyze the code-base for such known patterns using automated tools, which is referred to as static source code checkers.

---

while attackers can detect such vulnerabilities by examining binary or source patches.

[11] According to the National Vulnerability Database (NVD).

[12] The timespan from the introduction of a vulnerability to its removal.

Perhaps the simplest form of "static analysis" on the source, is the simple text search over the source code for potentially vulnerabilities. As simple as it may sound, this technique is often used by attackers and kernel security researchers: for instance, Dan Rosenberg found in 2010 a local privilege escalation vulnerability in the Linux kernel [Ros] by simply using `grep` for uses of the unsafe `__copy_to_user` function; Kees Cook also found a similar vulnerability in a similar way [Coo11].

Of course, more precise tools are required for checking for many common vulnerabilities, e.g., the use of attacker-supplied (or *tainted*) values as array index, or the use of a tented buffer as direct parameter to a format string. One such tool is CQual [Fos+03], which does require its user to annotate tainted and untainted variables by the use of special types.

CQual has been used successfully to find unsafe user/kernel pointer dereference vulnerabilities in the Linux kernel [JW04]. Prior to that, Engler et al. [Eng+00] have used an extensible compiler, `xg++`, to create extensions that specifically check certain classes of vulnerabilities, such as unsafe user pointer dereference, attacker-initiated deadlocks, double-frees.

Interestingly, as of 2013, neither of these two static analysis tools are in use as part of the build process of the Linux kernel[13]. Instead, Linux kernel developers are using: Sparse [Tor03] which can be described as a simpler version of the work of Johnson and Wagner [JW04]; Coccinelle [LMU05; Pad+08; PLM06] which is currently used to detect 22 different coding mistakes[14] ranging from deprecated usages to vulnerabilities; and some scripts for helping detect common defects, such as the `scripts/checkstack.pl` perl script that lists all kernel functions by stack-space usage — indeed, each process is allocated a fixed 4KB or 8KB kernel stack, with little run-time checks on stack usage, which can lead to security vulnerabilities such as CVE-2010-3848[15].

The limits of static analysis are well-known [CM04]. Basically, Rice's theorem indicates that, in the general case, it is possible to reduce static analysis (for any non-trivial program property) to the halting problem. This means that, in practice, static analysis is prone to false negatives or false positives (or both). It is therefore better to

[13] Or, to the best of my knowledge, by any other party.

[14] See Coccinelle scripts shipped with the Linux kernel sources, in the `scripts/coccinelle` sub-directory.

[15] Note that the CVE description incorrectly refers to this vulnerability as a "stack-based buffer overflow, whereas it should be qualified as a "stack overflow", or "stack exhaustion" vulnerability.

consider that static analysis tools are designed to help a human code reviewer, rather than automatically detect and fix vulnerabilities [Hee11]. Moreover, static analysis tools tend to be specialised and only detect classes of vulnerabilities which are known — especially for a program of the scale of an operating system kernel, many "new" or "specific" classes of vulnerabilities can be found over time.

As an example for unknown classes of vulnerabilities, the large class of vulnerabilities that are format strings did not stir much interest until the remote `wuftpd` exploit of tf8 (CVE-2000-0573). A similar remark can be made about Linux kernel NULL pointer dereferences, for which no effective mitigations existed until the 2009 `sock_sendpage` local privilege escalation exploit (CVE-2009-2692)[16]. As an example of an OS-specific vulnerability, Travis Ormandy found in the Windows NT kernel a vulnerability which allows switching of kernel stacks because of incorrect assumptions made in the general protection fault handler[17]. Such cases show the limit, in terms of proactiveness, of static analysis techniques.

2.2.2.3 Fuzz-testing and Symbolic execution

To go beyond the limits of static analysis, one can attempt to generate a large number of specially-crafted inputs to trigger vulnerabilities at run-time [MFS90]. The TRINITY fuzzer [Dav] does so by enumerating a large number of Linux system calls together with potentially interesting inputs and sequences: for instance, one needs to first open a file descriptor in order to make use of read and write system calls. Using this approach, many defects have been found in the Linux kernel (e.g., CVE-2013-2094 for which a private exploit existed).

A more systematic approach to reach a large number of program states (and potentially trigger corner-case vulnerabilities) is selective symbolic execution in the kernel [CKC11]. Renzelmann, Kadav, and Swift [RKS12] have applied this approach to find a significant number of vulnerabilities in Linux kernel drivers, without even requiring the hardware corresponding to those drivers to be present.

[16] The first mitigation was introduced in 2007, commit ed03218, and was available to SELinux users. However, it was possible to bypass this first mitigation easily until 2009. Note that this class of vulnerabilities existed since the early days of the Linux kernel, since Linux did not make use of segmentation for separating user and kernel address space.

[17] See `http://lists.grok.org.uk/pipermail/full-disclosure/2010-January/072549.html` for a detailed description.

The advantage of run-time techniques such as fuzz-testing and symbolic execution is the low likelihood of false-positives: if an attacker-supplied input is indeed crashing a program, it is likely that this was not in its specifications. In addition, such approaches can be very proactive in finding new vulnerabilities[18]. Path explosion, i.e., the problem of potentially having to execute an exponential number of program states in order to find a significant portion of vulnerabilities, is the main limitation of run-time approaches.

2.2.2.4 Exploit Mitigation

Another approach is to attempt to make vulnerabilities more difficult to exploit (typically to prevent privilege escalation). This usually entails making it difficult for attackers to transform a memory corruption vulnerability into an arbitrary code execution vulnerability, or to prevent memory corruption in the first place [Vee+12].

For instance, Secure Virtual Architecture (SVA) [Cri+07] compiles the existing kernel sources into a safe instruction set architecture, which is translated to native instructions by the SVA VM. This provides among other guarantees, a variant of type safety and control flow integrity for the Linux kernel.

The grsecurity and PaX team [St] have also produced numerous kernel exploit mitigation for Linux. This includes: (*a*) Kernel stack randomization, which randomizes the location of the kernel stack base at each system call. In particular this makes exploitation of kernel stack exhaustion vulnerabilities more difficult. (*b*) Kernel page execution protection, which "emulates" setting kernel data pages without the executable flag on architectures where this is not MMU-supported. This prevents some simple code injection-based payloads. A similar protection exists for return-to-userland attacks (a straightforward payload for memory-corruption-based privilege elevation is to include the payload in the userland exploit process, this prevents such payloads, and predates Intel's recent Supervisor Mode Execution Protection (SMEP) extensions which add hardware support to help achieving the same goal). (*c*) A comparable protection exists for data: improperly dereferencing userland pointers in kernel mode is a common vulnerability (with NULL-pointer dereferences occuring most often). PaX makes use of segmentation (when available) to provide kernel/userland memory separation and prevent exploitation

[18] In fact, Miller, Fredriksen, and So [MFS90] have presumably found the first instance of string format vulnerabilities thanks to fuzz-testing.

of such vulnerabilities[19].

Limitations of exploit mitigation solutions include that their scope is usually limited to a given class of vulnerabilities, or sometimes only to a specific type of payload. For example, injecting kernel code is not necessary to execute arbitrary code with an arbitrary kernel write vulnerability. Moreover, it is often difficult to securely recover from prevented attacks (or false positives) without crashing the kernel with such defenses [LAK09].

2.2.2.5 Extension Isolation

A number of approaches exist that retrofit micro-kernel–like features into monolithic OS kernels, mostly targeting fault isolation of kernel extensions such as device drivers [Cas+09; Mao+11; Swi+02]. For instance, the work of Swift et al. [Swi+02] wraps calls from device drivers to the core Linux kernel API (and vice-versa), as well as use virtual memory protection mechanisms, which leads to a more reliable kernel in the presence of faulty drivers. In the presence of a malicious attacker who can compromise such devices, however, this is in general insufficient. This can be mitigated with more involved approaches such as LXFI [Mao+11], which requires interfaces between the kernel and extensions to be annotated manually.

By deprivileging kernel modules, kernel extension isolation can greatly reduce the impact of some vulnerabilities [Mao+11]. However, it is also inherently limited by the privileges of the drivers: e.g., a hard-drive driver will be able to modify the on-disk filesystem data, which in many cases will be sufficient for a full system compromise. Similarly, in the absence of a properly configured IOMMU, drivers of DMA-capable devices are privileged to read and write to system memory. Nevertheless, extension isolation is a very proactive approach in the sense that it can potentially mitigate all types of vulnerabilities.

2.2.2.6 Attack Surface Reduction

Instead of deprivileging parts of the kernel, a simpler approach is to prevent unnecessary kernel code from being executed by attackers in the first place. Interestingly, although

[19] See the excellent description at `http://grsecurity.net/~spender/uderef.txt` for more details. Perhaps surprisingly to some, the technique is inspired by the use of segmentation in the Windows 95 kernel.

many existing approaches could be qualified as reducing kernel attack surface to some extent, very few approaches have been proposed to specifically reduce kernel attack surface. SECCOMP [Goo09] is one of such approaches, and tackles this problem by allowing processes to be sandboxed at the system call interface. Chapter 4 and Chapter 5 will present two new mechanisms for reducing kernel attack surface, and will compare them to existing approaches.

Similarly to extension isolation, attack surface reduction is a proactive approach to limit kernel vulnerabilities: if a vulnerability cannot be triggered by an attacker, it won't be exploitable and it does not matter which type of vulnerability is concerned.

2.2.2.7 Summary

None of the above approaches are, or claim to be, sufficient to eradicate all kernel vulnerabilities. However, each approach can prevent a significant number of vulnerabilities which do not overlap with those that can be prevented by other approaches. For instance, attack surface reduction will prevent all types of vulnerabilities that exist in rarely-used parts of the kernel, while static analysis can only detect some known classes of vulnerabilities. Conversely, static analysis can find vulnerabilities in commonly used critical parts of the kernel (such as the system call interface), while attack surface reduction may not. Hence, a systematic use of a combination of these techniques would provide the best results in terms of ensuring commodity kernel's protection. Perhaps surprisingly, attack surface reduction has not seen much interested so far, even though, as I show in this thesis, the benefits of attack surface reduction can be significant.

2.3 Kernel Protection Metrics

To paraphrase Lord Kelvin's often-cited aphorism, we can only improve that we can measure: quantifying is an important first step to understanding. However, quantifying security (especially in terms of the "perceived" level of security of the system, e.g., from a risk-management perspective) in a satisfactory manner is particularly difficult. Yet, from an engineering and research perspective, it is paramount to know how different technical solutions compare, and we now review how the effectiveness of OS kernel protection mechanisms has been quantified in the literature.

2.3.1 CVE studies

Perhaps the most common approach to quantify the effectiveness of kernel protection mechanisms is the use of CVEs. The authors would typically select a subset of all CVEs for the kernel, and, for each, evaluate whether their approach would have prevented the vulnerability.[20]

Chen et al. [Che+11] provide a good example of such a study for the Linux kernel, and manually evaluate different kernel security mechanism (not necessarily kernel protection mechanisms) according to the number of CVEs that would have been prevented, from a total of 141 CVEs. Their results show that most of the mechanisms they studied are insufficient in preventing kernel vulnerabilities, at least when taken individually (with SD [Cho+05] scoring second with 23 CVEs prevented, and SUD [BWZ10] scoring first with 29 CVEs prevented).

However, from the point of view of a system administrator these results might not be particularly interesting. SUD may have prevented CVEs in a lot of obscure drivers but the system that is being used might not even have the corresponding hardware, and no exploits would have been possible in the first place.

Beyond this issue, CVE data inherently suffer from selection bias, even within one given software such as the Linux kernel. For instance, vulnerability researchers might focus only on some parts of the kernel, e.g., because they are familiar with it [Kur+13]. Therefore the distribution of CVEs within the kernel might not necessarily reflect the distribution of kernel vulnerabilities attackers can be targeting.

2.3.2 Code Quality Metrics

Other authors may prefer using code quality metrics, such as source lines of code (SLOC), for judging the complexity of the code base and inferring the number of vulnerabilities that might exist in it. This is traditionally used in the Operating Systems community as a thumb-rule to compare the complexity [21] of kernels. when this is used to argue in favor of the security of one operating system versus another, there is usually several orders of magnitude of a difference: e.g., when comparing a micro-kernel such as

[20] This is not always done for purposes of quantification. Rather, the authors may merely aim to show anecdotal evidence that their approach can improve security by simply selecting a few vulnerabilities as a showcase: see for instance Mao et al. [Mao+11].

[21] In the layman sense of the word, not the algorithmic complexity

MINIX3 [Her+06b] (10K SLOC) to the Linux kernel (10M SLOC).

It has been shown that a correlation exists between the number of defects and code quality metrics [Che78; She+85; Sin+06; SW08]. In particular, a positive, although weak, correlation exists between CVE-based metrics and code quality metrics [Kur+13; SW08]. As such, performing source code analysis with code quality metrics can be a reasonable approach to quantify security — however, security comparisons between software should be handled with care: for instance studies such as [Kur+13; SW08] limit themselves to comparisons between different versions or configurations of the same software.

2.3.3 Static Checkers

Another approach is to use static checkers to find defects in an entire code repository directly. In their seminal work, Chou et al. [Cho+01] compare the number of defects in the Linux kernel across different versions and kernel subsystems. This type of analysis can be considered to produce more significant results (in estimating and comparing the number of exploitable security vulnerabilities) than CVE-based and code-quality metrics-based approaches.

Indeed, this approach does not suffer as much from selection bias, and the correlation to vulnerabilities is higher (since some defects detected are security-relevant). However, the approach does not take reachability of the code into account: for instance, for many, one key take-away of the Chou et al. [Cho+01] is that drivers are highly vulnerable. While this is correct in terms of the defects in the code base, this does not directly translate into exploitable vulnerabilities as we will see in this thesis. Indeed, many of those drivers might not be even compiled for the distribution kernel used by most users. Similarly, even when they are available as modules, often, an attacker cannot exploit a vulnerability in them unless the corresponding hardware for the driver is present on the machine.

2.4 Conclusion

Many classes of kernel security mechanisms exist, and each can be security-relevant in different threat models. However, the security of the kernel (kernel protection) is a precondition to ensuring that other services run correctly. In addition, many critical kernel vulnerabilities, leading in some cases to privilege escalation exploits that give

full control to attackers, have plagued commodity OS kernels over the years. Hence, improving kernel protection is a long-standing and pressing issue.

This thesis focuses on detecting and preventing such exploits, through kernel self-protection. Although many kernel self-protection mechanisms exist, little work has been done in attack surface reduction, or in defining kernel attack surface precisely. Chapter 3 lays the foundation to achieve this goal: it defines the kernel attack surface, and shows the general approach to measure attack surfaces.

3

Kernel attack surface: Quantification Method

> “I often say that when you can measure what you are speaking about, and express it in numbers, you know something about it”
>
> Lord Kelvin, *Electrical Units of Measurement*

The general idea of reducing attack surface to improve security is well known. Saltzer [Sal74] refers to the *economy of mechanism* principle, while Anderson [And72] refers to the reference monitor being small enough to be verifiable. It is also regarded as a recommended software development practice nowadays [OWA]. However, the notion of attack surface reduction is often vaguely defined.

This chapter starts by defining the kernel attack surface precisely, and then proceeds with quantification techniques (metrics). The quantification methods explained here will be used in the remainder of the thesis.

More precisely, we present three distinct security models, and, for each of them, security metrics that we use in this thesis to evaluate and quantify the security of a running Linux kernel. The dependence between notions defined or used here are summarized in Figure 3.1.

3.1 Preliminary definitions

Definition 1 (Call graph)**.** A *call graph* is a directed graph (F, C), where $F \subseteq \mathbb{N}$ is the set of nodes and represents the set of functions as declared in the source of a program, and $C \subseteq F \times F$ the set of arcs, which represent all direct and indirect function calls. We denote the set of all call graphs by $\mathscr{G}$.

In practice, static source code analysis at compile time (that takes all compile-time configuration options into account) is used to obtain such a call graph.

Definition 2 (Entry and barrier functions)**.** A security model defines a set of *entry functions* $E \subseteq F$, which corresponds to the set of functions directly callable by an attacker, and a set of *barrier functions* $X \subseteq F$, which corresponds to the set of functions that, even if reachable, would prevent an attacker from progressing further into the call graph.

E would typically be the interface of the program that is exposed to the attacker, whereas X would typically be the set of functions that perform authorization for privileges that the attacker is not assumed to have in the security model (e.g., administrator privileges).

Definition 3 (Attack Surface)**.** Given a call graph $G = (F, C)$, a set of entry functions $E \subseteq F$ and a set of barrier functions $X \subseteq F$, let G' be the subgraph of G induced by the nodes $F' = F \setminus X$, and let $E' = E \setminus X$. The *attack surface* is then the subgraph G_{AS} of G' induced by all nodes $f \in F'$ such that there exists $e \in E'$ and a directed path from e to f. By abusing notation, we denote $G_{AS} = (G, E, X)$.

Essentially, the attack surface represents the set of functions that an attacker can potentially take advantage of.

The rationale behind this definition is that for most types of kernel vulnerabilities due to defects in the source code, the attacker needs to trigger the function containing the vulnerability through a call to an entry function (which, for local attackers, would be a system call). For example: for exploiting a double-free vulnerability, the attacker will need to provoke the extraneous free; for exploiting a stack- or heap-based buffer overflow, the function writing to the buffer will be reachable to the attacker; for exploiting a user-pointer dereference vulnerability, the attacker owning the user-space process will often provoke the dereference through the system call interface.

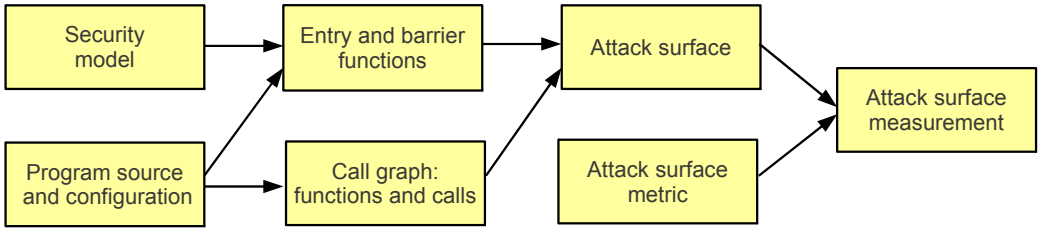


Figure 3.1. Dependencies between notions defined in this section.

Of course, the attacks could in some rare cases be indirect: e.g., for buffer-overflows, the attacker can provoke a write to a large buffer which is not overflown, but at a later point a kernel thread would copy this buffer into a smaller buffer, with improper bound checks. Clearly, in this case, the vulnerability is in the second function, which is not reachable tot he attacker, but the attacker still needs the first function to be reachable (hence the rationale still holds). We assume here, based on our knowledge of existing Linux kernel exploits, that such attacks are rare and do not significantly influence measurements.

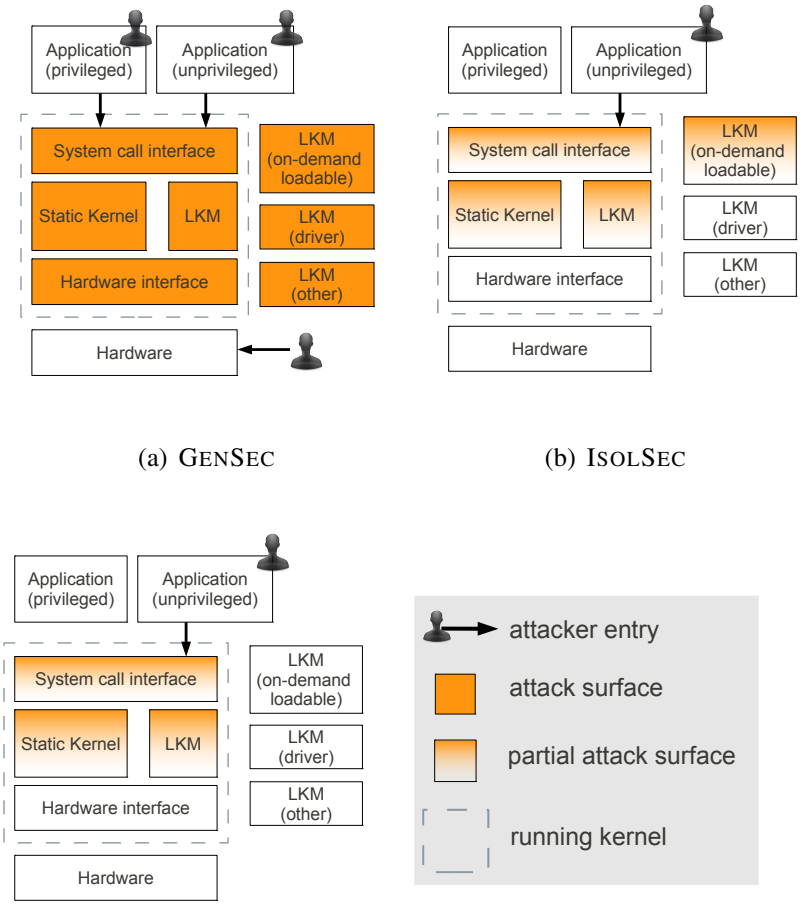
In practice, we first build a static call graph, at compile-time, of the Linux kernel using LLVM [LA04], frama-c [Fra] and ncc [Ncc] tools, for a given kernel configuration. The call graph takes function pointers into account, although conservatively (e.g., some function pointer targets may never be callable in practice, this stems from fundamental limitations of static analysis). This call graph is then used, knowing the entry points of the attacker (e.g., the system call interface), to perform a reachability analysis of the functions that an attacker could potentially call.

3.2 Security Models

Quantifying a program's security without specifying a security model is attractive because it provides an "absolute value" to compare other programs to. However, taking into account a security model, and more generally the actual use of the program, can only result in security metrics that reflect the system's security better. As a simple example, it is common practice to measure a kernel's security by the total SLOC. However, the source code will often contain branches that will never be compiled such as architecture-specific code for other architectures. Hence, limiting the SLOC by excluding unused architecture-specific code, because this code cannot be exercised by an attacker, is already

an improvement in precision.

We now consider the case of the Linux kernel. First, we define a generic security model that covers the dependability of the entire running kernel, and then a more specific model covering local attacks from unprivileged user space directed against the kernel. They are depicted in Figure 3.2.



(a) GENSEC (b) ISOLSEC

(c) STATICSEC

Figure 3.2. Three possible security models for quantifying kernel attack surface. GENSEC is a strawman security model, mainly for explanation purposes. ISOLSEC corresponds to a local unprivileged attacker on an unmodified Linux distribution. STATICSEC differs from ISOLSEC by assuming that no additional LKMs can be loaded once the attacker starts to perform its attack.

In both cases, the hardware and the compile-time configuration of the kernel are fixed and taken into account. In both cases, the high-level security goal is to provide

the traditional confidentiality, integrity and availability guarantees for the kernel: for instance, an attacker could target full control with arbitrary code execution in kernel mode, or more limited attacks such as information leakage (e.g., recover uninitialized kernel memory content) to breach confidentiality, and denial of service by crashing the kernel to reduce the system's availability. In addition, we assume that the hardware and the firmware the system is running on are trusted.

3.2.1 Generic Model GENSEC

The GENSEC model covers all possible kernel failures, to obtain an attack surface that is similar to the notion of TCB used for measuring security in prior work (e.g., [Har+05; McC+10]).

More precisely, the attacker is both local and remote, i.e., it has an account on the target system, but can also interact with all hardware devices (e.g., sending layer-1 traffic to network interface cards). We also assume that the attacker has some amount of control over a privileged application. This means the model includes failures due to defects in the kernel in code paths that are only accessibly from a privileged application.

Therefore, in this model, a defect in any part of the running kernel — including the core kernel and all loaded LKMs, as well as any LKM that might be loaded in the future, e.g., when new hardware is plugged in — can cause a failure.

This security model may not seem intuitive, but corresponds to what is implicitly assumed when considering the entire compiled kernel included in the TCB, a common practice.

GENSEC attack surface. In the GENSEC model above, the attack surface is composed of the entire running kernel, as well as LKMs that can be loaded. Hence, the barrier functions set X is empty, and all entry points of the kernel are included in E (both hardware interrupts and system calls, as well as kernel initialization code).

3.2.2 Isolation Model ISOLSEC

The ISOLSEC model reflects a common model in multi-user systems and in systems implementing defense in depth, where it is assumed an attacker has local access, e.g, by compromising an unprivileged isolated (or sandboxed) process on the system, and aims to escape the isolation by directly targeting the kernel. In this model, the attacker

is malicious and has unprivileged local access, therefore it can exercise the system call interface, but not all code paths: for instance, the attacker cannot make the system call for the insertion of a new kernel module. We will detail below, when evaluating the attack surface, exactly which barrier functions should be considered.

We also assume that the attacker can target code in LKMs, including LKMs that are loaded on-demand by the system. As the attacker is not able to plug hardware into the target system, we assume that bugs in LKMs not related to installed hardware cannot lead to failures.

ISOLSEC attack surface. An attacker in the ISOLSEC model has the set of all system calls as entry points E. The set of barrier functions X contains functions that are only accessible from privileged applications and LKMs that cannot possibly be loaded by an action triggered by the attacker. We provide a more detailed description of those functions in the next three paragraphs.

