Quantifiable Run-time Kernel Attack Surface Reduction

Anil Kurmus[1], Sergej Dechand[2], and Rüdiger Kapitza[3]

[1] IBM Research – Zurich
kur@zurich.ibm.com
[2] Universität Bonn
dechand@cs.uni-bonn.de
[3] TU Braunschweig
kapitza@ds.tu-bs.de

Abstract. The sheer size of commodity operating system kernels makes them a prime target for local attackers aiming to escalate privileges. At the same time, as much as 90% of kernel functions are not required for processing system calls originating from a typical network daemon. This results in an unnecessarily high exposure. In this paper, we introduce kRazor, an approach to reduce the kernel's attack surface by limiting the amount of kernel code accessible to an application. KRAZOR first traces individual kernel functions used by an application. KRAZOR can then detect and prevent uses of unnecessary kernel functions by a process. This step is implemented as a kernel module that instruments select kernel functions. A heuristic on the kernel function selection allows KRAZOR to have negligible performance overhead. We 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 Introduction

Vulnerabilities in commodity operating-system kernels, such as Windows, OS X, Linux, and their mobile counterparts, are routinely exploited. For instance, the Linux kernel had more than 100 Common Vulnerabilities and Exposures (CVE) entries in 2013 and recent public local privilege escalation exploits, e.g., for CVE-2013-2094 and CVE-2012-0056.

As better exploit hardening and sandboxing mechanisms are deployed for protecting user-space processes, the interest in attacking the kernel increases for attackers. For example, some iPhone jailbreaks operated with the help of iOS kernel exploits [11]. More recently, during the 2013 Pwnium contest, an attacker escaped the Chromium browser's sandbox by exploiting a Linux kernel vulnerability [12].

Intuitively, many kernel features are unnecessary, especially when operating a workload that is known in advance, such as a web server or a router. Yet those features increase the Trusted Computing Base (TCB) size, and existing solutions such as recompiling the kernel with less feature (kernel specialization), is difficult to adopt in practice, e.g., due to the loss of distribution support.

In this paper, we explore and compare novel and lightweight run-time techniques to reduce the kernel's attack surface on a per-application basis, quantify the attack surface reduction achieved by each of them, and consider performance as well as false positive trade-offs.

As each application makes use of distinct kernel functionality, we *scope* the use of kernel functionality per-application. To do so, we implement KRAZOR, 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 KRAZOR are that of any learning-based approach: false positives, whereby a kernel function has been incorrectly learned as unnecessary, can happen. To demonstrate the feasibility of our approach, we deploy KRAZOR 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 [15, 25, 31, 45, 54] or system call sandboxing [2, 9, 18, 19, 46], KRAZOR instruments at the level of individual kernel functions (and not merely the system call interface). This makes the approach *quantifiable*, and *non-bypassable*.

We quantify security benefits by using the attack surface measurement framework described in [34]. 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 KRAZOR intercepts calls to individual kernel functions, it is particularly well-suited for measurements by such a framework. In turn, this quantification enables objective comparison of security trade-offs between KRAZOR variations.

The non-bypassable property is achieved by applying the *complete mediation* principle: we 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 we 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.

Our 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 KRAZOR), by making the learning phase twice as long.

The main contributions of this paper are:

- A quantifiable, automated and non-bypassable, run-time attack surface reduction tool, KRAZOR, that operates by learning the kernel functions necessary for a given workload on a given system, and applies it at the granularity of an application.

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

Table 1. 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 *quantifiable* refers to the existence of attack surface measurements. The ± sign refers to cases where results may vary between good (✓) and bad (–).

- A case study: a long-duration, real-world measurement of the attack surface reduction and false positives achieved by KRAZOR, which also serves as a demonstration that a large part of the system-call reachable kernel code-base is not used for many traditional, security-sensitive applications.
- Quantification of the security benefits of run-time attack surface reduction under four distinct approaches for false-positive reduction.

The remainder of this paper is structured as follows: Section 2 presents related work. Section 3 provides background on security metrics and motivates the benefits and challenges of run-time attack surface reduction. Section 4 presents the design and implementation of KRAZOR. Section 5 evaluates attack surface reduction, false-positives, and performance. Finally, we discuss advantages and limitations in Section 6, and conclude the paper in Section 7.

2 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 improvements. 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. The advantages of each area of work are summarized in Table 1.

2.1 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 [1, 37]. This design goal led to micro-kernels being a good choice for security-sensitive systems [23, 26, 29]. For instance, MINIX 3 [22–24], is a micro-kernel designed for security: in particular, its kernel is particularly small, at around 4,000 source lines of code (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 [20, 21] 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 [32].

Kernel extension fault isolation. To remedy with this lack of "compatibility", one can attempt to isolate kernel modules of commodity OSes directly, especially device drivers [3, 6, 39, 52]. One of the first such approaches, Nooks [52], 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 [39], 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. Manually modifying the kernel source code [35] (e.g., by removing unnecessary system calls) based on a static analysis of the applications and the kernel provides a way to build a tailored kernel for an application. Chanet et al. [7] use link-time binary rewriting for a comparable result. The first use of kernel specialization with a quantification of security improvements is in [34], leveraging the built-in configurability of Linux to reduce unneeded code with an automated approach. Although this approach does not require any changes to the source code of the operating system, it still requires recompiling the kernel.

2.2 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. We 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., [13, 15, 16, 25, 31, 45, 54] and references in [14]). 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 [4]. 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 [30, 38, 53, 55]. 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 [2, 9, 18, 19, 33, 46] 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 irrecoverably 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, we 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 [17, 56]. As a consequence, the attack surface on some policies can be larger than expected. Fundamentally, KRAZOR 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 [57]. In particular, on Linux, the LSM framework was created [58] as a generic way of integrating mandatory access control (MAC) mechanisms, such as [50], into the kernel. Unlike system-call interposition, this approach can be shown to provide complete mediation [27]. 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.

2.3 Other techniques that improve kernel security

There is a wide range of techniques that can improve kernel security without reducing the kernel attack surface, we 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., [5, 48]), 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 [8, 28, 51]. For instance, the `UDEREF` feature of PaX prevents the kernel from (accidentally or maliciously) accessing user-space data and the `KERNEXEC` feature prevents attacks where the attacker returns into code situated in user-space (with kernel privileges). SVA [8] compiles the existing kernel sources into a safe instruction set architecture which is translated into native instructions by the SVA VM, providing type safety and control flow integrity.

We consider all aforementioned techniques as supplemental to kernel attack surface reduction: they can be used in conjunction to improve overall kernel security.

3 Background

This section provides a summary of security metrics previously used for measuring kernel attack surface reduction, and explains motivations and challenges of kernel attack surface reduction.

3.1 Defining and quantifying kernel attack surface

Attack Surface. Most kernel exploits take advantage of defects in the kernel source code (although, exceptionally, they can also take advantage of compiler or hardware defects). In the process of writing an exploit, it is not only necessary for the attack to find a defect (such as a double-free) in the source code, but also to find a way to trigger it. Hence, any code that a given attacker could trigger is in the *attack surface* of the kernel, regardless of it containing defects.

More formally, the attack surface is defined as a subgraph of the kernel's call graph; it is the subgraph obtained by performing a reachability analysis on the kernel's call graph, after starting at the *entry* functions, i.e., the interface with the kernel for the attacker (here, system calls). Additionally, when performing this reachability analysis, we take into consideration functions that may not be reachable for other reasons, e.g., because the attacker is not privileged enough, or because they belong to a kernel module which the attacker cannot load. Those functions are referred to as *barrier* functions.

Security Model. A security model that models the attacker (and the kernel) is needed in order to assign a set of functions as entry or barrier functions.

We chose a variant of the ISOLSEC security model previously defined in [34] with some adaptations for our use, and named it STATICSEC. In a nutshell, the GENSEC model makes the simplistic assumption that the entire kernel is the attack surface. This model is suitable for comparison with previous work (with classical TCB metrics) and

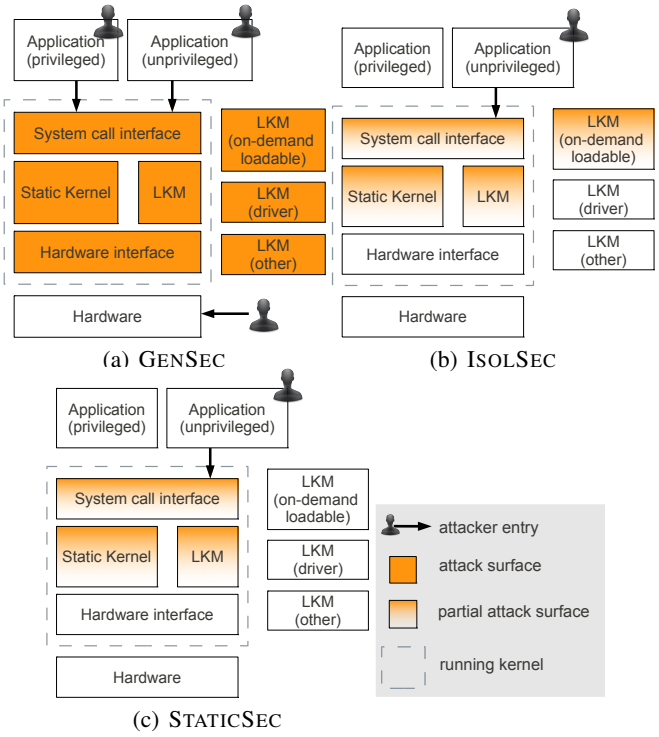

Fig. 1. Three possible security models for quantifying kernel attack surface. GENSEC is a strawman security model for explanation purposes. STATICSEC is the model used in our evaluations, and differs from ISOLSEC by assuming that no additional LKMs can be automatically loaded.