Functions that are not reachable because of lacking permissions are highly dependent on the isolation technology used (e.g., LSM-based [Coo10; HHT04; MMC06], chroot, LXC [Lxc], seccomp [Goo09]) and the policies applied to the application, and at a first approximation, we only consider the default privilege checking in use in the Linux kernel: POSIX capabilities. Hence, we assume that the set of barrier functions X includes those functions performing POSIX capability checks (functions calling the `capable()` function).

However, this is not sufficient. Linux proposes a variety of pseudo-filesystems, namely `sysfs`, `debugfs`, `securityfs` and `procfs`, in which filesystem operations are dispatched to specific code paths in the kernel, mostly in LKMs, and are often used to expose information or fine-tuning interfaces to user-space processes which, in general, are privileged. However, these privilege checks are performed at the virtual filesystem layer, using POSIX ACLs: hence, they do not contain calls to the `capable()` function, and need to be considered separately. In addition, as those filesystems should not be accessible from an unprivileged application that is sandboxed (e.g., this is the case even with a simple `chroot` jail), we include all functionality provided by those four pseudo-filesystems as barrier functions X.

Finally, as a consequence of our assumptions on LKMs in the ISOLSEC model, we include in X all LKMs that are either (a) not loaded while the workload is running, but not loadable on demand, or (b) a hardware driver that is not loaded while the workload is

running.

For these reasons, we mark in Figure 3.2 the kernel components which can contain funcions in the attack surface only as "partial attack surface": their inclusion depends on being reachable, after consideration of the barrier functions.

3.2.3 Alternative Isolation Model StaticSec

In contrast to the IsolSec model which assumes the attacker can trigger on-demand loading of some LKMs, the StaticSec model does not. Although on-demand LKMs loading by an attacker is realistic on many current Linux distributions by default, disabling this behavior is straightforward (e.g, by enabling the `modules_disabled` system control parameter available since Linux 2.6.31). Hence, the StaticSec model assumes only the LKMs loaded for the specific workload running on the machine are available to the attacker. All three models are summarized in Figure 3.2 for comparison.

StaticSec attack surface. The entry functions in E are the same as in the IsolSec model. The barrier function set X only differs in that the last step of LKM function removal is simplified such that we include in X all functions in LKMs that are not loaded while the workload is running.

3.3 Attack Surface Measurements

To quantify security improvements in terms of the attack surface, we need a metric that reflects its size. Although we are not the first to make this observation [HPW05; MW11], we propose the first approach that quantifies the attack surface within a particular security model by using call graphs. In the following, we present a general approach to measure an attack surface in a security model as well as specific metrics that we will use in the case of the Linux kernel.

Definition 4 (Code-quality metric)**.** A *code-quality metric* μ is a mapping associating a non-negative value to the nodes of the call graph:

$$\mu : F \to \mathbb{R}^+$$

Example. A function's SLOC (denoted *SLOC*), the cyclomatic complexity [McC76]

(denoted *cycl*), or a CVE-based metric associating the value 1 to a function that had a CVEs in the past 7 years, and 0 otherwise (denoted CVE), are code-quality metrics that we use in this thesis.

CVE-based metrics provide *a posteriori* knowledge on vulnerable functions: they allow an estimate of the number of CVEs an attack surface reduction method would have avoided, in the past. However, this metric, alone, is unsatisfactory for multiple reasons. For instance, CVEs only form a sample of all vulnerabilities existing in an application, and this sample is likely to be biased: vulnerabilities tend to be searched and discovered non-uniformly across the code base, with often-used parts being more likely to be tested and audited. Additionally, past CVEs are not necessarily a good indicator of future CVEs: although a function with a history of vulnerabilities might be prone to more vulnerabilities in the future (e.g., due to sloppy coding style), the opposite is also likely, since this might indicate that the function has now been thoroughly audited. For this reason, we also use *a priori* metrics such as lines of code and cyclomatic complexity, which, although imperfect for predicting vulnerabilities, do not suffer from the aforementioned issues and can be easily collected through static analysis.

Definition 5 (Attack Surface Metric)**.** An *attack surface metric* associated with a code-quality metric μ assigns a non-negative real value to an attack surface:

$$\begin{aligned} AS_\mu : \mathscr{G} &\to \mathbb{R}^+ \\ G_{AS} &\mapsto AS_\mu(G_{AS}) \end{aligned}$$

and satisfies the property:

$$\begin{aligned} &\forall E' \subseteq E, \forall X' \supseteq X,\ AS_\mu(G'_{AS}) \leq AS_\mu(G_{AS}) \\ &\text{with} \quad G_{AS} = (G,E,X), G'_{AS} = (G,E',X') \end{aligned}$$

That is, the more entry points, the higher the attack surface measurement; the more barrier functions, the lower the attack surface measurement.

Lemma 1. *Let m be a mapping:*

$$\begin{aligned} m : \mathscr{G} &\to \mathbb{R} \\ G &\mapsto m(G) \end{aligned}$$

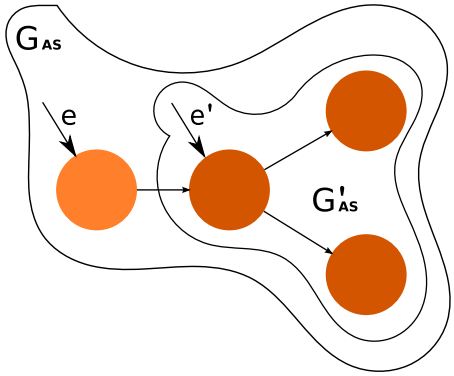


Figure 3.3. Example attack surfaces G_{AS} (with $E = \{e\}$ and $X = \emptyset$) and G'_{AS} (with $E' = \{e'\}$ and $X' = \emptyset$). Note that $E' \not\subseteq E$ and $G'_{AS} \subseteq G_{AS}$.

If m satisfies:

$$\forall G \in \mathscr{G}, \quad m(G) \geq 0$$

$$\forall G' \subseteq G \in \mathscr{G}, \quad m(G') \leq m(G)$$

then it is an attack surface metric.

Proof. Let $G_{AS} = (G,E,X), G'_{AS} = (G,E',X')$ such that $E' \subseteq E$ and $X' \supseteq X$. Then:

$$G'_{AS} \subseteq G_{AS}$$

Hence m satisfies the property in Definition 5:

$$m(G'_{AS}) \leq m(G_{AS})$$

□

Note that this property is not necessary to satisfy Definition 5, because a smaller set of functions (in G'_{AS}) should not necessarily mean a smaller attack surface measurement. This is sensible, because in practice some functions can reduce the overall attack surface (e.g., by sanitizing input), and an attack surface metric could take this into account (e.g., Murray, Milos, and Hand [MMH08] propose such a metric for measuring TCB size). Such an example is depicted in Figure 3.1: A metric satisfying Lemma 1 would always measure a lower attack surface for G'_{AS} than for G_{AS}, whereas this is not necessary for a metric satisfying Definition 5.

Proposition 1. *The following two functions are attack surface metrics:*

$$AS1_{\mu}(G_{AS}) = \sum_{i \in F_{AS}} \mu(i)$$

$$AS2_{\mu}(G_{AS}) = {\mu_{AS}}^{T} L(\widetilde{G_{AS}}) \mu_{AS}$$

where $G_{AS} = (F_{AS}, C_{AS})$, ${\mu_{AS}}^{T} = (\mu(1), \ldots, \mu(|F|))$, *and* $L(G)$ *is the Laplacian matrix of a simple (non-directed) graph:*

$$L(G) = D - A$$

where D is a diagonal matrix with the degrees of the nodes on the diagonal, and A the adjacency matrix of the graph ($A_{ij} = 1$ *when the (i,j) edge exists,* 0 *otherwise). As* G_{AS} *is directed, we transform it into a simple graph by ignoring the direction on its arcs, which we denote* $\widetilde{G_{AS}}$.

$AS1$ provides a simple and intuitive formulation of an attack surface metric: for instance, $AS1_{SLOC}$ is a sum of the lines of code in the attack surface. However, it values each function equally. $AS2$ takes advantage of the functions position in the call graph, and attaches more value to code-quality metrics in functions that have a large number of callers (and callees) that have a lower code-quality measurement. The Appendix contains a proof, and a more detailed explanation of the formulation of $AS2$. We use both these attack surface metrics in various evaluations in this thesis. However, we primarily use $AS1$, because it has a more intuitive interpretation.

3.4 Related Work

The need for better security metrics is widely accepted in both academia and industry [Bel06; Jaq07; SBE11; Sch99]. Howard, Pincus, and Wing [HPW05] were the first to propose the use of code complexity and bug count metrics to compare the relative "attackability" of different software, and others have followed [MW11; Sin+06; SW08]. Murray, Milos, and Hand [MMH08] underline the fact that TCB size measurements by SLOC, while good, might not be precise enough because additional code can sometimes reduce the attack surface (e.g., sanitizing input). Manadhata and Wing [MW11] present

an attack surface metric based on an insightful I/O automata model of the target system, taking into account in particular the data flow from untrusted data items and the entry points of the system. The definition of attack surface used in their work closely relates to ours, with the differences that our modeling is solely based on static call graphs and a measure of code complexity of each underlying function. In contrast, this work measures the attack surface with respect to a particular attacker model.

3.5 Summary

The metrics introduced in this section are for the purpose of a precise evaluation of the security gains of our approach. These metrics contain metrics used commonly in prior work, such as total TCB size in SLOC ($AS1_{SLOC}$ in the GENSEC model). We do not claim the metrics presented in this section are the panacea in measuring attack surfaces. Rather, we propose new metrics that take into account what attackers are capable of. This will allow us to discuss attack surface reduction results in Section 6.2.4 for additional insights into the advantages and disadvantages of tailoring the Linux kernel configuration.

This `NULL` *page intentionally mapped.*

4 Compile-time Kernel-Tailoring

> “If you don’t have a dog, your neighbor can’t poison it.”
>
> Sergey Nikitin, *If You Don’t Have an Aunt* (Russian song)

The first approach to reduce the kernel attack surface leverages compile-time kernel configuration options. The Linux kernel is highly configurable [Tar+11], with over 5000 features one can chose from when compiling a Linux kernel. Popular Linux distributions cannot afford to ship and maintain a large number of different kernels. Therefore, they configure their kernels to be as generally usable as possible, which requires the kernel package maintainers responsible to enable as much functionality (i.e., KCONFIG features) as possible. Unfortunately, this also maximizes the attack surface. As many security-sensitive systems do not require the genericalness provided (e.g., embedded systems, hypervisors in cloud infrastructures), the attack surface can be reduced by simply disabling unnecessary features. What features are necessary, however, depends on the actual workload of the corresponding use-case. Therefore, our approach consists of two phases. In the analysis phase, the workload is analyzed at run time. The second phase calculates a reduced Linux configuration that enables only the functionality that has been observed in the analysis phase.

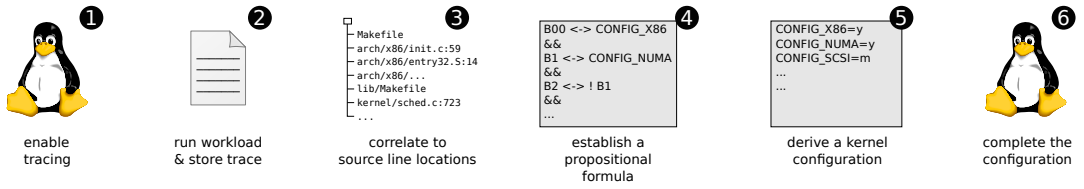


Figure 4.1. Kernel tailoring steps.

4.1 Design

4.1.1 Configuration Mechanisms in Linux

Many kernel users are familiar with the Linux kernel's build-time configuration menu. This menu allows one to select features will be compiled into the kernel. The dependencies between these features is specified through a Domain-Specific Language (DSL) called KCONFIG, which has been extensively studied [SB10; ZK10].

It is in theory possible to use this configuration menu to manually select and deselect the relevant kernel configuration options and achieve a minimal set of kernel configuration options. However, due to the large number of features in recent Linux kernels (about 5000 features for Ubuntu 12.04), this is very difficult to achieve even for highly skilled and experienced users. For instance, Linus Torvalds sent an e-mail to the kernel mailing list asking to simplify the configuration of distribution kernels, as it is very difficult to know exactly what a distribution requires to function properly [Tor12].

4.1.2 Kernel Tailoring

The goal of our approach is to automatically compile a Linux kernel with a configuration that has only those features enabled which are necessary for a given use case. This section shows the fundamental steps of our approach to tailor such a kernel. The six steps necessary are shown in Figure 4.1.

❶ **Enable tracing.** The first step is to prepare the kernel so that it records which parts of the kernel code are executed at run time. We use the Linux-provided FTRACE feature, which is enabled with the KCONFIG configuration option `CONFIG_FTRACE`. Enabling this configuration option modifies the Linux build process to include profiling code that can be evaluated at runtime.

In addition, our approach requires a kernel that is built with debugging information

such that any function addresses in the code segment can be correlated to functions and thus source file locations in the source code. For Linux, this is configured with the KCONFIG configuration option `CONFIG_DEBUG_INFO`.

The fact that the for instance the Linux Kernel that is shipped by Ubuntu ships with this feature turned on shows that the performance overhead of FTRACE is negligible for all typical usages. For this reason, in our experiments we use Ubuntu: this allows us to use the pre-shipped kernel for tracing.

To also cover code that is executed at boot time by initialization scripts, we need to enable the FTRACE as early as possible. For this reason, we modify the initial RAM disk, which contains programs and LKMs for low-level system initialization [1]. Linux distributions use this part of the boot process to detect installed hardware early in the boot process and, mostly for performance reasons, load only the required essential device drivers. This basically turns on tracing even before the first process (`init`) starts.

❷ **Run workload.** In this step, the system administrator runs the targeted application or system services. The FTRACE feature now records all addresses in the text segment that have been instrumented. For Linux, this covers most code, except for a small amount of critical code such as interrupt handling, context switches and the tracing feature itself.

To avoid overloading the system with often accessed kernel functions, FTRACE's own ignore list is dynamically being filled with functions when they are used. This prevents such functions from appearing more than once in the output file of FTRACE. We use a small wrapper script for FTRACE to set the correct configuration before starting the trace, as well as to add functions to the ignore list while tracing and to parse the output file, printing only addresses that have not yet been encountered.

During this run, we copy the output of the tracing wrapper script at constant time intervals. This allows us to compare at what time what functionality was accessed, and therefore to monitor the evolution of the tailored kernel configuration over time based on these snapshots.

❸ **Correlation to source lines.** A system service translates the raw address offsets into source line locations using the ADDR2LINE tool from the `binutils` tool suite. Because LKMs are relocated in memory depending on their non-deterministic order of loading, the system service compares the raw, traced addresses to offsets in the LKM's code segment. This allows the detection of functionality that is not compiled statically

[1] This part of the Linux plumbing is often referred to as "early userspace"

into the Linux kernel. This correlation of absolute addresses in the code segment with the debug symbols allows us to identify the source files and the `#ifdef` blocks that are actually being executed during the tracing phase.

❹ **Establishment of the propositional formula.** This step translates the source-file locations into a propositional formula. The propositional variables of this formula are the *variation points* the Linux configuration tool KCONFIG controls during the compilation process. This means that every C Preprocessor (CPP) block, KCONFIG item and source file can appear as propositional variable in the resulting formula. This formula is constructed with the variability constraints extracted from `#ifdef` blocks, KCONFIG feature descriptions and Linux Makefiles. The extractors we use have been developed, described and evaluated in previous work [Die+12; Sin+10; Tar+11]. The resulting formula holds for every KCONFIG configuration that enables all source lines simultaneously.

❺ **Derivation of a tailored kernel configuration.** A SAT checker proves the satisfiability of this formula and returns a concrete configuration that fulfills all these constraints as example. Note that finding an optimal solution to this problem is an NP-hard problem and was not the focus of our work. Instead, we rely on heuristics and configurable search strategies in the SAT checker to obtain a sufficiently small configuration.

As the resulting kernel configuration will contain some additional unwanted code, such as the tracing functionality itself, the formula allows the user to specify additional constraints to force the selection (or deselection) of certain KCONFIG features, which can be specified in whitelists and blacklists. This results in additional constraints being conjugated to the formula just before invoking the SAT checker.

❻ **Completing the Linux kernel configuration.** The resulting kernel configuration now contains all features that have been observed in the analysis phase. The caveat is that the resulting propositional formula can only cover KCONFIG features of code that has been traced. In principle, features that are left unreferenced are to be deselected. However, features in KCONFIG declare non-trivial dependency constraints [ZK10], which must all hold for a given configuration in order to produce a *valid* KCONFIG configuration. The problem of finding a feature selection with the smallest number of enabled features, (which is generally not unique) has the complexity *NP-hard*. We therefore rely on heuristics to find a sufficiently small configuration that satisfies all constraints of KCONFIG but is still significantly smaller compared to a generic distribution kernel.

4.2 Evaluation

In this section, we present two use cases, namely a Linux, Apache, MySQL and PHP (LAMP)-based server and a graphical workstation that provides an network file system (NFS) service, both on distinct, non-virtualized hardware, that we use to evaluate the effects of kernel-configuration tailoring. This evaluation demonstrates the approach with practical examples, verifies that the obtained kernel is functional, i.e., no required configuration option is missing in the tailored kernel, and shows that the performance of the kernel with the configuration generated remains comparable to that of the distribution kernel. We quantify the attack surface reduction achieved with the formalisms described in Chapter 3.

4.2.1 Overview

In both use cases, we follow the process described in Section 4.1.2 to produce a kernel configuration that is tailored to the respective use case. For each use case, we detail the workload that is run to collect traces in the following subsections. Both machines use the 3.2.0-26 Linux kernel distributed by Ubuntu as baseline, which is the kernel shipped at the time of this evaluation in Ubuntu 12.04.

To compare the performance, we use benchmarks that are specific to the use case. We repeat both experiments at least 10 times and show 95%-confidence intervals in our figures where applicable. The benchmarks compare the original, distribution-provided kernel to the tailored kernel generated. All requests are initiated from a separate machine over a gigabit Ethernet link. To avoid interferences by start-up and caching effects right after the system boots, we start our workload and measurements after a warm-up phase of 5 min.

To measure the attack surface reduction, we first calculate code-quality metrics for each function in the kernel by integrating the FRAMA-C [Fra] tool into the kernel build system. For CVEs, we parse all entries for the Linux kernel in the NVD[2]. For entries with a reference to the GIT repository commit (only those CVEs published after 2005), we identify the C functions that have been changed to patch a security issue, and add each function to a list. Our metric assigns a value of 1 to functions that are in this list, and 0 otherwise. We also generate static call graphs for each use case by using both FRAMA-C

[2] http://nvd.nist.gov/

and NCC [Ncc] and combining both call graphs to take into account calls through function pointers, which are very widely used in the Linux kernel. In the case of the GENSEC model, we compute the *AS*1 and *AS*2 attack surface metrics directly over all functions in this graph, for both the baseline and the tailored kernel. In the case of the ISOLSEC model, we compute the subgraph corresponding to the attack surface by performing a reachability analysis from functions corresponding to system calls (entry points) and removing all barrier functions as detailed in Section 3.2.2. We then evaluate the security improvements by computing the attack surface reduction between the baseline kernel and a tailored kernel.

4.2.2 LAMP-stack use case

4.2.2.1 Description

This use case employs a machine with a 2.8 GHz Celeron CPU and 1 GB of RAM. We use the Ubuntu 12.04 Server Edition with all current updates and no modifications to either the kernel or any of the installed packages. As described in Section 4.1.2, we extend the system-provided initial RAM disk (`initrd`) to enable tracing very early in the boot process. In addition, we set up an web platform consisting of APACHE2, MYSQL and PHP. The system serves static documents, the collaboration platform DOKUWIKI [Goh] and the message board system PHPBB3 [Php] to simulate a realistic use case.

The test workload for this use case starts with a simple HTTP request using the tool WGET, which fetches a file from the server right after the five-minute warm-up phase. This is followed by one run of the HTTPERF [MJ98] tool, which accesses a static website continuously, increasing the number of requests per second for every run. Finally, we run the SKIPFISH [ZHR] security scan on the server. SKIPFISH is a tool performing automated security checks on web applications, hence exercising a number of edge-cases, which is valuable not only to exercise as many code paths as possible, but also to test the stability of the tailored use case.

4.2.2.2 Results

Figure 4.2 depicts the number of KCONFIG features that our tool obtains from the trace logs collected at the times given. After the warm-up phase, connecting to the server via `ssh` causes a first increase in enabled KCONFIG features. The simple HTTP request

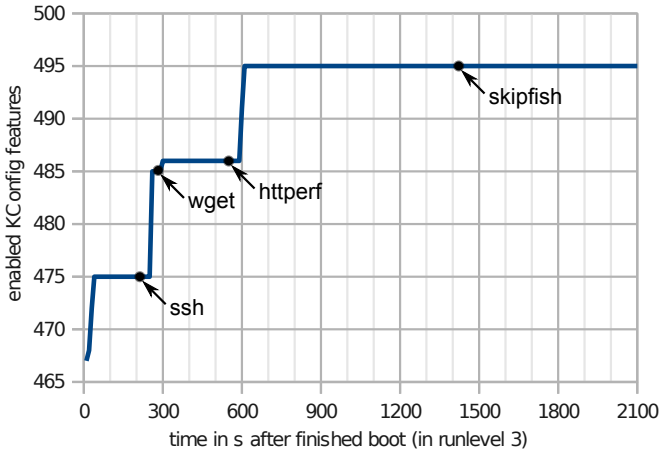


Figure 4.2. Evolution of KCONFIG features enabled over time. The bullets mark the point in time at which a specific workload was started.

triggers only a small further increase, and the configuration converges quickly after the HTTPERF tool is run, and shows no further changes when proceeding to the SKIPFISH scan. This shows that, for the LAMP use case, a tracing phase of about ten minutes is sufficient to detect all required features.

Tailoring. The trace file upon which the kernel configuration is generated is taken 1,000 sec after boot, i.e., after running the tool HTTPERF, but before running the SKIPFISH tool. It consists of 8,320 unique function addresses, including 195 addresses from LKMs. This correlates to 7,871 different source lines in 536 files. Our prototype generates the corresponding configuration in 145 seconds and compiles the kernel in 89 seconds on a commodity quad-core machine with 8 GB of RAM.

When comparing the original kernel to the distribution kernel shipped with Ubuntu, we observe a reduction of KCONFIG features that are statically compiled into the kernel of over 70%, and almost 99% for features that lead to compilation as LKMs (cf. Table 4.1). Consequently, the overall size of the text segment for the tailored kernel is over 90% lower than that of the baseline kernel supplied by the distribution.

To relate to the savings in terms of attack surface, we show the number of source code files that the tailored configuration does not include when compared to the distribution configuration in Figure 4.3. The figure breaks down the reduction of functionality by subdirectories in terms of source files that get compiled. The highest reduction rates are observed inside the `sound/` (100%), `drivers/` (95%), and `net/` (87%) directories. As the

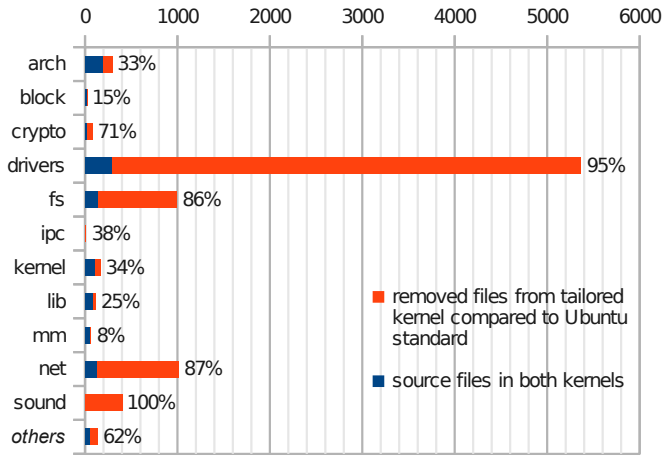


Figure 4.3. Reduction in compiled source files for the tailored kernel, compared with the baseline in the LAMP use case (results for the workstation with NFS use case are similar). For every subdirectory in the Linux tree, the number of source files compiled in the tailored kernel is depicted in blue and the remainder to the number in the baseline kernel in red. The reduction percentage per subdirectory is also shown.

web server does not play any sounds, the trace file does not indicate any sound-related code. Similarly, the majority of drivers are not needed for a particular hardware setup. The same applies to most of the network protocols available in Linux, which are not required for this use case. Out of 8,670 source files compiled in the standard Ubuntu distribution kernel, the tailored kernel only required 1,121, which results in an overall reduction of 87% (cf. Table 4.1).

Stability. To ensure that our tailored kernel is fully functional, we run SKIPFISH [ZHR] once on the baseline kernel and then compare the results to a scan on the tailored kernel. The report produced by the tool finds no significant difference from one kernel configuration to the other, hence the tailored kernel can handle unusual web requests equally well. Furthermore, this shows that for this use case even a kernel tailored from a trace file which only covers a smaller test workload than the target scenario is suitable for stable operation of the service.

Performance. We measure the performance with the HTTPERF tool. The result is compared with a run performed on the same system that runs the baseline kernel. Figure 4.4 shows that the tailored kernel achieves a performance very similar to that of the kernel provided by the distribution.

Security. Finally, we compute attack surface reduction with *AS*1 and *AS*2 in the

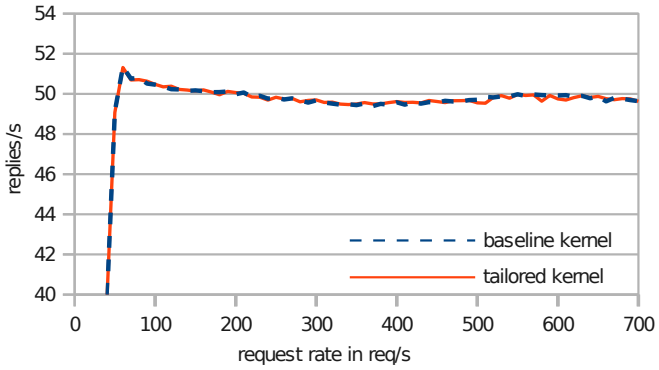


Figure 4.4. Comparison of reply rates of the LAMP-based server using the kernel shipped with Ubuntu and our tailored kernel. Confidence intervals were omitted, as they were too small and thus detrimental to readability.

GENSEC and ISOLSEC models after generating the relevant call graphs. The numbers in Table 4.1 show that the $AS1_{SLOC}$, $AS1_{cycl}$ and $AS2_{SLOC}$ attack surface reduction is around 85% in the GENSEC model, and around 80% in the ISOLSEC model. In both models, there are also 60% fewer functions that were affected by patches due to CVEs in the past. We also observe that $AS2_{cycl}$ is slightly lower, with an attack surface reduction around 60%. Overall, the attack surface reduction is between 60% and 85%.

4.2.3 Workstation/NFS use case

4.2.3.1 Description

For the workstation/NFS server use case, we use a machine with a 3.4 GHz quad-core CPU and 8 GB of RAM, running the Ubuntu 12.04 Desktop edition, again without modifications to packages or kernel configuration. The machine is configured to export a local directory via NFS.

To measure the performance of the different kernel versions, we use the BONNIE++ [Cok] benchmark, which covers reading and writing to this directory over the network. To achieve results that are meaningful, we disable caching on both server and client.

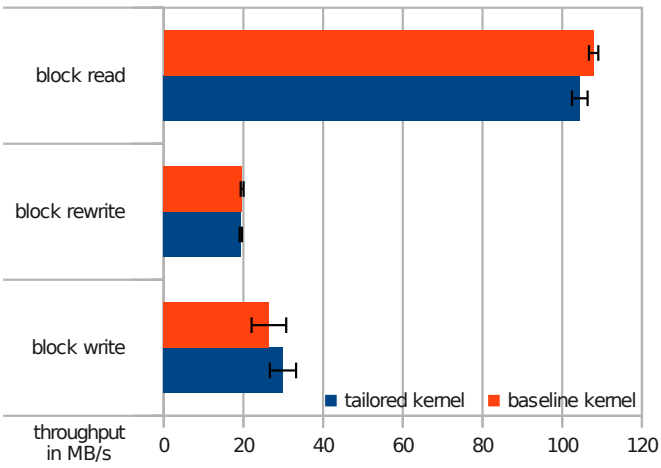


Figure 4.5. Comparison of the test results from the BONNIE++ benchmark, showing no significant difference between the tailored and the baseline kernel.

4.2.3.2 Results

The trace file of the configuration selected for further testing consists of 13,841 lines that reference a total of 3,477 addresses in modules. This resolves to 13,000 distinct source lines in 735 files. Building the formula and therefore the configuration takes 219 seconds, compiling the kernel another 99 seconds on the same machine as described above. We observe a reduction of KCONFIG features that are statically compiled into the kernel by 68%, 98% for features compiled into LKMs, and about 90% less code in the text segment.

Performance and Stability. We did not find any impact on the regular functionality of the workstation, i.e., all hardware attached, such as input devices, Ethernet or sound, remained fully operable with the tailored kernel booted. Using the tailored kernel, we run BONNIE++ again with the same parameters, and compare the results with those of the distribution kernel. Figure 4.5 shows that also in this use case the kernel compiled with our tailored configuration achieves a very similar performance.

Security. Attack surface reduction results are similar to the LAMP use case. The numbers in Table 4.1 show that the $AS1_{SLOC}$, $AS1_{cycl}$ and $AS2_{SLOC}$ attack surface reduction is around 80% in the GENSEC model, and around 75% in the ISOLSEC model. In both models, there are also 50% fewer functions that were affected by patches due to CVEs in the past. We also observe that $AS2_{cycl}$ is slightly lower as well, with attack surface reduction around 60%. Overall, our measurements suggest the attack surface

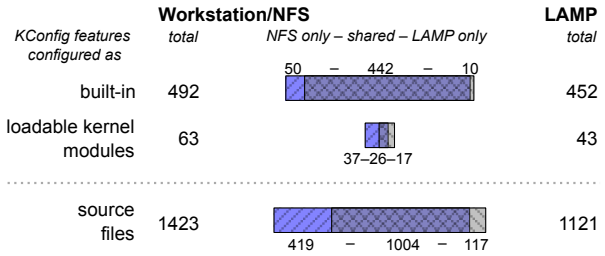


Figure 4.6. Comparison of the two generated configurations from the use cases in terms of KCONFIG features leading to built-in code and code being compiled as LKM. Below, the total number of compiled source files is compared between the two resulting kernels.

reduction is between 50% and 80%.

4.3 Discussion

4.3.1 Attack surface measurements

This section discusses the results of our attack surface measurements.

Use cases. Figure 4.6 shows that the tailored kernel configurations are largely similar for both cases. We observe a number of features that differentiate the use cases, both in terms of hardware and workload. The workstation/NFS use case requires the highest number of differentiating features (87 enabled KCONFIG options for NFS compared to 27 for LAMP). This can be explained by the setup (the desktop version of Ubuntu has the X11 window system installed and running, whereas the server version has not) and by the workload: as NFS also runs in kernel mode, additional kernel features are required. This point is useful for understanding attack surface reduction results. Although both use cases show similar $AS1_{SLOC}$ reductions (around 80%), there is a slight difference for both GENSEC and ISOLSEC and the various *AS* metrics in the reduction achieved in favor of the LAMP use case (see Table 4.1). This is simply because the workstation/NFS use case requires a larger kernel than the LAMP one.

Supplementary anecdotal evidence. Out of the 422 CVEs we have inspected, we detail the case of one highly publicized vulnerability for illustration purposes. CVE-2010-3904 documents a vulnerability that is due to a lack of verification of user-provided pointer values, in RDS, a rarely used socket type. An exploit for obtaining local privilege escalation was released in 2010 [Ros]. We verified that in the case of the workstation/NFS

use case, both tailored kernel configurations have the functionality removed, and thus would have prevented the exploitation of the vulnerability in the GENSEC and ISOLSEC models. In contrast, the baseline kernel contains the previously-vulnerable feature in the GENSEC and ISOLSEC models. Table 4.2 shows this result, as well as results for three other vulnerabilities presented in Chapter 2.

CVE sampling bias. The results in Table 4.1 show slightly lower CVE reduction numbers than for all other metrics, especially in the case of *AS*1. We hypothesize that this small difference is due to a sampling bias: code that is used more often is also audited more often, more bug reports concerning it are submitted, and better care is taken in documenting the vulnerabilities of such functions. We also observe the average number of CVEs per function is lower in the functions that are in the tailored kernel, when compared to those functions that are not. Previous studies [Cho+01; Pal+11] have shown that code in the `drivers/` sub-directory of the kernel, which is known to contain a significant amount of rarely used code, on average contains significantly more bugs than any other part of the kernel tree. Consequently, it is likely that unused features provided by the kernel still contain a significant amount of relatively easy-to-find vulnerabilities. This further confirms the importance of attack surface reduction as presented in this paper.

Nevertheless, we still take the CVE reduction numbers into account, because they reflect a posteriori knowledge about vulnerability occurrences. All our measurements indicate attack surface reduction lies approximately within 50% and 85% across all parameters (use cases, security models, metrics), which is a very positive result for kernel tailoring.

Attack surface metric comparison. The *AS*1 and *AS*2 results are quite close, which, considering how different their formulations are, shows the robustness of the simple attack surface definition introduced in Chapter 3. *AS*2 is also of interest because it introduces the use of the Laplacian, which is instrumental in many applications of graph theory (e.g., for data mining [BN01]), for the purpose of attack surface measurements.

Comparison to kernel extension isolation. Approaches such as [Mao+11; Swi+02] provide a way, through impressive technical feats, of isolating LKMs from the kernel, i.e., running them with lesser privileges. This means, ideally, the compromise of an LKM by an attacker cannot lead to kernel compromise. To evaluate how such solutions compare to kernel tailoring, we again make use of the attack surface formalism introduced

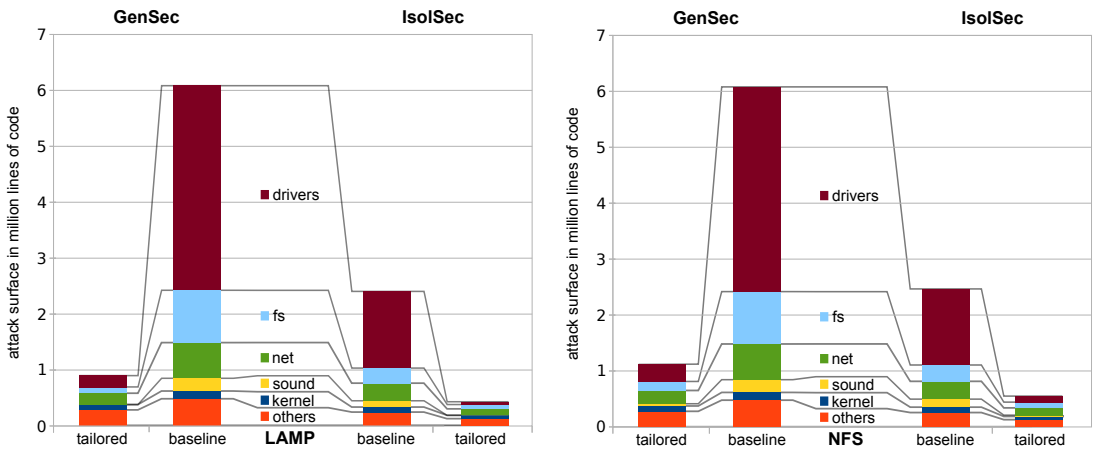


Figure 4.7. $AS1_{SLOC}$ attack surface measurements per kernel subsystem in both security models and use cases.

in Chapter 3. Assuming that these isolation solutions are ideal (i.e., that their own implementation does not increase the kernel's attack surface and the attackers are not able to bypass the isolation), we remove all LKMs from the baseline kernel's attack surface in the IsolSec model, hence obtaining a lower bound of the real attack surface of such LKM-isolated kernels. Our results in Table 4.3 show that kernel tailoring is superior to LKM isolation: for instance, the $AS1_{SLOC}$ measurement of the ideal LKM isolation is four times higher. We also evaluate whether combining both approaches could be beneficial, i.e., first generating a tailored kernel and then applying an ideal LKM isolation. The results show that the resulting attack surface is not significantly lower than that obtained by kernel tailoring alone, which further confirms the improvements of our approach, even when compared to an ideal LKM isolation solution. Additionally, we remark that this lower bound is also applicable to approaches that prevent automatic-loading of LKMs, such as the well-known grsecurity kernel patch with the `MODHARDEN` option [St].

Security models. The attack surface reduction is important in both security models, but more so in the GenSec model. This can be attributed to the fact that the GenSec model includes a large number of drivers, whereas the IsolSec model does less. As can be seen from Figure 4.7, the attack surface reduction is particularly high for drivers. In other words, tailoring appears to be slightly more effective in the GenSec model than in the IsolSec model. This is to be expected, since our approach reduces the kernel's attack surface system-wide (and not per-process). Figure 4.7 also shows that, both in the baseline and tailored kernels and independently of the use case, the IsolSec attack

surface is about half of the GENSEC attack surface. In other words, the attack surface of a local attacker (as defined in the ISOLSEC model) is about half of what is generally considered as the TCB of the kernel.

Importance of kernel configuration. When quoting SLOC measurements of the Linux kernel as a simple way of quantifying TCB size, we advocate specifying the kernel configuration the measurement corresponds to. Indeed, our results show that, depending on the kernel configuration, the total number of lines of code can vary by up to an order of magnitude. An other important factor is the kernel version, since the Linux kernel increased significantly in size over the past years.

4.3.2 Kernel tailoring

We will discuss now the key strengths and weaknesses of the kernel-tailoring tool with respect to various properties.

Effectiveness. Although in absolute terms the attack surface of the tailored Linux kernel remains high (for *AS*1, about 500K SLOC in the ISOLSEC model, and 1000K SLOC in the GENSEC model), Table 4.1 shows that for both use cases and across all meaningful metrics, the attack surface is reduced by almost an order of magnitude. As such, vulnerabilities existing in the Linux kernel sources are significantly less likely to impact users of a tailored kernel. This makes the approach presented an effective means for improving security in various use cases.

Applicability. The approach presented relies on the assumption that the use case of the system is clearly defined. Thanks to this a-priori knowledge, it is possible to determine which kernel functionalities the application requires and therefore, which kernel configuration options have to be enabled. With the increasing importance of compute clouds, where customers use virtual machines for very dedicated services such as the LAMP stack presented in Section 4.2, we expect that our approach will prove valuable for improving the security in many cloud deployments.

Usability. Most of the steps presented in Section 4.1.2 require no domain-specific knowledge of Linux internals. We therefore expect that they can be conducted in a straightforward manner by system administrators without specific experience in Linux kernel development. The system administrator, however, continues to use a code base that constantly receives maintenance in the form of bug fixes and security updates

from the Linux distributor. We therefore are confident that our approach to tailor a kernel configuration for specific use-cases automatically is both practical and feasible to implement in real-world scenarios.

Extensibility. The experiments in Section 4.2 show that, for proper operation, the resulting kernel requires eight additional KCONFIG options, which the `ftrace` feature could not detect. By using a whitelist mechanism, we demonstrate the ability to specify wanted or unwanted KCONFIG options independently of the tracing. This allows our approach to be assisted in the future by methods to determine kernel features that tracers such as FTRACE cannot observe.

Safety. Many previous approaches that reduce the Linux kernel's TCB (e.g., [Goo09], [KSK11]) introduce additional security infrastructure in form of code that prevents functionality in the kernel from being executed, which can lead to unexpected impacts and the introduction of new defects into the kernel. In contrast, our approach modifies the kernel configuration instead of changing the kernel sources (e.g., [Lee+04; St]) or modifying the build process (e.g., [Cri+07]). In that sense, our approach, by design, cannot introduce new defects into the kernel.

However, as the configurations produced are specific to the use case analyzed in the tracing phase, we cannot rule out that the tailored configuration uncovers bugs that could not be observed in the distribution-provided Linux kernel. Although we have not encountered any of such bugs in practice, we would expect them to be rather easy to fix, and of rare occurence, as the kernels produced contain a strict subset of functionality. In some ways, our approach could therefore even help improve Linux by uncovering bugs that are hard to detect.

This also emphasizes the importance of the analysis phase, which must be sufficiently long to observe all necessary functionality. In case of a crash or similar failure, however, we could only attribute this to a bug in either the kernel or the application implementation that needs to be fixed. In other words, this approach is safe by design.

4.4 Related Work

This chapter is related to previous research from many areas: improving OS kernel reliability and security, reducing the attack surface of the kernel towards user-space applications and specializing kernels for embedded systems.

Kernel specialization. Several researchers have suggested approaches to tailor the configuration of the Linux kernel, although security is usually not a goal. Instead, most often improvements in code size or execution speed are targeted. For instance, Lee et al. [Lee+04] manually modify the source code (e.g., by removing unnecessary system calls) based on a static analysis of the applications and the kernel. Chanet et al. [Cha+05], in contrast, propose a method based on link-time binary rewriting, and also employ static analysis techniques to infer and specialize the set of system calls to be used. Both approaches, however, do not leverage any of the built-in configurability of Linux to reduce unneeded code. Moreover, our approach is completely automated and it is significantly safer, because we do not make any unsupported changes to the kernel.

Micro-kernel architectures and retrofitting security. TCB size reduction has always been a major design goal for micro-kernels [Acc+86; Lie95], and in turn facilitates a formal verification of the kernel [Kle+09] or its implementation in safer languages, such as OCaml [Mad+10]. Our work achieves this goal with a widely-used monolithic kernel, i.e., Linux, without the need of new languages or concepts.

A number of approaches exist that retrofit micro-kernel–like features into monolithic OS kernels, mostly targeting fault isolation of kernel extensions such as device drivers [Cas+09; Mao+11; Swi+02]. For instance, the work of Swift et al. [Swi+02] wraps calls from device drivers to the core Linux kernel API (and vice-versa), as well as use virtual memory protection mechanisms, which leads to a more reliable kernel in the presence of faulty drivers. In the presence of a malicious attacker who can compromise such devices, however, this is in general insufficient. This can be mitigated with more involved approaches such as LXFI [Mao+11], which requires interfaces between the kernel and extensions to be annotated manually. An alternative is to prevent potential vulnerabilities in the source code from being exploitable in the first place. For instance, SVA [Cri+07] compiles the existing kernel sources into a safe instruction set architecture, which is translated to native instructions by the SVA VM. This provides among other guarantees, a variant of type safety and control flow integrity. However, it is very difficult to recover from attacks (or false positives) without crashing the kernel with such defenses [LAK09]. In contrast, kernel tailoring only uses the built-in configurability of Linux, hence kernel crashes can only be due to defects already present in the kernel.

Kernel attack surface reduction. The ISOLSEC model used in this work is commonly used when building *sandboxes* or *isolation* solutions, in which a set of processes

must be contained within a particular security domain (e.g., with [Coo10; HHT04; MMC06], which are all based on the Linux Security Module (LSM) framework [Wri+02]). As we have demonstrated, adjusting the kernel configuration also significantly reduces the attack surface in such a model (this corresponds to the ISOLSEC model). The idea of directly restricting or monitoring for intrusion detection the system call interface on a per-process basis has been extensively explored (e.g., [Ko+00; Pro03] and references in [FHS08]), although not often with specific focus on reducing the kernel's attack surface (i.e., reducing $AS1_{SLOC}$ in the ISOLSEC model), or in other words, to specifically prevent vulnerabilities in the kernel from being exploited by reducing the amount of code reachable by an attacker in this model

SECCOMP [Goo09] directly tackles this issue by allowing processes to be sandboxed at the system call interface. KTRIM [KSK11] goes beyond simply limiting the system call interface, and explores the possibility of finer-granularity kernel attack surface reduction by restricting individual functions (or sets of functions) inside the kernel. In contrast, this work focuses on compile-time removal of functionality from the kernel at a system-wide level instead of a runtime removal at a per-application level. In future work, we will investigate how dynamic approaches such as SECCOMP or KTRIM can be combined with the static tailoring of the kernel configuration most effectively.

4.5 Summary

Linux distributions ship "generic" kernels, which contain a considerable amount of functionality that is provided just in case. For instance, a defect in an unnecessarily provided driver may be sufficient for attackers to take advantage of. The genericalness of distribution kernels, however, is unnecessary for concrete use cases. This chapter presents an approach to optimize the configuration of the Linux kernel. The result is a hardened system that is tailored to a given use case in an automated manner. We evaluate the security benefits by measuring and comparing the attack surface of the kernels that are obtained. The notion of attack surface is formally defined and evaluated in a very generic security model, as well as a security model taking precisely into account the threats posed by a local unprivileged attacker.

We apply the prototype implementation of the approach in two scenarios, a Linux, Apache, MySQL and PHP (LAMP) stack and a graphical workstation that serves data

via network file system (NFS). The resulting configuration leads to a Linux kernel in which unnecessary functionality is removed at compile-time and thus, inaccessible to attackers. We evaluate this reduction using a number of different metrics, including SLOC, the cyclomatic complexity and previously reported vulnerability reports, resulting in a reduction of the attack surface between about 50% and 85%. Our evaluations also indicate that this approach reduces the attack surface of the kernel against local attackers significantly more than previous work on kernel extension isolation for Linux. We are convinced that the presented approach improves the overall system security and is practical for most use cases because of its applicability, effectiveness, ease and safety of use.

		Baseline		Tailored		Reduction	
		LAMP	NFS	LAMP	NFS	LAMP	NFS
Kernel (vmlinux) size in Bytes		9,933,860		4,228,235	4,792,508	56%	52%
LKM total size in Bytes		62,987,539		2,139,642	2,648,034	97%	96%
Options set to 'y'		1,537		452	492	71%	68%
Options set to 'm'		3,142		43	63	99%	98%
Compiled source files		8,670		1,121	1,423	87%	84%
GENSEC	Call graph nodes	230,916		34,880	47,130	85%	80%
	Call graph arcs	1,033,113		132,030	178,523	87%	83%
	$AS1_{SLOC}$	6,080,858		895,513	1,122,545	85%	82%
	$AS1_{cycl}$	1,268,551		209,002	260,189	84%	79%
	$AS1_{CVE}$	848		338	429	60%	49%
	$AS2_{SLOC}$	58,353,938,861		11,067,605,244	11,578,373,245	81%	80%
	$AS2_{cycl}$	2,721,526,295		1,005,337,180	1,036,833,959	63%	62%
	$AS2_{CVE}$	20,023		7,697	9,512	62%	52%
ISOLSEC	Call graph nodes	92,244	96,064	15,575	21,561	83%	78%
	Call graph arcs	443,296	462,433	64,517	89,175	85%	81%
	$AS1_{SLOC}$	2,403,022	2,465,202	425,361	550,669	82%	78%
	$AS1_{cycl}$	504,019	518,823	99,674	126,710	80%	76%
	$AS1_{CVE}$	485	524	203	276	57%	47%
	$AS2_{SLOC}$	15,753,006,783	15,883,981,161	4,457,696,135	4,770,441,587	72%	70%
	$AS2_{cycl}$	918,429,105	929,197,559	374,455,910	391,855,241	59%	57%
	$AS2_{CVE}$	10,151	11,127	4,287	5,489	57%	51%

Table 4.1. Summary of kernel tailoring and attack surface measurements.

	Necessary configuration option	LAMP	Workstation/NFS
CVE-2013-2094 (Perf.)	`CONFIG_PERF_EVENTS`	–	–
CVE-2012-0056 (Mem.)	`CONFIG_PROC_FS`	–	–
CVE-2010-4158 (BPF)	`CONFIG_NET`	–	–
CVE-2010-3904 (RDS)	`CONFIG_RDS`	✓	✓

Table 4.2. Prevention of some past kernel vulnerabilities through tailoring. Legend: ✓: Compiled out in the tailored kernel, not vulnerable, –: Remained compiled in.

	Ideal LKM isolation	Kernel Tailoring		Both combined	
		LAMP	Workstation/NFS	LAMP	Workstation/NFS
$AS1_{SLOC}$	2,064,526	425,361	550,669	420,373	489,732
$AS1_{cycl}$	444,775	99,674	126,710	98,534	113,735
$AS1_{CVE}$	390	203	276	203	240
$AS2_{SLOC}$	11,826,476,219	4,457,696,135	4,770,441,587	4,452,329,879	4,663,745,009
$AS2_{cycl}$	851,676,457	374,455,910	391,855,241	374,214,950	386,472,434
$AS2_{CVE}$	7,725	4,287	5,489	4,287	4,849

Table 4.3. Comparison of IsolSec attack surface measurements between an ideal LKM isolation approach (a lower bound of the attack surface of kernel extension fault isolation approaches) and our approach, when applied to the current Ubuntu 12.04 Kernel. The third column represents attack surface measurements that would result if both approaches were combined.

This `NULL` *page intentionally mapped.*

5 Run-time Kernel Trimming

> "It is nice to know the dictionary definition for the adjective "elegant" in the meaning "simple and surprisingly effective""
>
> E.W. Dijkstra, *On the nature of Computing Science*

My second approach to reduce the kernel attack surface operates at run-time.

It hinges on the fact that each application makes use of distinct kernel functionality, hence one can *scope* the use of kernel functionality per-application. To do so, I implement kernel trimming (or KTRIM), a proof-of-concept tool that reduces the per-application attack surface by instrumenting the kernel and preventing access to a set of functions, with only small performance penalties. Because this approach simply requires loading a kernel module and does not require recompilation or binary rewriting, the approach is easy to deploy in practice. The limitations of KTRIM are that of any learning-based approach: false positives, whereby a kernel function has been incorrectly learned as unnecessary, can happen. To understand the feasibility of the approach, I deploy KTRIM on a server used for real-world workloads for more than a year, and observe no false positives during a full year.

The approach is structured in four phases designed to meet the challenges of deploying a low overhead and low false-positive run-time attack surface reduction tool. Performance overhead is kept low by avoiding to instrument frequently-called kernel functions, and false positives can be reduced by grouping functions that are likely to be called under

similar conditions, at the cost of lower attack surface reduction.