provides an upper bound on the attack surface measurements. The ISOLSEC model assumes that the attacker is local and unprivileged, and only has access to the system call interface. This model is typically suitable for environments where process sandboxing is used to restrict the impact of vulnerabilities in user-space components, which corresponds precisely to the security model of this work, since we target protection of the kernel against local attackers. However, the ISOLSEC model assumes that the attacker can trigger additional loadable kernel modules (LKMs) to be loaded. In contrast, the STATICSEC model assumes only the LKMs loaded for the specific workload running on the machine are available to the attacker. This is realistic because disabling this behavior is straightforward (e.g., by enabling the `modules_disabled` system control parameter available since Linux 2.6.31) and is a well-known approach to improve security of Linux servers. Hence, we opted for this model to evaluate the attack surface reduction that can be achieved by KRAZOR. Clearly, using the ISOLSEC model instead would result in higher attack surface reduction results. All three models are summarized in Figure 1 for comparison.

We note that the ISOLSEC or STATICSEC security model also specifies the attacker model which KRAZOR assumes: the attacker controls a local unprivileged process (e.g., because it remotely compromised the web server), targeting the kernel by making use of

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

Table 2. 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`.

kernel vulnerabilities (which can be information leaks, denial of service, or full kernel compromise to achieve privilege escalation).

Metrics. To measure the attack surface and quantify security improvements, one could use various attack surface metrics. A simple one is the sum of the SLOC count over each of the functions in the attack surface, also denoted AS_{SLOC}. Similarly, we can use cyclomatic complexity of each function[41] as a metric instead of the SLOC, or use a CVE-based metric associating the value 1 to a function that had a CVE in the past 7 years (a total of 422 CVEs for the Linux kernel), and 0 otherwise. We respectively denote those attack surface metrics AS_{cycl} and AS_{CVE}.

3.2 Motivations and challenges for run-time attack surface reduction

The results for compile-time attack surface reduction in [34] 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}). We now 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 kernel functions are used by the `qemu-kvm` process, which is running in one node serving as KVM hypervisor on our test bed. The lowest number is achieved by the MYSQL daemon. Table 2 compares these results and shows that, potentially, restricting

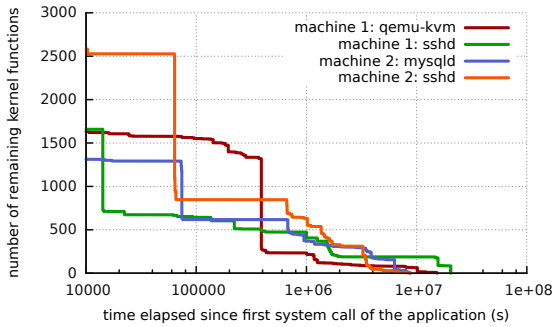

Fig. 2. Evolution of the number of unique kernel functions used by applications: after a few months, no new kernel functions were triggered.

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 our 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 2, 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 is 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 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.

4 Run-time kernel attack surface reduction

In this section, we detail the design and implementation of KRAZOR, a tool that aims to achieve 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 3 and detailed below.

❶ **Pre-learning phase.** The goal of this phase is to prepare an enforcement phase (and incidentally, learning phase) with low performance overhead. At first, KRAZOR 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, we select, each time the trace buffer fills up, functions which are called beyond a given threshold

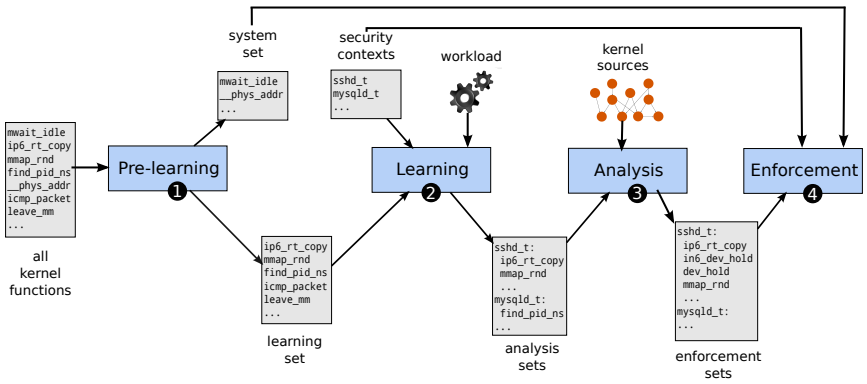

Fig. 3. KRAZOR run-time kernel attack surface reduction phases.

and disable tracing for those functions. These functions form the *system set*, while the remaining kernel functions form the *learning set*.