Unlike methods such as anomalous system call monitoring [For+96; HFS98; Kru+03; Mut+06; WD01] or system call sandboxing [AR00; Dan+; Gol+96; Goo09; Pro03], KTRIM instruments at the level of individual kernel functions (and not merely the system call interface). This makes the approach *quantifiable*, and *non-bypassable*.

Naturally, I quantify security benefits by using the attack surface measurement framework described in Chapter 3. The attack surface can essentially be computed by defining entry points for the attacker (system calls) and performing reachability analysis over the kernel call graph. Because KTRIM intercepts calls to individual kernel functions, it is particularly well-suited for measurements by this framework. In turn, this quantification enables objective comparison of security trade-offs between KTRIM variations.

The non-bypassable property is achieved by applying the *complete mediation* principle: I reckon that, in the context of attack surface reduction, kernel functions can be considered as resources to which access must be authorized. A reliable way to retrofit such an authorization mechanism is to place authorization hooks as close to the resource as possible, which I achieve by instrumenting the entry of most kernel functions. This contrasts with existing system-call interposition techniques which can only reduce kernel attack surface at the coarse granularity of the system call interface. Therefore, they cannot provide reliable metrics on the amount of kernel code removed from the attack surface.

My evaluation results show that by varying the nature of the analysis phase, it is possible to provide a trade-off between attack surface reduction and the minimal time span of the learning phase. For instance, it is possible to improve attack surface reduction from 30% to 80% (when compared to the attack surface of the kernel with respect to an unprivileged attacker controlling a local process in the absence of KTRIM), by making the learning phase twice as long.

	Functions	Ratio
Baseline RHEL 6.1 kernel	31,429	1
Lowest potential attack surface at run-time (`qemu-kvm`)	5,719	1:6
Lowest potential attack surface at run-time (`mysqld`)	3,663	1:9

Table 5.1. Comparison between the number of functions in the STATICSEC attack surface for two kernels and the number of kernel functions traced for `qemu-kvm` and `mysqld`.

5.1 Overview

5.1.1 Motivations and challenges for run-time attack surface reduction

The results for compile-time attack surface reduction in Chapter 4 are very enticing. In particular, the results show that the kernel attack surface can be reduced by 80 to 85% (when measured with AS_{SLOC}). However, we can make three observations that show the added benefits of a run-time approach.

Improved compatibility and flexibility. The first observation is straightforward: compile-time attack surface reduction requires recompiling the kernel, which can be problematic for some practical deployments where the use of a standard distribution kernel is mandated (e.g., as part of a support contract with the distributor). By providing attack surface reduction as a kernel module, this requirement can be met. Additionally, this provides greater flexibility because it becomes possible to easily enable and disable attack surface reduction without rebooting.

Finer scope-granularity. Attack surface reduction at compile time results in system-wide attack surface reduction. A run-time approach can have finer scope, e.g., by reducing the attack surface for a group of processes, or by having different policies for each group of processes.

Higher attack surface reduction potential. Because of this finer per-process granularity, run-time attack surface reduction could achieve higher attack surface reduction. To evaluate the validity of this assertion, we devise the following experiment. On two machines which serve as development servers, we collect, during 8 months on one machine and a year and a month on a second machine, kernel traces corresponding to the use of various daemons and UNIX utilities. We observe that the highest number of unique

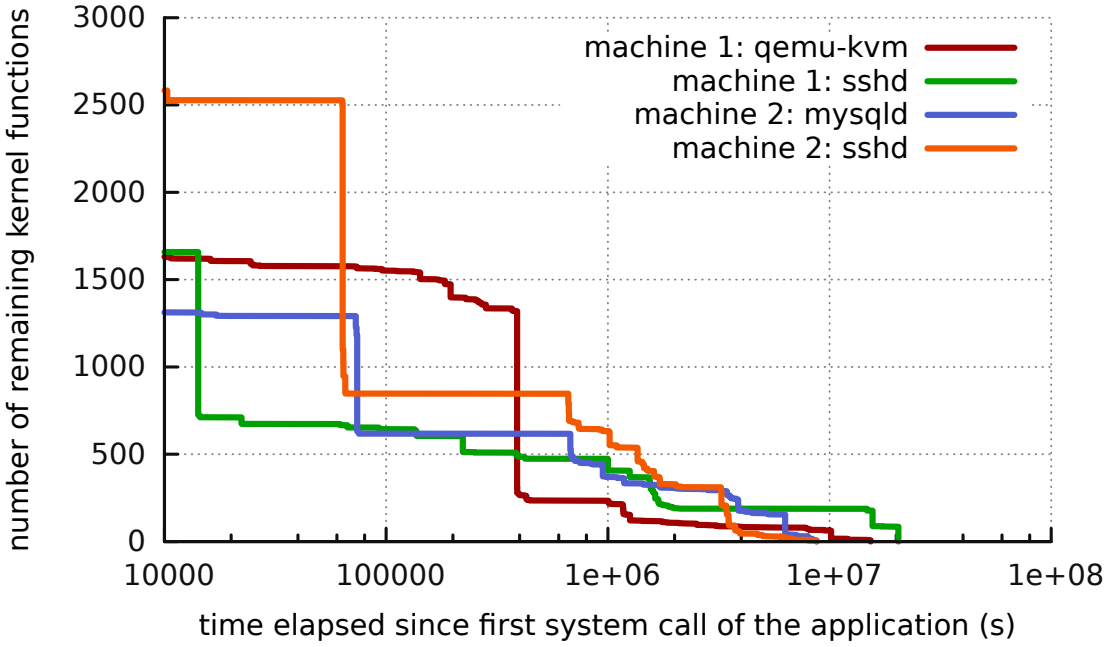


Figure 5.1. Evolution of the number of unique kernel functions used by applications: after several months of use, no new kernel functions were triggered.

kernel functions are used by the `qemu-kvm` process, which is running in one node serving as KVM hypervisor on the test bed. The lowest number is achieved by the MySQL daemon. Table 5.1 compares these results and shows that, potentially, restricting the kernel attack surface at run-time can result in an attack surface that is about 5 to 10 times lower than that of a distribution kernel.

Rate of convergence and the challenge of false positives. In a preliminary experiment, no synthetic workloads were run on the machines. Instead, the machines were traced during their *real-world* usage. Over time, because the workload on a system can change, new kernel functions can be used by an application. In Figure 5.1, we fix the total number of kernel functions used by a given program, and plot the number of unique functions that remain after the first system call performed. The figure shows that it takes significant time to converge to the final set of functions used by the program. For example, the `MySQL` daemon took 103 days to converge to its final set of kernel functions (out of a total tracing duration of 403 days). Hence, an important challenge in building an attack surface reduction is to design an approach that will result in fast convergence even in the presence of incomplete traces. This can also be formulated as reducing the false

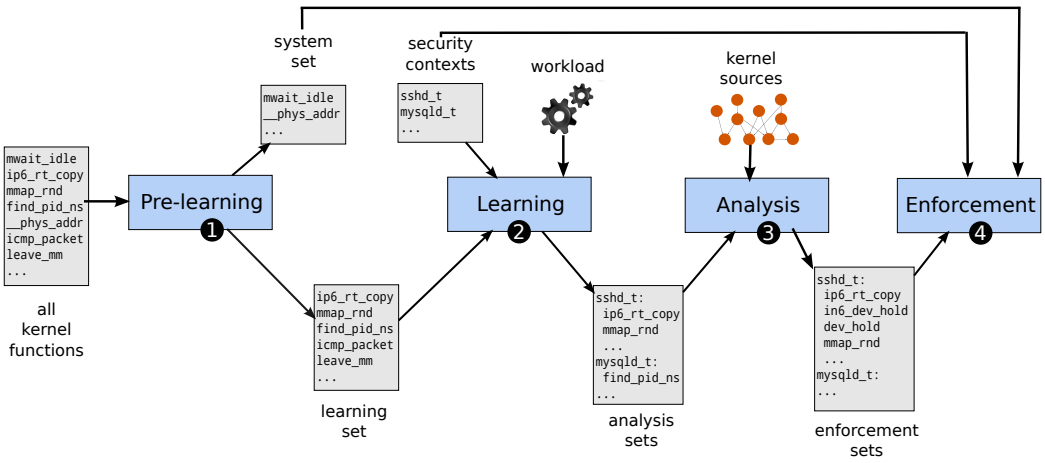


Figure 5.2. KTRIM run-time kernel attack surface reduction phases.

positives of the detection system. The approach we take here is to *group* kernel functions together (e.g., all functions declared in a given source file) to reduce the likelihood of false positives.

5.2 Design

I now detail my design and implementation of KTRIM, a tool that aims to achieves the benefits of run-time attack surface reduction, while trying to meet its challenges, in particular the reduction of false-positives. The four major phases for run-time attack surface reduction are depicted in Figure 5.2 and detailed below.

5.2.1 Pre-learning phase

The goal of phase ❶ is to prepare an enforcement phase (and incidentally, learning phase) with low performance overhead. At first, KTRIM sets up tracing for all kernel functions that can be traced. In other words, each kernel function is instrumented and each call to a kernel function is logged. In the case of Linux, this is achieved by using the FTRACE tool and the kernel's `debugfs` interface. Since some kernel functions are called thousands of times per second, this results in significant performance overhead at first, and also fills up the log collection buffer very quickly, which leads to missed traces. In order to cope with

this practical limitation, I select, each time the trace buffer fills up, functions which are called beyond a given threshold and disable tracing for those functions. These functions form the *system set*, while the remaining kernel functions form the *learning set*.

My experiments show this heuristic is useful for keeping a low performance overhead in the enforcement phase: instrumenting every single kernel function would cause significant overhead. For instance, functions related to memory management (`kfree`, `__page_cache_alloc`, `get_page`), or synchronization (`_spin_lock`, `mutex_lock`) always find their place in the system set with this heuristic: they are called very often and instrumenting them would be both detrimental for performance and would not significantly reduce kernel attack surface (since most applications would end up using them anyway). Listing 5.1 shows a more subtle example of a function included in the system set: `ext4_claim_free_blocks` is repeatedly called in a loop, and this resulted in the function being included in the system set, whereas its caller, `ext4_mb_new_blocks`, was not.

```
ext4_fsblk_t ext4_mb_new_blocks(...)
{
  ...
    while (ar->len && ext4_claim_free_blocks(sbi, ar->len)) {
      /* let others to free the space */
      yield();
      ar->len = ar->len >> 1;
    }
  ...
}
```

Listing 5.1. Excerpt of an `ext4` kernel function for allocating new blocks for the filesystem. The function called repeatedly in the while loop was included in the system set by the pre-learning phase.

5.2.2 Learning phase

In phase ❷, a workload is run and traces are collected to learn which kernel functions are necessary for the operation of a target program, for this specific workload as well as the system configuration and hardware specific to this machine — as different configuration and hardware will result in different kernel functions being exercised. For example, the

filesystem used to store the files of an application will result in different kernel functions being called at each I/O operation.

For each target program for which the kernel attack surface should be reduced (e.g., `sshd` and `mysqld` in Figure 5.2) a *security context* is specified. The security context is used to identify processes during the learning phase and the enforcement phase, in the same manner security contexts are used to specify subjects in access control frameworks such as SELinux. For this reason, in the current implementation of KTRIM, I thus make use of SELinux [SVS01] security contexts as security context (in Figure 5.2, this is represented by the `sshd_t` and `mysqld_t` SELinux types). Then, each function trace collected is associated with this security context, resulting in one *analysis set* per security context.

I have implemented this step in two different ways: first, I implemented as a kernel module using the KPROBES dynamic instrumentation framework. In this case, a probe is specified for each kernel function in the learning set, and the structure specifying the probe contains a bit-field which tracks the security contexts which have made use of the corresponding function (associating also a time-stamp to that access, for the purposes of creating statistics for this chapter). However, as some system administrators have been wary of installing a kernel module, I have also created a user-space tool based on FTRACE, which logs and tracks all kernel functions in the learning set. The functionality that is provided with both approaches is equivalent, although the KPROBES based approach is more efficient.

5.2.3 Analysis phase

In phase ❸, I expand each analysis set to reduce false positives during enforcement. Indeed, some kernel functions can be rarely exercised at run-time, such as fault handling routines, and a learning phase that would not be exhaustive enough would not catch such functions.

I evaluate three methods to achieve this goal, in addition to keeping the analysis set unchanged (no grouping). The first, *file grouping*, performs expansion by grouping functions according to the source file the function is defined in. The second, *directory grouping*, performs expansion by grouping functions according to their source directory.

Finally, I perform *cluster grouping*, by performing k-means clustering of the kernel

call graph. Although other unsupervised machine-learning algorithms (such as hierarchical clustering) could be used, I chose k-means because of its well known scalability (due to the size of the kernel call graph). In particular, I make use of the very scalable *mini-batch* k-means algorithm described in [Scu10]. In my experiments, clustering individual functions led to unevenly-sized clusters and unsatisfactory evaluation results. Therefore, I opted for using file grouping: each node in the call graph became a file, and a file calls another target file if and only if there exists a function inside that file calling a function in the target file. I also converted the graph to undirected, and used the adjacency matrix thus obtained for clustering. The various parameters necessary for the clustering algorithm were tweaked iteratively, best results were obtained by using $k = 1000$, $b = 2000$, $t = 60$ with the notations of [Scu10].

Of course, one is not limited to these three choice, and other algorithms can be designed for this phase. However these algorithms, especially cluster grouping, performed reasonably well and I did not feel the need to explore further.

In effect, this phase increases the coarseness of the learning phase, trading off attack surface reduction for a lower false acceptance rate and faster convergence.

5.2.4 Enforcement phase

Finally, I enforce in phase ❹ that each process (defined by its security context) makes calls within the set of functions that are not in the corresponding *enforcement set*. To achieve this goal, I monitor calls to each kernel function that is not in the system set, and verify that the call is permitted for the current security context. In the implementation, I make use of the Linux kernel's KPROBES feature to insert probes at the very beginning of each of those functions.

Currently, two options exist for the enforcement phase: the first is to simply log the violation, and the second one is a fail-stop behavior, triggering a kernel oops (which will attempt to kill the current process, failing that the kernel will crash). This enforcement option can be chosen separately for each security context (i.e., for security contexts where one is certain that the learning workload is thoroughly completed, enforcement can be set to fail-stop mode, while other security contexts can be left in detection-only mode.

Doing so in a manner that maintains the non-bypassability is a significant implementation challenge. Indeed, when we intercept calls deep within the kernel, there are kernel

invariants that must be honored. In particular, the kernel can already be holding locks, and one cannot perform any operation that would attempt taking the same lock, as that would result in a deadlock.

5.3 Evaluation

5.3.1 Evaluation use case

To measure the security benefits, in terms of attack surface reduction as well as false positives, and performance, I opt for targeting daemon processes on a server during its use for professional software development and testing, for a period of 403 days. The server is an IBM x3650, with a quad-core Intel Xeon E5440 CPU and 20 GB RAM, running the Red Hat Enterprise Linux Server release 6.1 Linux distribution (Linux kernel version 2.6.32-131). The daemons I target on the server are OPENSSH (version 5.3p1), MYSQL (version 5.1.52) and NTP (version 4.2.4p8). The same server also hosts KVM virtual machines, and I trace `qemu-kvm` which is the user-space process running drivers on the host for virtualizing hardware to the guest virtual machines.

5.3.2 Attack surface reduction

I compute attack surface reduction by using the enforcement set for each application, combined with the system set, as barrier functions when performing reachability analysis over the call graph. The kernel call graph is generated using the NCC and FRAMA-C tools. In particular, source lines of code (SLOC) and cyclomatic complexity metrics are calculated on a per-function basis by FRAMA-C. This approach follows the quantification explained in Chapter 3.

Table 5.2 summarizes attack surface reduction results for all services, grouping algorithms, and attack surface metrics in my setup. Attack surface reduction can vary roughly between 30% and 80%, depending mostly on the grouping algorithm. Within a grouping algorithm, results are consistent (e.g., about 75% without grouping compared to about 40% with cluster grouping) across different metrics and services.

5.3.3 False positives

In this setup, we observe the usage of a daemon in its real-world usage. As a consequence, it is possible that some previously unused feature of the daemon is finally used after several months of usage. To measure how well different grouping algorithms fare in that regard, I opt to use the first 20% of the collected traces as a learning phase, and the remaining 80% as an enforcement phase[1]. Any function that is called during the enforcement phase but is not in the enforcement set (or system set) is then accounted as a false-positive. The results in terms of number of (unique) functions causing false positives, are shown in Table 5.3, together with the convergence rate. I observe that, when grouping by directory or by clustering, this time frame for the learning phase is largely sufficient in all cases. For the two other grouping techniques, only `qemu-kvm` converges prior to the 20% time-frame for all grouping techniques.

5.3.4 Performance

I measure performance during the enforcement phase with the LMBENCH 3 benchmarking suite. I perform 5 runs and collect the average latency, which is reported in Table 5.4. Most overheads are very low (especially considering this is a micro-benchmark): the pre-learning phase is effective in segregating performance-sensitive kernel functions. However, some operations (e.g., empty file creation) can incur significant overhead (75%), which shows that my heuristic approach still has room for improvement — although file creation is not a performance-critical operation in most workloads.

As a macro-benchmark, I use the `mysqlslap` load-generation and benchmarking tool for MYSQL. I run a workload of 5000 SQL queries (composed of 55% INSERT and 45% SELECT queries, including table creation and dropping time), and measure the average duration over 30 runs. This workload is run 50 times, resulting in 50 averages, which I compute a 95%-confidence interval over. Results in Table 5.5 show that KTRIM incurs no measurable overhead. In addition, the results confirm the pre-learning phase's effectiveness: without this phase, KTRIM would incur more than 100% overhead on this test.

[1]This setting is solely used for the estimation of false-positives. The attack surface reduction numbers make use of the entire trace dataset as a learning phase (to provide the most accurate results).

5.3.5 Detection of past vulnerabilities

I now focus on the case of four vulnerabilities for the Linux kernel for which a public kernel exploit was available (I use the same vulnerabilities described in Chapter 2). For each, I provide a short reminder of the vulnerability, and pinpoint the individual kernel function responsible for the vulnerability.

KTRIM would have detected exploits targeting such vulnerabilities in many cases (see Table 5.6). This means, for example, if a remote attacker had taken control of `mysqld` through a remote exploit, or if a virtual-machine-guest exploited a `qemu-kvm` vulnerability such as CVE-2011-1751 (virtunoid exploit) on the machine, and then attempted to elevate his privileges on the host using an exploit for the kernel, KTRIM would detect the exploit. In particular, note that it does not matter how the exploit is written: this detection is non-bypassable for the attacker because the access to the function containing the vulnerability is detected by KTRIM in the enforcement phase, and, by definition, it's not possible to write an exploit for a vulnerability without triggering the vulnerability.

Finally, the AS_{CVE} metric results (in Table 5.2) provide the best numbers for estimating KTRIM's effectiveness in detecting exploits for past CVEs. The following examples are anecdotal and for illustrative purposes.

`perf_swevent_init` **(CVE-2013-2094).** This vulnerability concerns the Linux kernel's recently introduced low-level performance monitoring framework. It was discovered using the TRINITY fuzzer, and, shortly after its discovery, a kernel exploit presumably dated from 2010 was publicly released, suggesting that the vulnerability had been exploited in the wild for the past few years. The vulnerability is an out-of-bounds access (decrement by one) into an array, with a partially-attacker-controlled index. Indeed, the index variable, `event_id` is declared as a 64 bit integer in the kernel structure, but the `perf_swevent_init` function assumes it is of type `int` when checking for its validity: therefore the attacker controls the upper 32 bits of the index freely. In the publicly released exploit, the `sw_perf_event_destroy` kernel function is then leveraged to provoke the arbitrary write, because it makes use of `event_id` as a 64-bit index into the array. This results in arbitrary kernel-mode code execution.

`check_mem_permission` **(CVE-2012-0056).** A detailed description of this cleverly discovered vulnerability is provided by its author online[2]. It essentially consists in

[2] `http://blog.zx2c4.com/749`

tricking a set-user-id process into writing to it's own memory (through `/proc/self/mem`) attacker-controlled data, resulting in obtaining root access. The vulnerability is in the kernel function responsible for handling permission checks on `/proc/self/mem` writes: `__check_mem_permission`. Although KTRIM does not intercept this function directly, it intercepts the `check_mem_permission` function which is the unique caller of `__check_mem_permission` (in fact, this function is inlined by the compiler, which explains why KTRIM does not instrument it).

`sk_run_filter` **(CVE-2010-4158).** This vulnerability is in the Berkeley Packet Filter (BPF) [MJ93] system used to filter network packets directly in the kernel. It is a "classic" stack-based information leak vulnerability: a carefully crafted input allows an attacker to read uninitialized stack memory. Such vulnerabilities can potentially breach confidentiality of important kernel data, or be used in combination with other exploits, especially when kernel hardening features are in use (such as kernel base address randomization). In my evaluation, KTRIM detects exploits targeting this vulnerability when no grouping is used, and under `sshd` or `mysqld`.

`rds_page_copy_user` **(CVE-2010-3904).** This vulnerability is in reliable datagram sockets (RDS), a seldom used network protocol. The vulnerability is straightforward: the developer has essentially made use of the `__copy_to_user` function instead of the `copy_to_user` function which checks that the destination address is not within kernel address space. This results in arbitrary writes (and reads) into kernel memory, and therefore kernel-mode code execution. This vulnerability is in an loadable kernel module (LKM) which is not in use on the target system, yet, because of the Linux kernel's on-demand LKM loading feature which will load some kernel modules when they are made use of by user-space applications, the vulnerability was exploitable on many Linux systems.

This vulnerability is detected by KTRIM, even after grouping. However, unlike the three previous exploits, this vulnerability would also have been prevent by approaches such as kernel extension isolation, or even more simply, the use of the Linux `modules_disabled` switch previously explained. Because of this, as explained in the STATICSEC model, this CVE (and many similar ones in other modules) is not counted in the AS_{CVE} metric.

5.3.6 Synthetic LAMP workload

In addition to the RHEL-based use case, to show the applicability of KTRIM across different linux kernel versions, distributions, and applications, I run a synthetic workload on an Ubuntu 12.04 server (kernel version 3.2.0-25) with a typical APACHE2, MYSQL and PHP5 (LAMP) installation in the presence of KTRIM in learning mode. I set up the web server for serving static content, in addition to the collaboration platform DOKUWIKI [Goh] and the message board system PHPBB3 [Php] in their default configuration.

The test workload is generated by the SKIPFISH [ZHR] web application security scanner, which automatically exercises all web pages hosted on the server, including edge-case inputs to attempt triggering vulnerabilities (the SKIPFISH run lasted about 10 hours).

Results in Table 5.7 show similar results to the RHEL-based use-case, thereby confirming the broader applicability of KTRIM.

5.4 Discussion

In this section, I discuss the results of kernel attack surface reduction as well as its issues.

5.4.1 Security contexts

KTRIM currently makes use of SELinux security contexts. Other possibilities for security contexts would include process owner `UID` (which is suitable for daemons), or the security contexts of other access control frameworks (e.g., AppArmor or TOMOYO). An important consideration for access control systems are the security context transitions that can occur. For traditional UNIX `UID`s, this typically corresponds to `suid` executables, which will run with the `UID` of their owner, effectively transitioning `UID`s. SELinux makes use of type transitions to achieve a similar effect, though they do not need to be used for elevating privileges alone, but are used more generally for switching privileges. This can be problematic for kernel attack surface reduction: if an attacker is allowed to change privileges and maintain the possibility of arbitrary code execution, she can mount attacks to the kernel beyond the restriction of the current security context. However, in cases

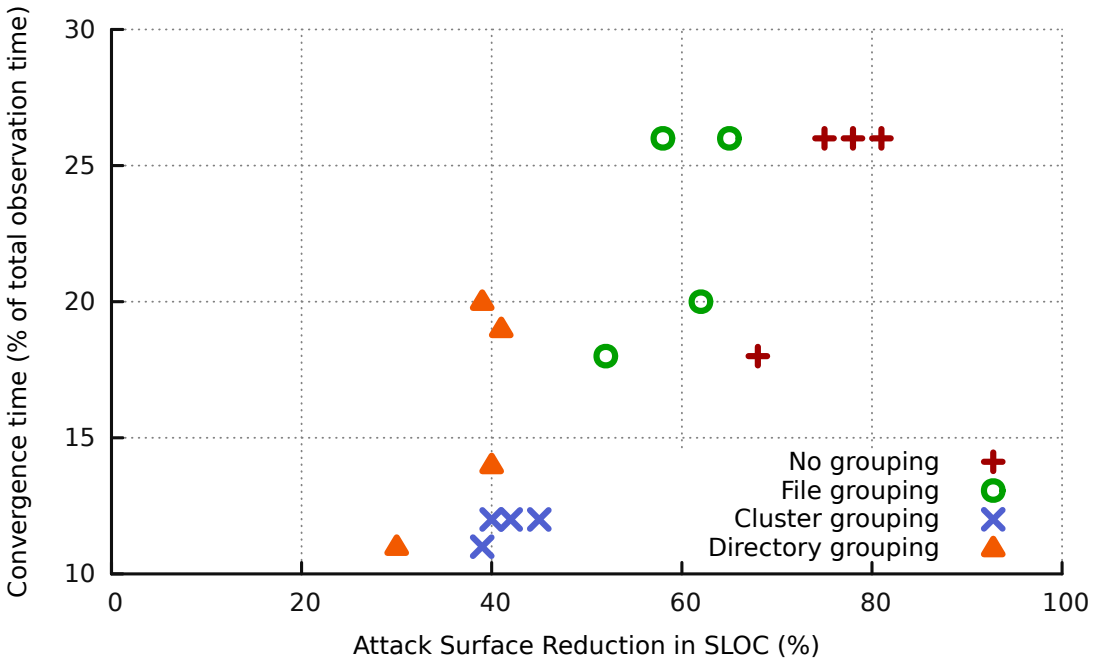


Figure 5.3. Attack surface reduction and convergence rate for the evaluated applications, under different grouping methods (RHEL use case only).

where sandboxing is used, processes can often be prevented from executing other binaries with security transitions.

5.4.2 Analysis phase: grouping algorithms and trade-offs

Figure 5.3 depicts convergence and attack surface trade-offs for all four grouping methods explored in this work. The closer a data point is to the bottom right corner of this graph, the better the trade-off. For instance, I observe that cluster grouping subsumes directory grouping: it achieves a better convergence rate at a slightly better attack surface reduction. Similarly, no grouping performs better than file grouping for 3 out of the 4 services evaluated (the exception being `ntpd`).

In practical deployments of KTRIM, these trade-offs should be adapted to the workload and target service: for instance, in use-cases where the workload is not well defined (e.g., development platforms), clustering grouping is a more attractive solution: it converges about twice as fast as the other algorithms.

5.4.3 False positives

In my evaluation of false-positives, I decided to reserve the last 80% of the traces for the enforcement phase. This corresponds roughly to a period of almost 3 months for the learning phase, which, although lengthy in some cases, is reasonable for services which are put into testing for several weeks before being put into production. In addition, the server I use in this evaluation is a development machine, whose use can change significantly over time, when compared to a typical production server. With that in mind, the results are very positive: for all grouping methods and services, no false positives were observed for about a full year.

5.4.4 Performance trade-offs

The pre-learning phase contains a tunable parameter that sets the threshold for disable tracing of performance-sensitive functions. Because the results showed low performance overhead, I did not tweak this parameter in the evaluations. However, I expect that increasing the threshold (i.e., reducing the size of the system set) will decrease performance, but improve attack surface reduction (because each application's traces are cluttered by the system set). Potentially, the convergence rate can also be improved when grouping is used (because the system set functions are not fed into the grouping algorithms: after grouping, the functions present there could unnecessarily increase the kernel attack surface).

5.4.5 Attack surface metrics

Attack surface reduction results remain consistent when comparing AS_{SLOC}, AS_{cycl}, AS_{CVE} and even the number of functions in the StaticSec attack surface. This is particularly remarkable, as SLOC and cyclomatic complexity are a priori metrics (i.e., they estimate future vulnerabilities by source code complexity) whereas CVE numbers are a posteriori metrics (i.e., this reflects the number of functions that have been found to be vulnerable by the past), and only a weak correlation between such metrics has been found in prior work [SW08].

5.4.6 Attack surface size and TCB

In absolute terms, results show that the kernel attack surface can be as low as 105K SLOC (without grouping) and 313K SLOC (with cluster grouping). This means that using the statistic that the Linux kernel has 10 million SLOC, overestimates the amount of code an attacker (in the STATICSEC model) can exploit defects in, by two orders of magnitude. While this number is still greater than the size of state-of-the-art reduced-TCB security solutions such as the MINIX 3 microkernel (4K SLOC [Her+06b]), the Fiasco microkernel (15K SLOC [Har+05]) or Flicker (250 SLOC [McC+08]), it is comparable to the TCB size of commodity hypervisors such as Xen (98K SLOC without considering the Dom0 kernel and drivers, which are often much larger [MMH08]).

Hence, one could be tempted to challenge the conventional wisdom that commodity hypervisors provide much better security isolation than commodity kernels. However, making such a statement would require comparable attack surface measurements to be performed on a hypervisor, after transposing the STATICSEC model.

5.4.7 Limits of existing instrumentation

In Section 5.1.1, I state that the attack surface for `mysqld` contains at least 3,663 functions. This number is obtained by adding up the system set size and the enforcement set size for `mysqld`. However, this lower-bound is difficult to reach in practice due to practical limitations of the tracing (FTRACE) and instrumentation (KPROBES) framework I use. Indeed, only a subset of all kernel functions can be instrumented in practice (about 20K functions out of 30K total, depending on the kernel version and loaded modules). In particular, functions that are susceptible to be used by the instrumentation framework itself, functions that are declared in assembly, and functions inlined by GCC cannot be instrumented, although they can be found by static analysis (and, more importantly, they can be target of attacks). Hence, the 3,663 functions registered is lower than the number of kernel functions that were really used by the application. In addition, I also take into account in the attack surface measurements that I can only enforce on functions which KPROBES can instrument, which results in further increase of the attack surface measurement. In the end, I obtain 7,498 functions (see Table 5.2) for the `mysqld` attack surface.

In other words, although current results already provide high attack surface reduction

(reducing kernel attack surface 5-fold in some cases), improving the instrumentation used in KTRIM (e.g., by instrumenting at the basic-block level) could result in even better results by allowing more code to be enforced on.

5.4.8 Improving the enforcement phase

Currently, the enforcement phase can only prevent code execution by a fail-stop behavior: the Linux kernel is written in the C language, hence with no exception handling mechanism in case the execution flow is to be aborted at an arbitrary function. As an example, the current execution could have taken an important kernel lock, and aborting the execution of the current flow abruptly would result in a kernel lock-up. This fail-stop behavior is a common problem to many kernel hardening mechanisms (e.g., see references in [LAK09]), and it would be possible to expand KTRIM with existing solutions. For example, Akeso [LAK09] allows rolling back to the start of a system call, from (most) kernel functions. This is essentially achieved by establishing a snapshot of shared kernel state at each system call.

5.5 Related Work

Two approaches can be envisioned to reduce the attack surface of the kernel: either making the kernel smaller (or switching to smaller kernels, which is often not an option in practice), or putting in place run-time mechanisms that restrict the amount of code accessible in the running kernel.

This works focuses on the run-time mechanisms: although there has been extensive work in providing better sandboxing and access control for commodity operating systems, little has been done to reduce kernel attack surface and quantify improvments. Most approaches that may reduce kernel attack surface have used the system call interface (or other existing hooks in the kernel, such as LSM hooks for Linux). In particular, no quantification of run-time kernel attack surface reduction has been done so far for these techniques.

5.5.1 Smaller kernels

The following summarizes related work on reducing the kernel attack surface at compile-time and, more generally, designing and developing smaller kernels.

Micro-kernels. Micro-kernels are designed with the explicit goal of being as small and modular as possible [Acc+86; Lie95]. This design goal led to micro-kernels being a good choice for security-sensitive systems [Her+06b; Hoh+04; Kle+09]. For instance, MINIX 3 [Her+06a; Her+06b; Her+08], is a micro-kernel designed for security: in particular, its kernel is particularly small, at around 4,000 SLOC. A significant practical drawback of all these approaches is the lack of compatibility with the wide variety of existing middleware, applications, and device drivers, which render their adoption difficult, except when used as hypervisors [Har+05; HL10] to host commodity OSes. However, when hypervisors are used, isolation is only provided between the guest operating systems, which might not be sufficient in some use cases. When this isolation is sufficient, it can translate into a significant performance overhead over single-OS implementations with more lightweight solutions such as containers [Kur+11].

Kernel extension fault isolation. There has been a number of approaches to retrofit micro-kernel-like features into commodity OS kernels, by targeting fault isolation of kernel extensions such as device drivers [BWZ10; Cas+09; Mao+11; Swi+02]. For instance, Nooks [Swi+02] can wrap calls between device drivers and the core kernel, and make use of virtual memory protection mechanisms, leading to a more reliable kernel in the presence of faulty drivers. However, in the presence of a malicious attacker who can compromise such devices, this is insufficient, and more involved approaches are required: e.g., LXFI [Mao+11], which requires interfaces between the Linux kernel and extensions to be manually annotated. A notable drawback common to all the techniques is that, by design, they only target kernel modules and not the core kernel.

Kernel specialization. Some researchers have suggested approaches to tailor the Linux kernel to a specific workload, although security is usually not a goal, but improvements in code size or execution speed are: Lee et al. [Lee+04] manually modify the source code (e.g., by removing unnecessary system calls) based on a static analysis of the applications and the kernel. Chanet et al. [Cha+05], in contrast, propose a method based on link-time binary rewriting similar to dead-code analysis, but also employ static analysis techniques to infer and specialize the set of system calls to be used.

5.5.2 System call monitoring and access control

A number of techniques make use of the system call interface or the LSM framework to restrict or detect malicious behavior. I explain their relation with kernel attack surface reduction here.

Anomalous system call monitoring. Various host-based intrusion detection systems detect anomalous behavior by monitoring system calls (e.g., [Fen+03; For+96; GRS04; HFS98; Kru+03; Mut+06; WD01] and references in [FHS08]). Most of these approaches detect normal behavior of an application based on bags, tuples or sequences of system calls, possibly taking into account system call arguments as well [Can+12]. Because behavioral systems do not make assumptions on the types of attacks that can be detected, they target detection of unknown attacks, unlike signature-based intrusion detection systems which can be easily bypassed by new attacks. It has also been shown that it is possible for attackers to bypass such detection mechanisms as well [Kru+05; Ma+12; TMK03; WS02]. Hence, although behavioral intrusion detection could, as a side effect, reduce the kernel attack surface (because a kernel exploit's sequence of system calls might deviate from the normal use of the application), it is bypassable by using one of many known techniques, especially in the context of kernel attack surface reduction. This argument is not applicable in the case where the anomaly detection is performed with a trivial window size of one, i.e., on a system-call basis – however, this corresponds to the essence of system-call-based sandboxing which is explained in the next paragraph.

System-call-based process sandboxing. Sandboxes based on system call interposition [AR00; Dan+; Gol+96; Goo09; KSK11; Pro03] provide the possibility to whitelist permissible operations for selected applications by creating a security policy. Although most of these sandboxes were primarily designed to provide better resource access control, they can also reduce the kernel attack surface, as the policy will restrict the access to some kernel code (e.g., because a system call is prevented altogether). A good example for achieving attack surface reduction with such an approach is provided by seccomp. In its latest instantiation, it allows a process to irrecovably set a system call authorization policy. The policy can also specify allowable arguments to the system call. Hence, this allows skilled developers to manually build sandboxes that reduce the kernel attack surface (e.g., the Chrome browser recently started using such a sandbox on Linux distributions that support it). However, this approach comes with two fundamental drawbacks. The first is that it is very difficult to quantify how much of the kernel's

attack surface has been reduced by analysing one such policy, without the full context of the system its running on. To explain this, I take the simple example of a process that is only allowed to perform reads and writes from a file descriptor which is inherited from (or passed by) another process (this is the smallest reasonable policy that one could use). By merely observing this policy, the attack surface exposed by the kernel to this application could be extremely large, since this file descriptor could be backing a file on any type of filesystem, a socket, or a pipe. More generally, the kernel keeps state that will affect the kernel functions that would handle the exact same system call. The second issue is that many system call arguments cannot be used to make a security decision (and reduce the kernel attack surface): this is a well known problem for system call interposition [Gar03; Wat07]. As a consequence, the attack surface on some policies can be larger than expected. Fundamentally, KTRIM can be seen as a generalization of system-call-based sandboxing because access control is performed at the level of each kernel function instead of limiting itself to the system call handlers only.

Access control. The significant vulnerabilities and drawbacks of system-call-based sandboxing for performing access control have led to mechanisms with tighter integration with the kernel [Wat13]. In particular, on Linux, the LSM framework was created [Wri+02] as a generic way of integrating mandatory access control (MAC) mechanisms, such as [SVS01], into the kernel. Unlike system-call interposition, this approach can be shown to provide complete mediation [JEZ04]. In a way, kernel attack surface reduction can also be seen as a resource access control problem. In this case, the resources to access are no longer files, sockets, IPCs, but the kernel functions themselves – however, in this case, the LSM framework would be of little use as a reference monitor (since only a select number of kernel functions are intercepted). It then becomes clear that the proper way of reducing the kernel attack surface should also be with a non-bypassable system that would perform the access control as close as possible to the protected resources: the kernel functions.

5.5.3 Other techniques that improve kernel security

There is a wide range of techniques that can improve kernel security without reducing the kernel attack surface, I mention a few of them here.

One approach is to concede that in practice kernels are likely to be compromised

and the question of detecting and recovering from the intrusion is therefore important. For this purpose, kernel rootkit detection techniques have been proposed (e.g., [Car+09; Ses+07]), as well as attestation techniques. Clearly, such techniques are orthogonal to attack surface reduction which aims to prevent the kernel from being attacked in the first place.

Another approach is to prevent potential vulnerabilities in the source code from being exploitable, without aiming to remove the vulnerabilities [Cri+07; KPK12; St]. For instance, the PaX patch `UDEREF` prevents the kernel from (accidentally or maliciously) accessing user-space data, while the `KERNEXEC` patch prevents attacks where the attacker returns into code situated in user-space (with kernel privileges). SVA [Cri+07] compiles the existing kernel sources into a safe instruction set architecture which is translated into native instructions by the SVA VM, providing to a certain extent, among other guarantees, type safety and control flow integrity.

I consider all aforementioned techniques as supplemental to kernel attack surface reduction: they can be used in conjunction to improve overall kernel security. Indeed, kernel hardening techniques only aim for specific classes of vulnerabilities whereas kernel attack surface reduction does not discriminate between vulnerability classes. This makes it possible to deal with unknown vulnerability classes or vulnerability classes for which there exists no efficient counter-measure. As such, attack surface reduction can be seen as a more proactive approach in dealing with kernel vulnerabilities than kernel hardening, which is itself a more proactive approach than traditional patch management which focuses on individual vulnerabilities.

5.5.4 Summary

In a nutshell, KTRIM is at the confluence of two research currents: it attempts to bring together the security benefits of building small kernels and the convenience and compatibility aspects of process sandboxing and host-based intrusion detection. The advantages of each area of work are summarized in Table 5.8.

5.6 Summary

KTRIM is a lightweight, per-application, run-time kernel attack surface reduction framework that can restrict the amount of kernel code accessible to an attacker controlling a process. Such scenarios, in which attackers control a process and aim to attack the kernel, occur increasingly often [Ess11; Eva12] because of the rise of application sandboxes and the general increase in user-space hardening. The main goal of KTRIM is to provide a way of reducing the kernel attack surface in a quantifiable and non-bypassable way. KTRIM incurs rather low overhead (less than 3% for most performance-sensitive system calls), and can be seen as a generalisation of system-call-sandboxing to the level of kernel functions. My evaluation shows that attack surface reduction is significant (from 30% to 80%) both in terms of lines of code and CVEs: for example, out of a total of 262 kernel functions which had CVEs in the past and are reachable from the system call interface, KTRIM detects that 225 of those functions are unnecessary for the operation of the MYSQL daemon on my evaluation platform.

In its current state, KTRIM is suitable for use cases that are well-defined, typically server environments or embedded systems, because it uses run-time traces to establish the set of permitted functions for a given process (identified by its security context), which are then monitored and logged for violations. I envision the learning phase would be turned on when the server is tested prior to being put into production. In production, KTRIM will be enforcing and detections of kernel attacks will occur and could be reported, for example, to security incident and event management (SIEM) tools typically deployed in enterprises.

By using k-means clustering in the analysis phase, I have shown that convergence time can be significantly reduced for all four services observed: no false positives were observed for about a full year even under non-controlled, real-world usage of the evaluated applications.

In addition, this approach further confirms that the notion of attack surface is a powerful way to quantify security improvements: it would be very difficult to quantify improvements with traditional TCB size measurements. I foresee that this notion can have wider application, e.g., using the attack surface delimited thanks to KTRIM could be used to steer source code analysis work preferably towards code that is reachable to attackers, and to prioritize kernel hardening efforts and vulnerability research.

		Baseline	KTRIM			
			None	File	Cluster	Directory
sshd	Functions	31,429	9,166 (71%)	14,133 (55%)	19,769 (37%)	19,801 (37%)
	AS_{SLOC}	567,250	139,388 (75%)	236,998 (58%)	343,178 (40%)	346,650 (39%)
	AS_{cycl}	154,909	37,663 (76%)	68,937 (55%)	97,913 (37%)	99,615 (36%)
	AS_{CVE}	262	78 (70%)	152 (42%)	187 (29%)	170 (35%)
mysqld	Functions	31,429	7,498 (76%)	12,283 (61%)	18,284 (42%)	19,015 (39%)
	AS_{SLOC}	567,250	105,137 (81%)	199,366 (65%)	312,574 (45%)	332,238 (41%)
	AS_{cycl}	154,909	28,571 (82%)	59,370 (62%)	89,924 (42%)	95,737 (38%)
	AS_{CVE}	262	37 (86%)	111 (58%)	162 (38%)	165 (37%)
ntpd	Functions	31,429	8,569 (73%)	13,306 (58%)	18,997 (40%)	19,336 (38%)
	AS_{SLOC}	567,250	126,559 (78%)	215,405 (62%)	327,137 (42%)	339,449 (40%)
	AS_{cycl}	154,909	34,334 (78%)	64,009 (59%)	93,959 (39%)	97,519 (37%)
	AS_{CVE}	262	69 (74%)	134 (49%)	170 (35%)	170 (35%)
qemu-kvm	Functions	31,429	11,223 (64%)	16,026 (49%)	19,993 (36%)	22,685 (28%)
	AS_{SLOC}	567,250	181,603 (68%)	271,959 (52%)	346,148 (39%)	395,675 (30%)
	AS_{cycl}	154,909	49,813 (68%)	79,608 (49%)	99,046 (36%)	112,783 (27%)
	AS_{CVE}	262	92 (65%)	155 (41%)	187 (29%)	174 (34%)

Table 5.2. Summary of KTRIM attack surface reduction results for four grouping algorithms in the analysis phase (None, File, Directory, and Cluster). The term *functions* refers to the number of functions in the STATICSEC attack surface.

		None	File	Cluster	Directory
`sshd`	Convergence rate	26%	26%	12%	20%
	False positives at 20%	20	3	0	0
`mysqld`	Convergence rate	26%	26%	12%	19%
	False positives at 20%	38	4	0	0
`ntpd`	Convergence rate	26%	20%	12%	14%
	False positives at 20%	10	0	0	0
`qemu-kvm`	Convergence rate	18%	18%	11%	11%
	False positives at 20%	0	0	0	0

Table 5.3. Convergence rate (convergence time to 0 false-positives by total observation time) and number of false positives for all analysis phase algorithms for four applications. A false positive is a (unique) function which is called during the enforcement phase by a program, but is not in the enforcement or system set.

	Baseline	KTRIM	Overhead
`open` and `close`	2.78	2.80	0.8%
Null I/O	.19	.19	0%
`stat`	1.85	1.86	0.5%
TCP `select`	2.52	2.65	5.2%
`fork` and `exec`	547	622	14%
`fork` and `exec sh`	1972	2025	2.7%
File create	31.6	55.4	75%
`mmap`	105.3K	107.5K	2.1%
Page fault	.1672	.1679	0.4%

Table 5.4. Latency time and overhead for various OS operations (in microseconds)

	Baseline	KTRIM	Without pre-learning
Average	2.30 ± 0.00	2.31 ± 0.00	4.67 ± 0.01
Overhead		0.4%	103%

Table 5.5. MySQL-slap benchmark: average time to execute 5000 SQL queries (in seconds)

	None	File	Cluster	Directory
CVE-2013-2094 (Perf.)	✓	–	–	–
CVE-2012-0056 (Mem.)	✓	M	✓	M
CVE-2010-4158 (BPF)	S, M	–	–	–
CVE-2010-3904 (RDS)	✓	✓	✓	✓

Table 5.6. Detection of previously exploited kernel vulnerabilities by KTRIM (for each grouping). Legend: ✓: detected for all use cases, S: detected in the `sshd` context, M: detected in the `mysqld` context, –: undetected.

		Baseline	KTRIM			
			None	File	Cluster	Directory
php5	Functions	26,240	6,821 (74%)	9,911 (62%)	14,242 (46%)	14,994 (43%)
	AS_{SLOC}	661,945	163,146 (75%)	235,110 (64%)	353,824 (47%)	361,389 (45%)
	AS_{cycl}	151,577	37,440 (75%)	52,657 (65%)	80,388 (47%)	80,881 (47%)
	AS_{CVE}	257	67 (74%)	117 (54%)	174 (32%)	159 (38%)
apache2	Functions	26,240	7,529 (71%)	11,944 (54%)	14,966 (43%)	17,152 (35%)
	AS_{SLOC}	661,945	187,114 (72%)	309,929 (53%)	388,998 (41%)	454,747 (31%)
	AS_{cycl}	151,577	43,166 (72%)	70,290 (54%)	89,100 (41%)	104,414 (31%)
	AS_{CVE}	257	85 (67%)	152 (41%)	192 (25%)	204 (21%)
mysqld	Functions	26,240	6,075 (77%)	8,561 (67%)	11,893 (55%)	14,985 (43%)
	AS_{SLOC}	661,945	145,742 (78%)	202,768 (69%)	304,501 (54%)	384,333 (42%)
	AS_{cycl}	151,577	33,692 (78%)	46,032 (70%)	69,181 (54%)	87,452 (42%)
	AS_{CVE}	257	45 (82%)	78 (70%)	154 (40%)	179 (30%)

Table 5.7. KTRIM attack surface reduction results for the synthetic LAMP workload.

	Compatibility	Performance	Non-Bypassable	Quantifiable	Automated
Microkernel	– –	∼	✓✓	✓	n/a
Kernel specialization	–	✓✓	✓✓	✓	✓
Anomalous syscall	✓✓	∼	–	–	✓
Seccomp	✓	✓	∼	–	–
KTRIM	✓✓	∼	✓	✓	✓

Table 5.8. Succinct comparison of various approaches that can reduce the kernel attack surface. The term *compatibility* refers to the ease of using the approach with existing software, middleware or hardware, and the term *quanitifiable* refers to the existence of attack surface measurements. The ∼ sign refers to cases where results may vary between good (✓) and bad (–).

This `NULL` *page intentionally mapped.*

6 Comparison and Case Study

> “If the truth were known about the origin of the word “Jazz” it would never be mentioned in polite society.”
>
> Attributed to *Clay Smith*

6.1 Comparison

I now provide a comparison of compile-time tailoring and run-time trimming. Table 6.1 summarizes this comparison.

	Tailoring	Trimming
Attack Surface Reduction		✓
False Positives	✓	
Enforcement	∼	∼
Performance	✓	
Usability		✓

Table 6.1. Succinct comparison of the trade-offs between compile-time tailoring vs. run-time trimming as presented in this thesis. The check mark is attributed to the approach with the advantage (if any).

6.1.1 Attack Surface Reduction

The attack surface reduction that can be achieved by trimming (with no grouping) is inherently higher than that achievable with tailoring. This is a simple matter of granularity: discerning at the level of individual functions (with more than 60K functions) will lead to a lower attack surface than discerning at the level of kernel configuration options (about 5K options).

However, the attack surface numbers in both evaluations cannot be directly compared. Indeed, we use the STATICSEC model for trimming whereas tailoring is evaluated under the GENSEC and ISOLSEC models. Trimming with no grouping essentially achieves the same attack surface reduction percentage than tailoring (about 80%). However, because the STATICSEC model is more restrictive (in the sense that it assumes less kernel functionality may be reachable to the attacker), this is in fact a result in favor of trimming.

To substantiate this claim, we have also performed attack surface reduction measurements when using the STATICSEC model for tailoring [Dec14]. In STATICSEC, the attack surface reduction achieved with tailoring is about 30% on average (across different use cases), compared to about 80% for trimming with no grouping. It then becomes clear that the attack surface achieved with tailoring is about 3 to 4 times higher than that achieved with trimming.

Finally, the attack surface obtained after trimming (without grouping) is a subset of the attack surface obtained with tailoring. Hence, this confirms the intuition that there is little benefit in performing trimming and tailoring at the same time (in terms of attack surface reduction).

6.1.2 False positives

Both approaches are based on a training phase whereby a trace of the workload is recorded. This is inherently prone to false-positives in the sense that the workload may significantly change and the tailored/trimmed kernel functionality may be insufficient to satisfy the changing workload. This is not necessarily a major inconvenience, since both approaches are designed for systems where the workload is essentially known in advance (e.g., servers, embedded systems). In addition, in the case of trimming, there is the possibility of additional flexibility in the enforcement phase as I discuss next.

Nevertheless, experiments show that in the case of tailoring, the convergence rate to

the target kernel functionality is very fast (e.g., about 5 minutes of a synthetic workload is sufficient). The same cannot be said for trimming, since one needs in the order of one to three months.

6.1.3 Enforcement

In the case of tailoring, there is only one possible enforcement strategy: relying on the existing kernel code (and application code) to deal with cases where a kernel functionality has not been compiled into the kernel. For instance, if one attempts to create a socket of an unknown type, the system call will simply return an error.

In the case of trimming, there are many possible enforcement strategies, although many have drawbacks. For instance, to return an error on a system call is, in the general case, impossible. Indeed, the violation is detected deep in the kernel, when locks might have been taken or other shared kernel state manipulated. Because no exception mechanism that would unwind such locks exist (or can be implemented without significant overhaul of the kernel architecture), this is not a satisfying strategy. An alternative, crashing the kernel on violations, is also too drastic in the presence of false positives, as it clearly creates the opportunity for denial of service attacks from attackers.

However, there are two viable options (both of which I have implemented and experimented with). The first one consists in detecting the violation. Care should be taken as to how this detection is done (e.g., send UDP packet to a remote trusted host in the `kprobe` handler). The second consists in gracefully degrading the kernel's performance, by trading-off some CPU cycles for increased checks (i.e., "kernel hardening"). This is performed by compiling the kernel in two versions (one base and one hardened version) whereby it is possibly to chose, in terms of functionality, any of the two versions of each kernel function. The hardened version is compiled with additional checks (including kernel stack clearance, kernel stack exhaustion checks, kernel function pointer protection). Of course, both of these options can be used simultaneously. The advantage of such approaches is that they provide additional flexibility for changing workloads: for instance, in the second case, false-positives will only result in reduced performance. And since the kernel functionality exercised in the false-positives is seldom used, this performance reduction is unlikely to be perceptible.

In conclusion, none of the two approaches have a clear advantage over the other in terms of enforcement. Although tailoring provides a cleaner interface for denying the use of removed functionality, trimming also provides the option of still allowing running such "unnecessary" functionality.

6.1.4 Performance

By design, tailoring has no performance impact: the exact same kernel functions remain. In fact, it could be argued that one could observe improvements under some workloads due to caching effects. Indeed, the kernel code size is smaller, which means it's more likely that memory locality is more likely to benefit kernel code. However, I did not observe any measurable difference in benchmarks, this effect is therefore not significant (at least on the servers and workloads we benchmarked).

The same cannot be said about trimming. Trimming makes use of dynamic instrumentation (via KPROBES), which can result in significant overhead, especially if commonly used kernel functions are instrumented. However, because of the pre-learning phase heuristic, this is mitigated in favor of a slightly larger attack surface. Hence, the observed performance overhead remains low. In future work, these performance optimizations could be further investigated. One could consider making use of more performant instrumentation techniques (e.g., making use of static instrumentation), and optimizing the approach through which kernel functions should be instrumented. For instance, a static instrumentation could first check whether the current process is part of the set of processes for which trimming occurs. If not, (e.g., for processes running as *root*), the static instrumentation can directly skip over the instrumentation checks.

6.1.5 Usability

Tailoring requires recompiling the kernel, whereas trimming does not. Recompiling the kernel can be a significant hindrance in practice. Some kernel distributors will not provide support for recompiled kernels. Similarly, recompiling the kernel implies rebooting, which in some use cases (such as servers) can be undesirable.

6.1.6 Summary

In use cases where applications, kernel, and hardware will rarely be updated, tailoring is a particularly good fit. This generally corresponds to embedded systems where Linux is used: for example set-top boxes, routers, switches, industrial control systems, medical instruments. Because such systems are rarely updated, attack surface reduction can delay or prevent such devices from being exploited by unpatched vulnerabilities. This is simply because, by analyzing Linux kernel CVEs we observe that fewer of these vulnerabilities exist on a tailored kernel in all use cases. In use cases where flexibility is more important, but where use cases are also known in advance, kernel trimming may be used. Although false positives may exist, because one may simply use the system in detection mode, they will not affect the stability of the system. Servers (such as cloud services) are a particularly good example. Between significant applications and kernel updates, servers will go through testing. This testing phase can be performed at the same time as the learning phase.

6.2 Case study

In this section, I present a case study of a vulnerability assessment of two architectures for implementing a cloud storage service. The analysis demonstrates the value of kernel self-protection techniques such as the two attack surface reduction techniques presented in this thesis.

6.2.1 Overview

Storage cloud services allow the sharing of storage infrastructure among multiple customers and hence significantly reduce costs. Typically, such services provide object or filesystem access over a network to the shared distributed infrastructure. To support multiple customers or *tenants* concurrently, the network-filesystem-access services must be properly isolated with minimal performance impact.

We consider here a *filesystem storage cloud* as a public cloud storage service used by customers to mount their own filesystems remotely through well-established network filesystem protocols such as NFS and the Common Internet Filesystem (CIFS, also known as *SMB*). Such a service constitutes a highly scalable, performant, and reliable

enterprise network-attached storage (NAS) accessible over the Internet that provides services to multiple tenants.

In general, a cloud service can be run at any of the following increasing levels of multi-tenancy:

- *Hardware level*: server hardware, OS, and application dedicated per client.
- *Hypervisor level*: share server hardware, and use virtualization to host dedicated OS and application per client.
- *OS level*: share server hardware and OS, and run a dedicated application per client.
- *Application level*: share server hardware, OS, and application server among clients.

Intuitively, the higher the level of multi-tenancy, the easier it seems to achieve a resource-efficient design and implementation; at the same time, though, it gets harder (conceptually and in terms of development effort) to securely isolate the clients from each other.

In this case study, we investigate a *hypervisor-level* and an *OS-level* multi-tenant filesystem storage cloud architecture, and compare them in terms of security, especially with respect to kernel protection. The hypervisor-level multi-tenancy approach is based on hardware virtualization (with para-virtualized drivers for improved networking performance). We refer to this architecture as the *virtualization-based multi-tenancy (VMT) architecture*. The OS-level multi-tenancy approach uses mandatory access control (MAC) in the Linux kernel and is capable of isolating customer-dedicated user-space services on the same OS. Such an architecture may also leverage, for instance, OS-level virtualization technologies such as *OpenVZ* or *Linux Containers (LXC)*. We refer to this architecture as the *operating-system-based multi-tenancy (OSMT) architecture* in the remainder of this study.

We have implemented both approaches on real hardware in the *IBM Scale-out NAS (SONAS)* [Son] and the *IBM General Parallel Filesystem (GPFS)* [SH02] technologies. We used open-source components such as *KVM* [Kiv+07] with *virtio* networking for virtualization and *SELinux* (`http://selinuxproject.org/`) for MAC.

6.2.2 Background

One can distinguish the following categories of general-purpose storage clouds (ignoring storage clouds that provide database-like structures on content):

- *Block storage clouds*, with a block-level interface, i.e., an interface that allows the writing and reading of fixed-sized blocks. Examples of such clouds include *Amazon EBS*.

- *Object storage clouds*, composed of buckets (or containers) that contain objects (or blobs). These objects are referred to by a key (or name). The API is usually very simple: typically a REST API with *create* and *remove* operations on buckets and *put*, *get*, *delete*, and *list* operations on objects. Example of such storage clouds include *Amazon S3*, *Rackspace Cloudfiles*, and *Azure Storage Blobs*.

- *Filesystem storage clouds*, with a full-fledged filesystem interface, therefore referred to also as "cloud NAS." Examples of such clouds include *Nirvanix CloudNAS*, *Azure Drive*, and *IBM Scale-Out Network Attached Storage (SONAS)*.

Application-level multi-tenancy is sometimes also referred to as native multi-tenancy. Some authors consider it the cleanest way to isolate multiple tenants [Cai+11]. However, achieving multi-tenancy securely is very challenging and therefore not common for filesystem storage clouds. The reasons lie in the complex nature of this task: unlike other types of storage clouds, filesystem storage clouds possess complex APIs that have evolved over time, which leads to large attack surfaces. The vulnerability track record of these applications seems to confirm this intuition. CIFS servers were vulnerable to various buffer-overflows (e.g., CVE-2010-3069, CVE-2010-2063, CVE-2007-2446, CVE-2003-0085, CVE-2002-1318, see `http://cve.mitre.org/`), format string vulnerability leading to arbitrary code execution (CVE-2009-1886), directory traversals (CVE-2010-0926, CVE-2001-1162), while NFS servers were also vulnerable to similar classic vulnerabilities as well as more specific ones such as filehandle vulnerabilities [Tra+04]. Moreover, adding multi-tenancy support into these server applications would require significant development (e.g., in order to distinguish between different authentication servers for specific filesystem exports) which will most likely result in new vulnerabilities. We discuss in Sections 6.2.3 and 6.2.4 architectures with lower levels of multi-tenancy. They effectively restrict the impact of arbitrary code execution vulnerabilities to the realm of a single tenant: by definition, this cannot be achieved with application-level multi-tenancy.

This study targets the IBM SONAS [Son] platform, which evolved from the IBM Scale-Out File Services (SoFS) [Oeh+08]. IBM SONAS provides a highly scalable

network-attached storage service, and therefore serves as a typical example of a filesystem storage cloud. IBM SONAS currently contains support for hardware-level multi-tenancy according to the architectures discussed in this work. Adding a higher-level of multi-tenancy is an important step to reduce the cost of a cloud-service provider.

6.2.3 System Description

Section 6.2.3.1 gives an overview of the general architecture of a filesystem storage cloud. Section 6.2.3.2 describes the MAC policies which are used in both architectures. Sections 6.2.3.3 and 6.2.3.4 introduce the two alternatives, detailing the internals of the interface nodes, the key element of the filesystem storage cloud architecture.

6.2.3.1 General Description

Figure 6.1 depicts the general architecture of a filesystem storage cloud that consists of the following elements:

- Customers and users: A *customer* is an entity (e.g., a company) that uses at least one network file system. A customer can have multiple individual *users*. We assume that multiple customers connect to the filesystem storage cloud and that each customer has a separate set of users. Data is separated between users from distinct customers, and user IDs are in a separate namespace for each customer.

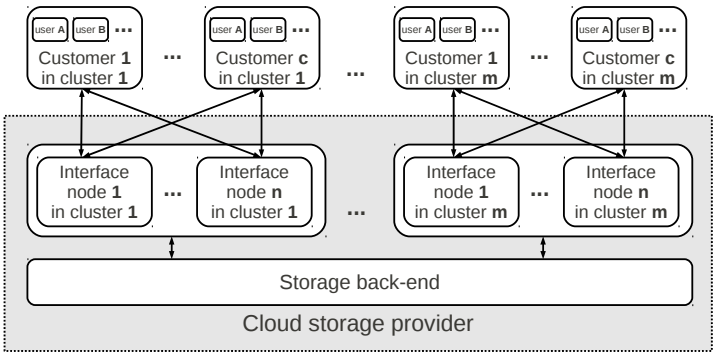


Figure 6.1. General architecture of a filesystem storage cloud.

Hence two distinct customers may allocate the same user ID without any conflict on the interface nodes or in the storage back-end.

- Interface nodes and cluster: An interface node is a system running *filer services* such as NFS or CIFS daemons. Interface nodes administratively and physically belong to the cloud service provider and serve multiple customers. A customer connects to the filesystem storage cloud through the interface nodes and mounts its filesystems over the Internet. Multiple interface nodes together form an interface cluster, and one interface node may serve multiple customers. A customer connects only to nodes in one interface cluster.

- Shared back-end storage: The shared back-end storage provides block-level storage for user data. It is accessible from the interface clusters over a network using a distributed filesystem such as GPFS [SH02]. It is designed to be reliable, highly available, and performant. We assume that no security mechanism exists within the distributed filesystem to authenticate and authorize nodes of the cluster internally.

- Customer boarding, unboarding, and configuration: Typically, interface nodes must be created, configured, started, stopped, or removed when customers are boarded (added to the service) or unboarded (removed from the service). This is performed by administration nodes not shown here, which register customer accounts and configure filesystems. Ideally, boarding and unboarding should consume a minimal amount of system resources and time.

As an example, a customer registers a filesystem with a given size from the filesystem storage cloud provider, and then configures machines on the customer site that mount this filesystem. The users of the customer can then use the cloud filesystem similar to how they use a local filesystem. Customers connect to the interface cluster via a dedicated physical wide-area network link or via a dedicated VPN over the Internet, ideally with low latency. The cloud provider may limit the maximal bandwidth on a customer link.

To ensure high availability and high throughput, a customer accesses the storage cloud through the clustered interface nodes. Interface nodes have to perform synchronization tasks within their cluster and with the back-end storage, generating additional traffic. An interface node has three network interfaces: one to the customer, one to other nodes in the cluster, and one to the back-end storage.

Dimensioning. The size of a filesystem storage cloud is determined by the following parameters, which are derived from service-level agreements and from the (expected or observed) load in the system: the number of customers c assigned to an interface cluster, the number of interface nodes n in a cluster (due to synchronization overhead, this incurs a trade-off between higher availability and better performance), and the number of clusters m attached to the same storage back-end.

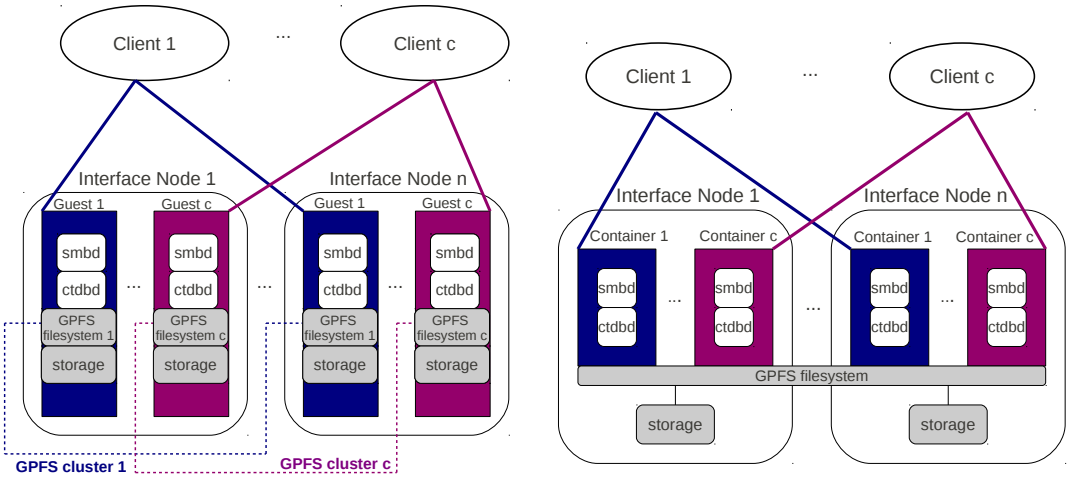
Customer and user authentication. Describing customer authentication would exceed the scope of this work; in practice, it can be delegated to the customer's VPN endpoint in the premises of the service-cloud provider. The authentication of users from a given customer also requires that customers provide a directory service that will serve authentication requests made by users. Such a directory service can be physically located on the customer's premises and under its administration or as separate service in the cloud. In either case, users authenticate to an interface node, which in turn relays such requests to the authentication service of the customer.

6.2.3.2 Mandatory Access Control Policies

We use mandatory access control on the filer services. In case of their compromise, MAC provides a first layer of defense on both architectures. For practical reasons, we have used SELinux. Other popular choices include *grsecurity RBAC* [St] or *TOMOYO*. These MAC systems limit the privileges of the filer services to those required, effectively creating a sandbox environment, by enforcing policies that are essentially a list of permitted operations (e.g., open certain files, bind certain ports, fork, ...).

As an example, the policies on the interface nodes basically permit the filer services to perform the following operations: bind on their listening port and accept connections, perform all filesystem operations on the customer's data directory (which resides on a distributed filesystem), append to the filer log files, and read the relevant service configuration files.

The protection provided by these policies can be defeated in two ways. One possibility is if the attacker manages to execute arbitrary code in kernel context (e.g., through a local kernel exploit), in which case it is trivial to disable any protections provided by the kernel, including MAC. The second possibility is by exploiting a hole in the SELinux policy, which would be the case, for example, if a filer service were authorized to load a kernel module.



(a) VMT architecture (b) OSMT architecture

Figure 6.2. Two architectures for multi-tenancy, shown for one interface cluster of each architecture.

An important example of the benefit of these policies is the restriction of accessible network interfaces to the customer and intra-cluster network interfaces only. Another example is the impossibility for processes running in the security context of the filer services to write to files they can execute, or to use `mmap()` and `mprotect()` to get around this restriction. In practice, this means, for example, that an attacker making use of a remote exploit on a filer service cannot just obtain a shell and download and execute a local kernel exploit: the attacker would have to find a way to execute the latter exploit directly within the first exploit, which, depending on the specifics of the vulnerabilities and memory protection features, can be impossible.

Note that, because of the way MAC policies are specified — that is, by white-listing the permitted operations — these examples (network interface access denied, no read and execute permission) are a consequence of the policy and do not have to be explicitly specified, which encourages policies to be built according to the least privilege principle.

6.2.3.3 VMT Architecture

We now introduce the first architecture, called the *virtualization-based multi-tenancy (VMT) architecture*. It is based on *KVM* as a hypervisor and implements multi-tenancy by running multiple virtual interface nodes as guests on the hardware of one physical

interface node. Such a filesystem storage cloud has a fixed number of physical interface nodes in every cluster, with each interface node running one guest for each customer. All guests that belong to the same customer form an interface-node cluster, which maintains the distributed filesystem with the data of the customer (labeled *GPFS cluster* in Figure 6.2(a), as explained below). Each virtual machine (VM) runs one instance of the required filer-service daemons, exporting the filesystems through the CIFS or NFS protocols, and has three separate network interfaces.

In terms of isolation, MAC can be applied at two levels in this architecture. The first level is inside a guest, for protecting filer services exposed to the external attackers, using the exact same policies as in the OSMT architecture[1] described below. The second level is on the host, with the idea of sandboxing guests (i.e., QEMU processes running on the host, in the case of KVM) by using multi-category security. Policies at this level do not depend on what is running inside the guests, therefore they can be applied to many virtualization scenarios. Such policies already exist and are implemented by sVirt (see `http://selinuxproject.org/page/SVirt`).

Figure 6.2(a) shows one of m clusters according to the VMT architecture, in which the distributed filesystem is GPFS and the daemons in each virtual machine are `smbd` (the Samba daemon, for CIFS) and `ctdbd` (the clustered trivial database daemon, used to synchronize meta-data among cluster nodes). They work together to export customer data using the CIFS protocol. The customers are also shown and connect only to their dedicated VM on each interface node. In terms of the dimension parameters from Figure 6.1, for every one of the m interface clusters, there are c *GPFS clusters*, each corresponding to a GPFS filesystem, and $c \cdot n$ guest virtual machines (n per customer) in this architecture.

When a new customer is boarded, it is assigned to a cluster and a configuration script automatically starts additional guests for that customer on all the physical interface nodes within this cluster. Furthermore, a new GPFS filesystem and cluster are created for the customer on the new virtual guests. Customer data can then be copied to the filesystem and accessed by users.

[1] Except for the use of the multi-category security functionality (see Section 6.2.3.4): categories are not required in the VMT architecture as only one customer resides in each guest.

6.2.3.4 OSMT Architecture

The second architecture, called the *operating-system-based multi-tenancy (OSMT) architecture*, is based on a lightweight separation of OS resources by the kernel. OS-level virtualization of this form can be achieved using *containers*, such as *LXC*, *OpenVZ* or *Zap pods* [Osm+02] for Linux, *jails* [KW00] for FreeBSD, and *zones* [PT04] for Solaris. Containers do not virtualize the underlying hardware, and thus cannot run multiple different OSes, but create new isolated namespaces for some OS resources such as filesystems, network interfaces, and processes. Processes running within a container are isolated from processes within other containers, thus they seem to be running in a separate OS. All processes share the same kernel, hence, one cannot encapsulate applications that rely on kernel threads in containers (such as the kernel NFS daemon).

In our implementation, isolation is performed using *SELinux multi-category-security (MCS)* policies [McC04] for shielding the processes that serve a particular customer from all others. It is then sufficient to write a single policy for each filer service that applies for all customers by simply adjusting the category of each filer service and other related components inside a container (e.g., configuration files, customer data). This ensures that no two distinct customers can access each other's resources (because they belong to different categories). In comparison to the VMT architecture, the policies in the guest that contain the filer services, and the policies in the host that isolate the customers are now combined into a single policy which achieves the same goals.

In addition, a *change-root (chroot)* environment is installed, whose only purpose is to simplify the configuration of the isolated services and the file labeling for SELinux. We refer to such a customer isolation domain as a *container* in the remainder of this work. A container dedicated to one customer on an interface node consists of a *chroot directory* in the root filesystem of the interface node, which contains all files required to access the filesystem for that customer. All required daemons accessed by the customer run within the container. Because of the chroot environment, the default path names, all configuration files, the logfile locations, and so on, are all the same or found at the same locations for every customer; this is implemented through read-only mount binds, without having to copy files or create hard links. This approach makes our container-based setup amenable to automatic maintenance through existing software distribution and packaging tools.

This form of isolation does not provide new namespaces for some critical kernel

resources (process identifiers are global, for instance); it does not allow a limitation of memory and CPU usage either. However, it causes a smaller overhead for isolation than hypervisor-based virtualization does.

Figure 6.2(b) shows an interface cluster following the OSMT architecture, in which the distributed filesystem is GPFS, shared by all containers within a cluster. Each container runs a single instance of each of the `smbd` and `ctdbd` daemons, accessed only by the corresponding customer. In the terms of the dimension parameters from Figure 6.1, for every one of the m interface clusters, there exists *one* GPFS filesystem (only one per cluster), n kernels (each kernel is shared by c customers), and $c \cdot n$ containers (n per customer).

Customer boarding is done by a script that creates an additional container on every interface node in the cluster, and a data directory for that customer in the shared distributed filesystem of the interface cluster. The daemons running inside the new containers are configured to export the customer's data directory using the protocols selected. No changes have to be made to the configuration of the distributed filesystem on the interface nodes.

6.2.4 Security Comparison

In this subsection, we discuss the differences between the VMT architecture and the OSMT architecture from a security viewpoint. Because we aim to compare the two approaches, we only briefly touch on those security aspects that are equal for the two architectures. This concerns, for instance, user authentication, attacks from one user of a customer against other users of the same customer, and attacks by the service provider ("insider attacks" from its administrators). These aspects generally depend on the network filesystem and the user-authentication method chosen, as well as their implementations. They critically affect the security of the overall solution, but are not considered further here.

6.2.4.1 Security Model

We consider only attacks by a malicious customer, i.e., attacks mounted from a user assigned to one customer against the service provider or against other customers. In accordance with the traditional goals of information security, we can distinguish three

types of attacks: those compromising the confidentiality, the integrity, or the availability of the service and/or of data from other customers.

Below we group attacks in two categories. First we discuss *denial-of-service (DoS) attacks* targeting service availability in Section 6.2.4.3. Second, we subsume threats against the confidentiality and integrity of data under *unauthorized data access* and discuss them in Section 6.2.4.4.

We assume that the cloud service provider is trusted by the customers. We also disregard customer-side cryptographic protection methods, such as filesystem encryption [DW10] and data-integrity protection [SWZ05]. These techniques would not only secure the customer's data against attacks from the provider but also protect its data from other customers. Such solutions can be implemented by the customer transparently to the service provider and generally come with their own cost (such as key management or the need for local trusted storage).

6.2.4.2 Comparison Method

An adversary may compromise a component of the system or the whole system with a certain *likelihood*, which depends on the vulnerability of the component and on properties of the adversary such as its determination, its skills, the resources it invests in an attack and so on. This likelihood is influenced by many factors, and we refrain from assigning numerical values or probabilities to it, as it cannot be evaluated with any reasonable accuracy [Sch04, Chap. 3–4].

Instead we group all attacks into three sets according to the likelihood that an attack is feasible with methods known today or the likelihood of discovering an exploitable vulnerability that immediately enables the attack. We roughly estimate the relative severity of attacks and vulnerabilities according to criteria widely accepted by computer emergency readiness teams (CERTs), such as previous exploits or their attack surfaces. Our three likelihood classes are described by the terms *unlikely*, *somewhat likely* and *likely*.

In Section 6.2.4.4 we model data compromise in the filesystem storage cloud through graphical *attack trees* [Sch99]. They describe how an attacker can reach its goal over various paths; the graphs allow a visual comparison of the security of the architectures.

More precisely, an attack tree is a directed graph, whose nodes correspond to states of the system. The initial state is shown in white (meaning that the attacker obtains

an account on the storage cloud) and the exit node is colored black (meaning that the attacker gained unauthorized access to another customer's data). A state designates a component of the system (as described in the architecture) together with an indication of the security violation the attacker could have achieved or of how the attacker could have reached this state.

An edge corresponds to an attack that could be exploited by an attacker to advance the state of compromise of the system. The intermediate nodes are shown in various shades of gray, roughly corresponding to the severity of the compromise. Every attack is labeled by a likelihood (unlikely, somewhat likely, or likely), represented by the type of arrow used.

6.2.4.3 Denial-of-Service Attacks

Server crashes. An attacker can exploit software bugs causing various components of an interface node to crash, such as the filer services (e.g., the NFS or CIFS daemon) or the OS kernel serving the customer. Such crashes are relatively easy to detect and the service provider can recover from them easily by restarting the component. Usually such an attack can also be attributed to a specific customer because the service provider maintains billing information of the customer; hence the offending customer can easily be banned from the system.

Both architectures involve running dedicated copies of the filer services for each customer. Therefore, crashing a filer service only affects the customer itself. Although the attack may appear likely in our terminology, we consider it not to be a threat because of the external recovery methods available.

Note that non-malicious faults or random crashes of components are not a concern because all components are replicated inside an interface cluster, which means that the service as a whole remains available. Crashes due to malicious attacks, on the other hand, will affect all nodes in a cluster as the attacker can repeat its attack.

Furthermore, any server crash has to be carried out remotely and therefore mainly affects the network stack. It appears much easier for a local user to crash a server, in contrast. For this, the attacker must previously obtain the privilege to execute code on the interface node, most likely through an exploit in one of the filer services. However, when attackers have obtained a local account on an interface node, they can cause much more severe problems than simply causing a crash (Section 6.2.4.4). Therefore we consider a

locally mounted DoS attack as an acceptable threat.

In the *VMT architecture*, a kernel crash that occurs only inside the virtual machine dedicated to the customer does not affect other customers, which run in other guests — at least according to the generally accepted view of virtual-machine security. However, the effects on other guests depend on the kind of the DoS attack. A network attack that exploits a vulnerability in the upper part of the network stack (e.g., UDP) most likely only crashes the targeted guest. But an attack on lower-layer components of the hypervisor (e.g., network interface driver), which run in the host, can crash the host and all guests at once. Moreover, additional vulnerabilities may be introduced through the hypervisor itself.

In the *OSMT architecture*, an attacker may crash the OS kernel (through a vulnerability in the network interface driver or a bug in the network stack), which results in the crash of the entire interface node and disables also the service to all other customers. Thus, the class of DoS attacks targeted against the OS kernel has a greater effect than in the VMT architecture.

Resource exhaustion. An attacker can try to submit many filesystem requests to exhaust some resource, such as the network between the customers and the interface nodes, the network between interface nodes and the resource cluster, or the available CPU and memory resources on the interface nodes. Network-resource exhaustion attacks affect both our designs in the same way (and more generally, are a common problem in most Internet services); therefore, we do not consider them further and discuss only the exhaustion of host resources.

In the *VMT architecture*, hypervisors can impose a memory bound on a guest OS and limit the number of CPUs that a guest can use. For example, a six-CPU interface node may be shared by six customers in our setup. Limiting every guest to two CPUs means that the interface node still tolerates two malicious customers that utilize all computation power of their dedicated guests, but continues to serve the other four customers with two CPUs.

The impact of a resource-exhaustion attack with a container setup in the *OSMT architecture* depends on the container technology used and its configuration.

In our study, we use a container technology (SELinux and chroot environment) that cannot restrict the CPU used or the memory consumed by a particular customer. Given proper dimensioning of the available CPU and memory resources with respect to the

expected maximal load per customer, however, a fair resource scheduler alone can be sufficient to render such attacks harmless.

With more advanced container technology, such as LXC (based on the recent cgroup process grouping feature of the Linux kernel), it is possible to impose fine-grained restrictions on these resources, analogously to a hypervisor. For instance, the number of CPUs attributed to a customer and the maximally used CPU percentage can be limited for every customer.

6.2.4.4 Unauthorized Data Access

We describe here the attack graphs in Figures 6.3 and 6.4 as explained in Section 6.2.4.2. Some attacks are common and apply to both architectures; they are described first. We then present specific attacks against the VMT and OSMT architectures. Each attack graph includes all attacks relevant for the architecture.

Common attacks.

Filer service compromise. Various memory corruption vulnerabilities (such as buffer overflows, string format vulnerabilities, double frees) are notorious for allowing attackers to execute arbitrary code with the privileges of the filer service. However, protection measures such as address space layout randomization, non-executable pages, position-independent executables, and stack canaries, can render many attacks impossible without additional vulnerabilities (e.g., information leaks). This is especially true for remote attacks, in which the attacker has very little information (e.g., no access to `/proc/pid/`) and less control over memory contents (e.g., no possibility of attacker-supplied environment variables) than for local attacks. For these reasons, we categorize these attacks as "somewhat likely."

Complementing the aforementioned attacks that permit arbitrary code execution, confused deputy attacks [Har88] form a weaker class of attacks. In such an attack, the attacker lures the target application into performing an operation unauthorized to the attacker without obtaining arbitrary code execution. Directory traversal, whereby an attacker tricks the filer service into serving files from a directory that should not be accessible, is a famous example of such attacks in the context of storage services (e.g., CVE-2010-0926, CVE-2001-1162 for CIFS). Clearly, such attacks leverage the privileges of the target process: a process that has restricted privileges is not vulnerable. Therefore, they form a weaker class of attacks: preventing unauthorized data access

to an attacker who has compromised the filer service through arbitrary code execution also prevents these attacks. Furthermore, confused deputy attacks are very unlikely to serve as a stepping stone for a second attack (e.g., accessing the internal network interface), which would be required to access another tenant's data in both architectures here. Consequently, we do not consider confused deputy attacks any further.

Kernel compromise. We distinguish between remote and local kernel attacks. The reasoning in the previous paragraph concerning the lack of information and memory control is essentially also valid for remote kernel exploits. However, for the kernel, the attack surface is much more restricted: typically network drivers and protocols, and usually under restrictive conditions (e.g., LAN access). Recently, Wi-Fi drivers have been found to be vulnerable (CVE-2008-4395), as well as the SCTP protocol (CVE-2009-0065) both of which would not be used in the context of a filesystem storage cloud. For these reasons, we categorize these attacks as "unlikely." In contrast to remote exploits, we categorize local kernel exploits as "somewhat likely" given the information advantage (e.g., `/proc/kallsysms`) and capabilities of a local attacker (e.g., mapping a fixed memory location). Many recently discovered local kernel vulnerabilities, and the general lack of kernel self-protection techniques used in practice, confirms this view.

SELinux bypass. The protection provided by SELinux can be bypassed in two ways. One of them is by leveraging a mistake in the security policy written for the application: if the policy is too permissive, the attacker can find ways to get around some restrictions. An example of such a policy vulnerability was found in sVirt [Woj]: an excessively permissive rule in the policy allowed an attacker in the hypervisor context to write directly to the physical drive, which the attacker can leverage in many ways to elevate his privileges. The second option for bypassing SELinux is by leveraging a SELinux implementation bug in the kernel. An example of such a vulnerability is the bypass of NULL pointer dereference protections. The Linux kernel performs checks when performing `mmap()` to prevent a user from mapping addresses lower than `mmap_min_addr` (which is required for exploiting kernel NULL pointer dereferences vulnerabilities). SELinux also implemented such a protection (with the additional possibility of allowing such an operation for some trusted applications). However, the SELinux access control decision in the kernel would basically override the `mmap_min_addr` check, weakening the security of the default kernel (CVE-2009-2695). For these reasons, we categorize these attacks as "somewhat likely."

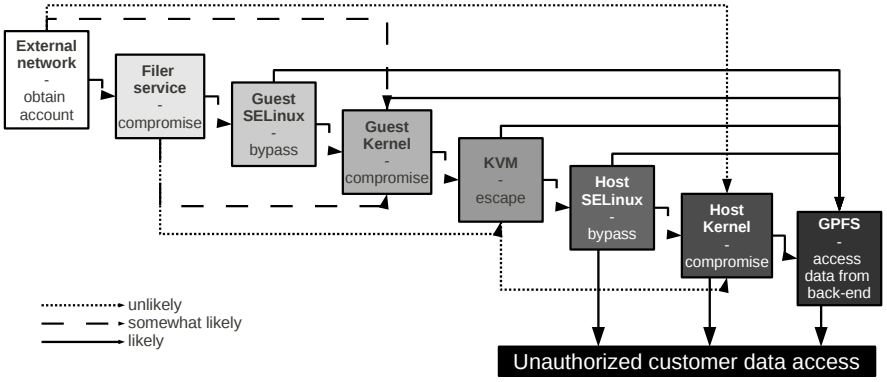


Figure 6.3. Attack graph for the VMT architecture.

Attacks against the VMT architecture.

VM escapes. Although virtual machines are often marketed as the ultimate security isolation tool, it has been shown [Kor09; Orm07] that many existing hypervisors contain vulnerabilities that can be exploited to escape from a guest machine to the host. We assume these attacks are "somewhat likely".

Filer service compromise: NFS daemon and SELinux. Apart from the helper daemons, which represent a small part of the overall code (e.g., `rpc.mountd`, `rpc.statd`, `portmapd`), most of the `nfsd` code is in kernel-space. This means it is not possible to restrict the privileges of this code with a MAC mechanism in the sense that a vulnerability in this code might directly lead to arbitrary code execution in kernel mode. The authors of [Bla+09] tried to implement such a protection within the kernel but this approach cannot guarantee sufficient isolation of kernel code simply because an attacker with `ring 0` privileges can disable SELinux. We categorize this attack as "somewhat likely."

Attacks against the OSMT architecture.

Container escapes. As mentioned in 6.2.3.4, we have implemented what we refer to as containers using a chroot environment. As is widely known, a chroot environment does not effectively prevent an attacker from escaping from the environment and provides limited isolation. For completeness, we include a container-protection layer which corresponds to the chroot environment (without SELinux) in Figure 6.4, and marked it as "likely" to be defeated. However, containers such as LXC do implement better containment using the *cgroups* feature of Linux. While these technologies have a clean and simple design, it is still likely that some vulnerabilities allowing escapes can be found,

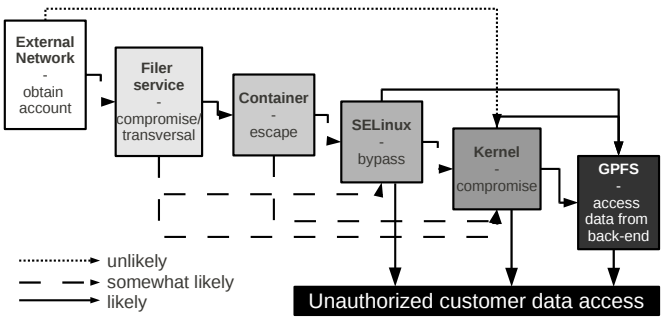


Figure 6.4. Attack graph for the OSMT architecture.

especially because they are very recent (one such current concern regards containers mounting a `/proc` pseudo-filesystem).

6.2.5 Conclusion and Applicability of Attack Surface Reduction

A high-level comparison of Figures 6.3 and 6.4 shows that the VMT architecture has many more layers and could lead to the conclusion that the VMT approach provides better security. However, we also have to take into account the various attacks: most notably, it is possible that an attacker uses the internal network interface effectively for customer data access, and that this network interface is accessible from within the guest VMs (which is required, because the distributed filesystem service runs in the guest). The possibility of this attack renders other layers of protection due to VM isolation much less useful in the VMT architecture.

In other words, a likely chain of compromises that can occur for each scenario is

- for VMT:

    1. attacker compromises filer service, obtaining local unprivileged access,[2]
    2. attacker exploits a local kernel privilege escalation vulnerability that can be exploited within the MAC security context of the filer service,
    3. attacker accesses files of a customer through the distributed filesystem (assuming no authentication or authorization of nodes and no access control on blocks).

[2] In the case of a kernel NFS daemon, it is possible that the attacker directly obtains `ring 0` privileges and can therefore skip the next step, however this is less likely.

- for OSMT:

    1. attacker compromises filer service, obtaining local unprivileged access,
    2. attacker exploits a local kernel privilege escalation vulnerability that can be exploited within the MAC security context of the filer service,
    3. attacker accesses files of a local co-tenant, or through the distributed filesystem for other customers.

First, although it is expected that hypervisor-level multi-tenancy can, in general, be a better security design than OS-level multi-tenancy, we have seen in the case of a filesystem storage cloud and under our assumptions (i.e., that in a distributed filesystem, each individual node is trusted), no solution was clearly more secure than the other. Both solutions could be used to achieve an acceptable level of security.

Second, with regards to this thesis, we have seen the importance of kernel self-protection in practical uses. Indeed, in both cases, the most likely chain of compromises includes making use of a kernel privilege escalation vulnerability. This thesis shows that the likelihood of such an even can be reduced by making use of either of the two kernel attack surface reduction techniques presented (tailoring and trimming).

7 Conclusion

> “It is a far, far better thing that I do, than I have ever done; it is a far, far better rest that I go to than I have ever known.”
>
> Charles Dickens, *A tale of two cities*

This thesis demonstrates the feasibility and the effectiveness of reducing kernel attack surface to improve kernel protection. This chapter concludes this thesis by summing up its results, contributions, and opens up towards areas of future work.

7.1 In a nutshell

Commodity OS kernels includes many exotic features, which are seldom used. Attackers tend to target such features, at least as often as other features. In fact, sometimes attackers are even more likely to target them, because of their ripeness for vulnerabilities induced by a lack of testing in such code paths. Past research mainly focused on either making kernels smaller and more secure by designing new operating systems from scratch, finding vulnerabilities using static and dynamic analysis, or mitigating the impact of vulnerabilities by making their exploitation more difficult.

Approach. In this thesis, I focus on reducing kernel attack surface. To achieve this goal, I propose two methods: the first makes uses of the built-in configurability of the kernel to disable features, while the second dynamically instruments a carefully selected set of kernel functions to detect, or even prevent, the use of such unnecessary features.

To enable understanding and evaluating the benefits of such approaches, I also define and measure how such seldom-used features are increasing the attack surface.

Research Questions. In the context of this thesis, I explored two main research questions:

Q1: *Is it possible to precisely define the kernel attack surface? Can it be measured?*

In Chapter 3, I defined the kernel attack surface based on call graph reachability and the assumption of a security model. I also defined two metrics two measure the size of these attack surfaces, with proofs based on properties that such metrics should intuitively fulfill. Then, in Chapter 4 and Chapter 5, I used these metrics to measure the attack surface of distribution kernels.

Q2: *Can we develop kernel protection mechanisms whose attack surface reduction is quantifiable? To what extent can these mechanisms be applied to commodity OSes in practice?*

This thesis demonstrates that tailoring and trimming are both kernel protection mechanism whose attack surface reduction is quantifiable. I have also shown that these can be used with ease on Linux. I discussed the limitations of each of these approaches, mainly recompilation for tailoring and false positives for trimming. Finally, both approaches are effective in preventing vulnerabilities present in the kernel sources from affecting the security of running systems.

Summary. Making use of kernel attack surface metrics is a useful way to guide the design and implementation of kernel attack surface methods. By keeping in mind the goal of reducing kernel attack surface, we designed tailoring and trimming, which detect or prevent the exploitation of a large number of kernel vulnerabilities.

7.2 Contributions

This thesis makes three major contributions.

The first contribution deals with methods for quantifying kernel attack surface. There, I define what the kernel attack surface is, and propose a definition as well as multiple methods of quantifying it. This is necessary because the current state of the art usually consists often in counting source lines of code (which unfortunately also includes code

unreachable to attackers), or, anecdotal evidence on past vulnerabilities that would be prevented. Hence, this method enables to develop novel kernel protection protection and compare them with improved scientific rigor.

The second contribution, compile-time kernel tailoring, is a kernel protection technique that aims to reduce attack surface. We show it is possible to automatically generate a set of kernel configuration option for a given workload (via kernel tracing), and that the resulting kernel can be proven to provide a smaller attack surface.

The third contribution, run-time kernel trimming, also aims to reduce attack surface, but at a finer granularity and without requiring to recompile the kernel. By selectively instrumenting kernel functions, it also incurs minimal performance overhead. Evaluation results show that the attack surface can be reduced significantly more than via tailoring.

Summary. Together, these three contributions demonstrate that kernel attack surface reduction is a viable and effective approach to improving kernel protection in commodity OSes such as Linux. The attack surface metrics lay the foundation for this work by enabling objective comparisons, while tailoring and trimming show the practical feasibility, effectiveness as well as limitations of such approaches.

7.3 Future Work

This work opens many areas of future work. We have seen attack surface reduction is effective even with techniques of moderate complexity (trimming and tailoring), hence, I would expect that improved techniques can reduce the attack surface even further. For instance, basic-block level tracing can provide finer-granularity information on the necessary functionality, when compared to the function-level tracing I have performed, although this brings a new set of challenges, especially in terms of quantification.

Similarly, a case can be made for transposing attack surface reduction to user space (e.g., network daemons such as a web server). As an example, the OpenSSL library, which is linked in many critical internet-facing daemons, consists of about 300,000 lines of code in its source tree. It would be beneficial to quantify how much of this code is actually reachable to a remote attacker, how much of this code is necessary for the typical operation of such a daemon. In turn, mechanisms comparable to kernel tailoring or trimming could be adapted. For instance, such an approach would have proactively prevented the now infamous “heartbleed” vulnerability [Adv14], since the OpenSSL

heartbeat functionality was reachable to remote attackers despite being unnecessary in a large number of use cases. In fact, as a response to the vulnerability, some OpenSSL distributors decided to turn of the feature via a compile-time switch [Raa14].

Finally, there is an exciting area of work relating to run-time kernel trimming enforcement. Recently, we were able to build OS kernels with both hardened and "vanilla" kernel code, while sharing all kernel data. This opens up the possibility of having a "split kernel", which can run hardened kernel code only when necessary: for instance some processes can be run under hardened kernel code while others use a vanilla kernel for their system calls. This results in best-of-both-worlds OS kernels that can include hardening features without sacrificing any performance in practice.

A

Proofs

The following is a proof of Proposition 1.

Proof (AS1 and AS2 are attack surface metrics). $AS1_{\mu}$ satisfies Definition 5 through Lemma 1, as adding new functions to the sum results in a larger attack surface measurement (since μ has non-negative values).

For $AS2_{\mu}$, the non-negativity is a known result of algebraic graph theory [Big74]: the Laplacian matrix of a simple graph is symmetric real and all eigenvalues are non-negative, hence, the quadratic form associated with the Laplacian ($x \mapsto x^T L(G)x$) can only take non-negative values.

Before proving that $AS2_{\mu}$ satisfies the second property in Lemma 1, we explicit the rationale behind chosing this metric. The metric contains a quadratic term that accounts for the relative "complexity" of a function in comparison to its callers and callees: if a function is calling (or is called by) a more complex function, its relative contribution to the attack surface should increase and vice versa. For instance, this can be written for a function n, called by functions m, m' and calling function m'', $\kappa(n)$ denoting the relative complexity of function n:

$$\kappa(n) = \mu(n)\left[(\mu(n) - \mu(m)) + (\mu(n) - \mu(m')) + (\mu(n) - \mu(m''))\right]$$

Generalizing to any function:

$$\kappa(n) = \mu(n)\left[deg(n)\mu(n) - \sum_{(i,n)\in\widetilde{C_{AS}}} \mu(i)\right]$$

Which, after summing over all functions, corresponds to ${\mu_{AS}}^T L(\widetilde{G_{AS}})\mu_{AS}$.

Let's now prove that adding a new node to an existing graph can only increase this quadratic term. Without loss of generality, we assume we starting with function set $F = [\![1 \dots N-1]\!]$ and affect N to the newly added function. This function is either called or is calling m functions in $I \subseteq F$ with $deg(N) = m < N$. We denote by κ the old relative complexity and κ' the new (after the addition of N to the graph), and deg corresponds as well to the old degree of a node, unless it is $deg(N)$. Then:

$$\forall i \in F, \kappa'(i) - \kappa(i) = \mu(i)\left[\mu(i) - \mu(N)\right]$$

Therefore:

$$\begin{aligned}
\kappa(N) + \sum_{i \in F} \left(\kappa'(i) - \kappa(i)\right) &= \mu(N)\left[deg(N)\mu(N) - \sum_{i \in I} \mu(i)\right] \\
&\quad + \sum_{i \in I} \mu(i)\left[\mu(i) - \mu(N)\right] \\
&= \sum_{i \in I} \mu(i)^2 + m\mu(N)^2 - 2\mu(N)\sum_{i \in I} \mu(i) \\
&= \sum_{i \in I} \left(\mu(i) - \mu(N)\right)^2 \geq 0
\end{aligned}$$

Hence, adding new functions can only increase the attack surface measurement $AS2_\mu$. □

Bibliography

[Acc+86] Mike Accetta, Robert Baron, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. "MACH: A New Kernel Foundation for UNIX Development". In: *Proceedings of the USENIX Summer Conference*. 1986, pages 93–113.

[Adv14] OpenSSL Security Advisory. *TLS Heatbeat read overrun (CVE-2014-0160)*. `https://www.openssl.org/news/secadv_20140407.txt`. 2014.

[And72] James P Anderson. *Computer Security Technology Planning Study. Volume 2*. Technical report. DTIC Document, 1972.

[AR00] A. Acharya and M. Raje. "MAPbox: Using parameterized behavior classes to confine untrusted applications". In: *Proceedings of the 9th conference on USENIX Security Symposium-Volume 9*. 2000, page 1.

[Bel06] S.M. Bellovin. "On the Brittleness of Software and the Infeasibility of Security Metrics". In: *Security Privacy, IEEE* 4.4 (2006), page 96. ISSN: 1540-7993.

[Ben+12] Boldizsár Bencsáth, Gábor Pék, Levente Buttyán, and Márk Félegyházi. "Duqu: Analysis, detection, and lessons learned". In: *ACM European Workshop on System Security (EuroSec)*. Volume 2012. 2012.

[Ber+94] Brian N. Bershad, Craig Chambers, Susan J. Eggers, Chris Maeda, Dylan McNamee, Przemyslaw Pardyak, Stefan Savage, and Emin Gun Sirer. "SPIN — An Extensible Microkernel for Application-specific Operating System Services". In: *ACM SIGOPS European Workshop*. 1994, pages 68–71.

[Big74] N. Biggs. *Algebraic Graph Theory*. 1974.

[Bla+09] M. Blanc, K. Guerin, J.F. Lalande, and V. Le Port. “Mandatory Access Control implantation against potential NFS vulnerabilities”. In: *International Symposium on Collaborative Technologies and Systems* (2009).

[BN01] Mikhail Belkin and Partha Niyogi. “Laplacian Eigenmaps and Spectral Techniques for Embedding and Clustering”. In: *Advances in Neural Information Processing Systems 14*. 2001, pages 585–591.

[BP84] Victor R. Basili and Barry T. Perricone. “Software errors and complexity: an empirical investigation”. In: *Commun. ACM* 27.1 (1984), pages 42–52. ISSN: 0001-0782.

[Bra+12] Sergey Bratus, Julian Bangert, Alexandar Gabrovsky, Anna Shubina, Daniel Bilar, and Michael E Locasto. “Composition Patterns of Hacking”. In: *Proceedings of the First International Workshop Cyber Patterns*. 2012, pages 80–85.

[BWZ10] Silas Boyd-Wickizer and Nickolai Zeldovich. “Tolerating malicious device drivers in Linux”. In: *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*. USENIXATC’10. 2010, pages 9–9.

[BZ06] *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys ’06)*. (Leuven, Belgium). 2006. ISBN: 1-59593-322-0.

[Cai+11] H. Cai, B. Reinwald, N. Wang, and C.J. Guo. “SaaS Multi-Tenancy: Framework, Technology, and Case Study”. In: *International Journal of Cloud Applications and Computing (IJCAC)* 1.1 (2011). ISSN: 2156-1834.

[Can+12] Davide Canali, Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. “A quantitative study of accuracy in system call-based malware detection”. In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ISSTA 2012. 2012, pages 122–132. ISBN: 978-1-4503-1454-1.

[Car+09] Martim Carbone, Weidong Cui, Long Lu, Wenke Lee, Marcus Peinado, and Xuxian Jiang. “Mapping kernel objects to enable systematic integrity checking”. In: *Proceedings of the 16th ACM conference on Computer and communications security*. CCS ’09. 2009, pages 555–565. ISBN: 978-1-60558-894-0.

[Cas+09] 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 22nd ACM Symposium on Operating Systems Principles (SOSP ’09)*. (Big Sky, Montana, USA). 2009, pages 45–58. ISBN: 978-1-60558-752-3.

[CCZ06] Huseyin Cavusoglu, Hasan Cavusoglu, and Jun Zhang. “Economics of Security Patch Management.” In: *WEIS*. 2006.

[Cha+05] Dominique Chanet, Bjorn De Sutter, Bruno De Bus, Ludo Van Put, and Koen De Bosschere. “System-wide Compaction and Specialization of the Linux Kernel”. In: *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES ’05)*. 2005, pages 95–104. ISBN: 1-59593-018-3.

[Che+11] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. “Linux Kernel Vulnerabilities: State-of-the-art Defenses and Open Problems”. In: *Proceedings of the Second Asia-Pacific Workshop on Systems*. APSys ’11. 2011, 5:1–5:5. ISBN: 978-1-4503-1179-3.

[Che78] Edward T. Chen. “Program complexity and programmer productivity”. In: *Software Engineering, IEEE Transactions on* 3 (1978), pages 187–194.

[Cho+01] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. “An empirical study of operating systems errors”. In: *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP ’01)*. (Banff, Alberta, Canada). 2001, pages 73–88. ISBN: 1-58113-389-8.

[Cho+05] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. “Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation”. In: *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*. SSYM’05. 2005, pages 22–22.

[CKC11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. “S2E: A Platform for In-vivo Multi-path Analysis of Software Systems”. In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. 2011, pages 265–278. ISBN: 978-1-4503-0266-1.

[CM04] Brian Chess and Gary McGraw. “Static analysis for security”. In: *Security & Privacy, IEEE* 2.6 (2004), pages 76–79.

[Cok] Russell Coker. *Bonnie++. Benchmark suite for hard drive and file system performance*.

[Coo10] Kees Cook. *Yama LSM*. 2010.

[Coo11] Kees Cook. *Kernel Exploitation Via Uninitialized Stack*. Defcon 19. 2011.

[Cri+07] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. “Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems”. In: *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP ’07)*. (Stevenson, WA, USA). 2007, pages 351–366. ISBN: 978-1-59593-591-5.

[CV65] F. J. Corbató and V. A. Vyssotsky. “Introduction and overview of the multics system”. In: *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*. AFIPS ’65 (Fall, part I). 1965, pages 185–196.

[Dan+] A. Dan, A. Mohindra, R. Ramaswami, and D. Sitaram. *Chakravyuha: A sandbox operating system for the controlled execution of alien code*. Technical report. IBM TJ Watson research center, 1997.

[Dec14] Sergej Dechand. *Attack Surface Measurements for State-of-the-Art Kernel Hardening Mechanisms*. Master's thesis, TU Braunschweig. 2014.

[Die+12] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. "A Robust Approach for Variability Extraction from the Linux Build System". In: *Proceedings of the 16th Software Product Line Conference (SPLC '12)*. (Salvador, Brazil, Sept. 2–7, 2012). 2012, pages 21–30. ISBN: 978-1-4503-1094-9.

[DW10] Sarah M. Diesburg and An-I Andy Wang. "A Survey of Confidential Data Storage and Deletion Methods". In: *ACM Computing Surveys* 43.1 (2010).

[EKO95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. "Exokernel: An Operating System Architecture for Application-Level Resource Management". In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM SIGOPS Operating Systems Review. 1995, pages 251–266.

[Eng+00] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. "Checking system rules using system-specific, programmer-written compiler extensions". In: *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*. OSDI'00. 2000, pages 1–1.

[Ess11] Stefan Esser. *iOS Kernel Exploitation*. `http://media.blackhat.com/bh-us-11/Esser/BH_US_11_Esser_Exploiting_The_iOS_Kernel_Slides.pdf`. 2011.

[Eva12] Chris Evans. *Pwnium 3 and Pwn2Own Results*. `http://blog.chromium.org/2013/03/pwnium-3-and-pwn2own-results.html`. 2012.

[Fen+03] Henry Hanping Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. "Anomaly Detection Using Call Stack Information". In: *Proceedings of the 2003 IEEE Symposium on Security and Privacy*. SP '03. 2003, pages 62–. ISBN: 0-7695-1940-7.

[FHS08] Stephanie Forrest, Steven Hofmeyr, and Anil Somayaji. "The Evolution of System-Call Monitoring". In: *Proceedings of the 2008 Annual Computer Security Applications Conference*. ACSAC '08. 2008, pages 418–430. ISBN: 978-0-7695-3447-3.

[For+96] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. "A Sense of Self for Unix Processes". In: *Proceedings of the 1996 IEEE Symposium on Security and Privacy*. SP '96. 1996, pages 120–. ISBN: 0-8186-7417-2.

[Fos+03] Jeffrey S Foster, R Johnson, J Kodumal, T Terauchi, U Shankar, K Talwar, D Wagner, A Aiken, M Elsman, and C Harrelson. *Cqual: A tool for adding type qualifiers to C*. 2003.

[Fra] *Frama-C: A framework for static analysis of C programs*.

[Gar03] T. Garfinkel. "Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools". In: *NDSS*. 2003.

[Goh] Andreas Gohr. *DokuWiki*.

[Gol+96] I. Goldberg, D. Wagner, R. Thomas, and E. A Brewer. "A secure environment for untrusted helper applications confining the Wily Hacker". In: *Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography-Volume 6*. 1996, page 1.

[Goo09] Google. *Seccomp Sandbox for Linux*. 2009.

[Gos00] James Gosling. *The Java language specification*. 2000.

[GRS04] Debin Gao, Michael K. Reiter, and Dawn Song. "Gray-box extraction of execution graphs for anomaly detection". In: *Proceedings of the 11th ACM conference on Computer and communications security*. CCS '04. 2004, pages 318–329. ISBN: 1-58113-961-6.

[Hal+] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. *Dowser: A Guided Fuzzer for Finding Buffer Overflow Vulnerabilities*. Usenix Login, Vol. 38, Number 6. `https://www.usenix.org/publications/login/december-2013-volume-38-number-6/dowser-guided-fuzzer-finding-buffer-overflow`.

[Har+05] H. Hartig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. "The Nizza secure-system architecture". In: *Collaborative Computing: Networking, Applications and Worksharing, 2005 International Conference on*. 2005, 10 pp.

[Har85] Norman Hardy. "KeyKOS Architecture". In: *ACM SIGOPS Operating Systems Review* 19.4 (1985), pages 8–25. ISSN: 0163-5980.

[Har88] N. Hardy. "The Confused Deputy". In: *ACM SIGOPS Operating Systems Review* 22.4 (1988). ISSN: 0163-5980.

[Hee11] Sean Heelan. "Vulnerability Detection Systems: Think Cyborg, Not Robot". In: *IEEE Security and Privacy* 9.3 (2011), pages 74–77. ISSN: 1540-7993.

[Her+06a] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. "Construction of a Highly Dependable Operating System". In: *Proceedings of the Sixth European Dependable Computing Conference*. EDCC '06. 2006, pages 3–12. ISBN: 0-7695-2648-9.

[Her+06b] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. “MINIX 3: a highly reliable, self-repairing operating system”. In: *SIGOPS Oper. Syst. Rev.* 40.3 (2006), pages 80–89. ISSN: 0163-5980.

[Her+08] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. “Countering IPC Threats in Multiserver Operating Systems (A Fundamental Requirement for Dependability)”. In: *Proceedings of the 2008 14th IEEE Pacific Rim International Symposium on Dependable Computing*. PRDC ’08. 2008, pages 112–121. ISBN: 978-0-7695-3448-0.

[HFS98] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. “Intrusion detection using sequences of system calls”. In: *J. Comput. Secur.* 6.3 (1998), pages 151–180. ISSN: 0926-227X.

[HHT04] Toshiharu Harada, Takashi Horie, and Kazuo Tanaka. “Task Oriented Management Obviates Your Onus on Linux”. In: *Proceedings of the Japan Linux Conference* (2004). ISSN: 1348-7868.

[HL10] Gernot Heiser and Ben Leslie. “The OKL4 microvisor: convergence point of microkernels and hypervisors”. In: *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*. APSys ’10. 2010, pages 19–24. ISBN: 978-1-4503-0195-4.

[Hoh+04] Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro. “Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors”. In: *Proceedings of the 11th workshop on ACM SIGOPS European workshop*. EW 11. 2004.

[HPW05] M. Howard, J. Pincus, and J. Wing. “Measuring Relative Attack Surfaces”. In: *Computer Security in the 21st Century* (2005), pages 109–137.

[IFF96] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. “Lua—an Extensible Extension Language”. In: *Softw. Pract. Exper.* 26.6 (1996), pages 635–652. ISSN: 0038-0644.

[Iso] *ISO 17799:2005*. `http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=39612`.

[Jaq07] A. Jaquith. *Security Metrics: Replacing Fear, Uncertainty, and Doubt*. 2007.

[JEZ04] Trent Jaeger, Antony Edwards, and Xiaolan Zhang. “Consistency analysis of authorization hook placement in the Linux security modules framework”. In: *ACM Trans. Inf. Syst. Secur.* 7.2 (2004), pages 175–205. ISSN: 1094-9224.

[JW04] Rob Johnson and David Wagner. “Finding user/kernel pointer bugs with type inference”. In: *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*. SSYM’04. 2004, pages 9–9.

[KDK14] Anil Kurmus, Sergej Dechand, and Rüdiger Kapitza. “Quantifiable Run-time Kernel Attack Surface Reduction”. In: *Proceedings of the 10th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA’14)*. 2014 (to appear).

[Kiv+07] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. “kvm: the Linux Virtual Machine Monitor”. In: *Proc. Linux Symposium*. Volume 1. 2007.

[Kle+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. “seL4: formal verification of an OS kernel”. In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP ’09)*. (Big Sky, Montana, USA). 2009, pages 207–220. ISBN: 978-1-60558-752-3.

[Ko+00] Calvin Ko, Timothy Fraser, Lee Badger, and Douglas Kilpatrick. “Detecting and countering system intrusions using software wrappers”. In: *Proceedings of the 9th conference on USENIX Security Symposium - Volume 9*. SSYM’00. 2000, pages 11–11.

[Kor09] K. Kortchinsky. *Cloudburst – Hacking 3D and Breaking out of VMware*. 2009.

[KPK12] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. “kGuard: lightweight kernel protection against return-to-user attacks”. In: *Proceedings of the 21st USENIX conference on Security symposium*. Security’12. 2012, pages 39–39.

[Kru+03] Christopher Kruegel, Darren Mutz, Fredrik Valeur, and Giovanni Vigna. “On the detection of anomalous system call arguments”. In: *Computer Security–ESORICS 2003* (2003), pages 326–343.

[Kru+05] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. “Automating mimicry attacks using static binary analysis”. In: *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*. SSYM’05. 2005, pages 11–11.

[KSK11] Anil Kurmus, Alessandro Sorniotti, and Rüdiger Kapitza. “Attack Surface Reduction For Commodity OS Kernels”. In: *Proceedings of the Fourth European Workshop on System Security*. 2011.

[Kur+11] Anil Kurmus, Moitrayee Gupta, Roman Pletka, Christian Cachin, and Robert Haas. “A Comparison of Secure Multi-Tenancy Architectures for Filesystem Storage Clouds”. In: *Middleware*. 2011, pages 471–490.

[Kur+13] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. "Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring". In: *Proceedings of the 20th Network and Distributed System Security Symposium*. 2013.

[KW00] P.H. Kamp and R.N.M. Watson. "Jails: Confining the omnipotent root". In: *Proc. International System Administration and Network Engineering*. 2000.

[LA04] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO '04. 2004, pages 75–. ISBN: 0-7695-2102-9.

[LAK09] Andrew Lenharth, Vikram S. Adve, and Samuel T. King. "Recovery domains: an organizing principle for recoverable operating systems". In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*. (Washington, DC, USA). 2009, pages 49–60. ISBN: 978-1-60558-406-5.

[Lam73] Butler W. Lampson. "A note on the confinement problem". In: *Commun. ACM* 16.10 (1973), pages 613–615. ISSN: 0001-0782.

[Lee+04] C.T. Lee, J.M. Lin, Z.W. Hong, and W.T. Lee. "An Application-Oriented Linux Kernel Customization for Embedded Systems". In: *Journal of information science and engineering* 20.6 (2004), pages 1093–1108. ISSN: 1016-2364.

[Lie95] Jochen Liedtke. "On μ-Kernel Construction". In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM SIGOPS Operating Systems Review. 1995.

[LMU05] Julia L. Lawall, Gilles Muller, and Richard Urunuela. "Tarantula: Killing Driver Bugs Before They Hatch". In: *Proceedings of the 4th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '05)*. 2005, pages 13–18.

[Los+98] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrell. "The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments". In: *In Proceedings of the 21st National Information Systems Security Conference*. 1998, pages 303–314.

[Lu+13] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. "A Study of Linux File System Evolution". In: *Proceedings of the 11th Conference on File and Storage Technologies (FAST '13)*. 2013.

[Lxc] *lxc: Linux Containers*. `http://lxc.sourceforge.net`.

[Ma+12] Weiqin Ma, Pu Duan, Sanmin Liu, Guofei Gu, and Jyh-Charn Liu. “Shadow attacks: automatically evading system-call-behavior based malware detection”. In: *J. Comput. Virol.* 8.1-2 (2012), pages 1–13. ISSN: 1772-9890.

[Mad+10] A. Madhavapeddy, R. Mortier, R. Sohan, T. Gazagnaire, S. Hand, T. Deegan, D. McAuley, and J. Crowcroft. “Turning Down the LAMP: Software Specialisation for the Cloud”. In: *Proceedings of the 2nd USENIX Conference on hot topics in cloud computing (HOTCLOUD’10)*. 2010, pages 11–11.

[Mao+11] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. “Software fault isolation with API integrity and multi-principal modules”. In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP ’11)*. (Cascais, Portugal). 2011, pages 115–128. ISBN: 978-1-4503-0977-6.

[McC+08] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. “Flicker: an execution infrastructure for tcb minimization”. In: *SIGOPS Oper. Syst. Rev.* 42.4 (2008), pages 315–328. ISSN: 0163-5980.

[McC+10] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. “TrustVisor: Efficient TCB Reduction and Attestation”. In: *Security and Privacy (SP), 2010 IEEE Symposium on*. 2010, pages 143 –158.

[McC04] B. McCarty. *SELinux: NSA’s Open Source Security Enhanced Linux*. 2004.

[McC76] T.J. McCabe. “A Complexity Measure”. In: *Software Engineering, IEEE Transactions on* SE-2.4 (1976), pages 308 –320. ISSN: 0098-5589.

[MFS90] Barton P. Miller, Louis Fredriksen, and Bryan So. “An empirical study of the reliability of UNIX utilities”. In: *Communications of the ACM* 33.12 (1990), pages 32–44. ISSN: 0001-0782.

[MJ93] Steven McCanne and Van Jacobson. “The BSD packet filter: a new architecture for user-level packet capture”. In: *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*. USENIX’93. 1993, pages 2–2.

[MJ98] David Mosberger and Tai Jin. “httperf. A tool for measuring web server performance”. In: *SIGMETRICS Performance Evaluation Review* 26.3 (1998), pages 31–37. ISSN: 0163-5999.

[MMC06] Frank Mayer, Karl MacMillan, and David Caplan. *SELinux By Example: Using Security Enhanced Linux*. 2006, page 456. ISBN: 978-0131963696.

[MMH08] Derek Gordon Murray, Grzegorz Milos, and Steven Hand. “Improving Xen security through disaggregation”. In: *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. VEE ’08. 2008, pages 151–160. ISBN: 978-1-59593-796-4.

[Mut+06] Darren Mutz, Fredrik Valeur, Giovanni Vigna, and Christopher Kruegel. "Anomalous system call detection". In: *ACM Trans. Inf. Syst. Secur.* 9.1 (2006), pages 61–93. ISSN: 1094-9224.

[MW11] P.K. Manadhata and J.M. Wing. "An Attack Surface Metric". In: *Software Engineering, IEEE Transactions on* 37.3 (2011), pages 371 –386. ISSN: 0098-5589.

[Ncc] *ncc: The new generation C compiler*.

[Oeh+08] S. Oehme, J. Deicke, J.P. Akelbein, R. Sahlberg, A. Tridgell, and RL Haskin. "IBM Scale out File Services: Reinventing network-attached storage". In: *IBM Journal of Research and Development* 52.4.5 (2008).

[Orm07] T. Ormandy. "An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments". In: *Proc. CanSecWest Applied Security Conference*. 2007.

[Osm+02] S. Osman, D. Subhraveti, G. Su, and J. Nieh. "The Design and Implementation of Zap: A System for Migrating Computing Environments". In: *ACM SIGOPS Operating Systems Review* 36.SI (2002). ISSN: 0163-5980.

[OWB04] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. "Where the bugs are". In: *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*. ISSTA '04. 2004, pages 86–96. ISBN: 1-58113-820-2.

[Pad+08] Yoann Padioleau, Julia L. Lawall, Gilles Muller, and René Rydhof Hansen. "Documenting and Automating Collateral Evolutions in Linux Device Drivers". In: *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (EuroSys '08)*. (Glasgow, Scotland). 2008.

[Pal+11] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia L. Lawall, and Gilles Muller. "Faults in Linux: Ten years later". In: *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*. 2011, pages 305–318.

[Php] *phpBB. Free and Open Source Forum Software*.

[PLM06] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. "Understanding Collateral Evolution in Linux Device Drivers". In: *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06)*. (Leuven, Belgium). 2006, pages 59–71. ISBN: 1-59593-322-0.

[Pro03] Niels Provos. "Improving host security with system call policies". In: *Proceedings of the 12th Conference on USENIX Security Symposium (SSYM '03)*. Volume 12. 2003, pages 18–18.

[PT04] D. Price and A. Tucker. “Solaris Zones: Operating System Support for Consolidating Commercial Workloads”. In: *Proc. System administration*. 2004.

[Raa14] Theo De Raadt. *Disable Segglemann’s RFC520 hearbeat.* `http://www.openbsd.org/cgi-bin/cvsweb/src/lib/libssl/ssl/Makefile?rev=1.29;content-type=text%2Fx-cvsweb-markup`. 2014.

[RKS12] Matthew J Renzelmann, Asim Kadav, and Michael M Swift. “Symdrive: Testing Drivers Without Devices”. In: OSDI’12. 2012.

[Ros] D. Rosenberg. *CVE-2010-3904 exploit*.

[Sai+04] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. “Design and Implementation of a TCG-based Integrity Measurement Architecture”. In: *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*. 2004, page 16.

[Sal74] Jerome H. Saltzer. “Protection and the control of information sharing in Multics”. In: *Communications of the ACM* 17 (7 1974), pages 388–402. ISSN: 0001-0782.

[SB10] Steven She and Thorsten Berger. *Formal Semantics of the Kconfig Language*. Technical Note. University of Waterloo, 2010.

[SBE11] S. Stolfo, S.M. Bellovin, and D. Evans. “Measuring Security”. In: *Security Privacy, IEEE* 9.3 (2011), pages 60 –65. ISSN: 1540-7993.

[Sch04] S.E. Schechter. “Computer Security Strength & Risk: A Quantitative Approach”. PhD thesis. Harvard University Cambridge, Massachusetts, 2004.

[Sch99] B. Schneier. “Attack trees”. In: *Dr. Dobbs journal* 12 (1999).

[Scu10] D. Sculley. “Web-scale k-means clustering”. In: *Proceedings of the 19th international conference on World wide web*. WWW ’10. 2010, pages 1177–1178. ISBN: 978-1-60558-799-8.

[Ses+07] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. “SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes”. In: *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. SOSP ’07. 2007, pages 335–350. ISBN: 978-1-59593-591-5.

[SH02] F. Schmuck and R. Haskin. “GPFS: A Shared-disk File System For Large Computing Clusters”. In: *Proc. File and Storage Technologies*. 2002.

[She+85] Vincent Y. Shen, Tze-Jie Yu, Stephen M. Thebaut, and Lorri R. Paulsen. “Identifying Error-Prone Software An Empirical Study”. In: *IEEE Trans. Softw. Eng.* 11.4 (1985), pages 317–324. ISSN: 0098-5589.

[Sin+06] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. "Reducing TCB complexity for security-sensitive applications: three case studies". In: *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06)*. (Leuven, Belgium). 2006, pages 161–174. ISBN: 1-59593-322-0.

[Sin+10] Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder-Preikschat. "Efficient Extraction and Analysis of Preprocessor-Based Variability". In: *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE '10)*. (Eindhoven, The Netherlands). 2010, pages 33–42. ISBN: 978-1-4503-0154-1.

[Son] *IBM Scale Out Network Attached Storage*. `http://www-03.ibm.com/systems/storage/network/sonas/`.

[Sos] *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*. (Big Sky, Montana, USA). 2009. ISBN: 978-1-60558-752-3.

[Spe] Brad Spengler. *Enlightenment exploit framework*. `https://grsecurity.net/~spender/exploits/enlightenment.tgz`.

[St] Brad Spengler and PaX team. *grsecurity kernel patches*.

[SVS01] Stephen Smalley, Chris Vance, and Wayne Salamon. *Implementing SELinux as a Linux security module*. Technical report. NAI Labs Report, 2001.

[SW08] Yonghee Shin and Laurie Williams. "Is complexity really the enemy of software security?" In: *Proceedings of the 4th ACM workshop on Quality of protection*. QoP '08. 2008, pages 47–50. ISBN: 978-1-60558-321-1.

[Swi+02] Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. "Nooks: an architecture for reliable device drivers". In: *Proceedings of the 9th ACM SIGOPS European Workshop "Beyond the PC: New Challenges for the Operating System"*. (Saint-Emilion, France). 2002, pages 102–107.

[SWZ05] Gopalan Sivathanu, Charles P. Wright, and Erez Zadok. "Ensuring Data Integrity in Storage: Techniques and Applications". In: *Proc. Storage Security and Survivability*. 2005.

[Tar+11] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. "Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem". In: *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2011 (EuroSys '11)*. (Salzburg, Austria). 2011, pages 47–60. ISBN: 978-1-4503-0634-8.

[TMK03] Kymie M. C. Tan, John McHugh, and Kevin S. Killourhy. "Hiding Intrusions: From the Abnormal to the Normal and Beyond". In: *Revised Papers from the 5th International Workshop on Information Hiding*. IH '02. 2003, pages 1–17. ISBN: 3-540-00421-1.

[Tom] *TOMOYO*. `http://tomoyo.sourceforge.jp/`.

[Tor03] Linus Torvalds. *Sparse - a Semantic Parser for C*. 2003.

[Tor12] Linus Torvalds. *[RFC] Simplifying kernel configuration for distro issues*. E-mail to the Linux kernel mailing list `https://lists.debian.org/debian-kernel/2012/07/msg00362.html`. 2012.

[Tra+04] A. Traeger, A. Rai, C.P. Wright, and E. Zadok. *NFS File Handle Security*. Technical report. Computer Science Department, Stony Brook University, 2004.

[Vee+12] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. "Memory Errors: The Past, the Present, and the Future". In: *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses*. RAID'12. 2012, pages 86–106. ISBN: 978-3-642-33337-8.

[Wah+93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. "Efficient Software-Based Fault Isolation". In: *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*. (Asheville, NC, USA). 1993, pages 203–216. ISBN: 0-89791-632-8.

[Wat+03] Robert Watson, Brian Feldman, Adam Migus, and Chris Vance. "Design and Implementation of the TrustedBSD MAC Framework". In: *3rd DARPA Information Survivability Conference and Exposition (DISCEX-III 2003), 22-24 April 2003, Washington, DC, USA*. 2003, pages 38–49.

[Wat+12] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. "A taste of Capsicum: practical capabilities for UNIX". In: *Commun. ACM* 55.3 (2012), pages 97–104. ISSN: 0001-0782.

[Wat07] Robert N. M. Watson. "Exploiting concurrency vulnerabilities in system call wrappers". In: *Proceedings of the first USENIX workshop on Offensive Technologies*. WOOT '07. 2007, 2:1–2:8.

[Wat13] Robert N. M. Watson. "A decade of OS access-control extensibility". In: *Commun. ACM* 56.2 (2013), pages 52–63. ISSN: 0001-0782.

[WD01] David Wagner and Drew Dean. "Intrusion Detection via Static Analysis". In: *Proceedings of the 2001 IEEE Symposium on Security and Privacy*. SP '01. 2001, pages 156–.

[WG92] Niklaus Wirth and Jürg Gutknecht. *Project Oberon: The Design of an Operating System and Compiler*. 1992. ISBN: 0-201-54428-8.

[Woj] R. Wojtczuk. *Adventures with a certain Xen vulnerability (in the PVFB backend)*. Message sent to bugtraq mailing list on October 15th, 2008.

[Wri+02] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman. “Linux Security Module Framework”. In: *Proceedings of the Ottawa Linux Symposium*. (Ottawa, OT, Canada, June 26–29, 2002). 2002, pages 604–617.

[WS02] David Wagner and Paolo Soto. “Mimicry attacks on host-based intrusion detection systems”. In: *Proceedings of the 9th ACM conference on Computer and communications security*. CCS ’02. 2002, pages 255–264. ISBN: 1-58113-612-9.

[Yee+09] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. “Native Client: A Sandbox for Portable, Untrusted x86 Native Code”. In: *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*. SP ’09. 2009, pages 79–93. ISBN: 978-0-7695-3633-0.

[Zha+11] Lu Zhao, Guodong Li, Bjorn De Sutter, and John Regehr. “ARMor: Fully Verified Software Fault Isolation”. In: *Proceedings of the 11th ACM Conference on Embedded Software (EMSOFT 2011)*. (Taipei, Taiwan). 2011.

[ZHR] Michal Zalewski, Niels Heinen, and Sebastian Roschke. *skipfish. Web application security scanner*.

[ZK10] Christoph Zengler and Wolfgang Küchlin. “Encoding the Linux Kernel Configuration in Propositional Logic”. In: *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010) Workshop on Configuration 2010*. 2010, pages 51–56.

[Dav] Dave Jones. *Trinity, a Linux system call fuzzer*. `http://codemonkey.org.uk/projects/trinity/`.

[OWA] OWASP. *Attack Surface Analysis Cheat Sheet*. `https://www.owasp.org/index.php/Attack_Surface_Analysis_Cheat_Sheet`.

[Wik] Wikipedia. *Buffer Overflow*. `en.wikipedia.org/wiki/Buffer_overflow`.