Our 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`, `get_page`, `__page_cache_alloc`), or synchronization (`_spin_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 singificantly reduce kernel attack surface (since most applications would end up using them anyway). Listing 1.1 shows a more subtle example of a function included in the system set: `ext4_claim_free_blocks` is repeadetly 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)) {
      ...
      ar->len = ar->len >> 1;
    }
  ...
}
```

Listing 1.1. 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.

❷ **Learning phase.** In this phase, 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 3) 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 KRAZOR, we thus make use of SELinux [50] security contexts as security context (in Figure 3, 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.

We have implemented this step in two different ways: first, we 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 paper). However, as some system administrators have been wary of installing a kernel module, we 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. The user-space tool which is used for phases ❶ and ❷ consists of 1600 lines of Python code.

❸ **Analysis phase.** In this phase, we 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.

We 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, we 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, we chose k-means because of its well known scalability (due to the size of the kernel call graph). In particular, we make use of the very scalable *mini-batch* k-means algorithm described in [47]. In our experiments, clustering individual functions led to unevenly-sized clusters and unsatisfactory evaluation results. Therefore, we opted for using file grouping: each node in our 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. We 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 [47].

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.

❹ **Enforcement phase.** Finally, we enforce 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, we 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, we make use of the Linux kernel's KPROBES feature to insert probes at the very beginning of each of those functions. The kernel module consists of 700 lines of C code, and receives the results of phases ❶ and ❷ through `procfs`.

Currently, two options exist for the enforcement phase: the first is to log the violation, and the second one is a fail-stop behavior, triggering a kernel oops (which will atempt 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.

5 Evaluation

5.1 Evaluation use case

To measure the security benefits, in terms of attack surface reduction as well as false positives, and performance, we 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 we 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 we trace `qemu-kvm` which is the user-space process running drivers on the host for virtualizing hardware to the guest virtual machines.

5.2 Attack surface reduction

We compute the reduced attack surface by using the enforcement set for each application 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, SLOC and cyclomatic complexity metrics are calculated on a per-function basis by FRAMA-C. This approach to quantifying attack surface is an extension of that we described and previously used in [34], with modifications mainly to support the kernel we used for our evaluation and the modified security model.

Table 3 summarizes attack surface reduction results for all services, grouping algorithms, and attack surface metrics in our 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. This also corresponds to a false-negative evaluation: since any kernel function in the attack surface can potentially have an exploitable vulnerability, the lower the attack surface reduction, the higher the false negatives.

		Baseline	kRazor			
			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 3. Summary of kRazor 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.

5.3 False positives

In our 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, we opt to use the first 20% of the collected traces as a learning phase, and the remaining 80% as an enforcement phase[4]. 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 4, together with the convergence rate. We 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.

5.4 Performance

We measure performance during the enforcement phase with the LMBench 3 benchmarking suite. We perform 5 runs and collect the average latency, which is reported in Table 5. Most overheads are very low (especially considering this is a micro-benchmark): the pre-learning phase is effective in segregating performance-sensitive kernel functions.

[4] 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).

		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 4. 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	KRAZOR	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. Latency time and overhead for various OS operations (in microseconds)

However, some operations (e.g., empty file creation) can incur significant overhead (75%), which shows that our heuristic approach still has room for improvement — although file creation is not a performance-critical operation in most workloads.

As a macro-benchmark, we use the `mysqlslap` load-generation and benchmarking tool for MYSQL. We 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 we compute a 95%-confidence interval over. Results in Table 6 show that KRAZOR incurs no measurable overhead. In addition, the results confirm the pre-learning phase's effectiveness: without this phase, KRAZOR would incur more than 100% overhead on this test.

	Baseline	KRAZOR	W/o pre-learning
Average	2.30 ± 0.00	2.31 ± 0.00	4.67 ± 0.01
Overhead		0.4%	103%

Table 6. MySQL-slap benchmark: average time to execute 5000 SQL queries (in seconds)

5.5 Detection of past vulnerabilities

We now focus on four vulnerabilities for the Linux kernel for which a public kernel exploit was available. We provide a description of each vulnerability, and pinpoint the individual kernel function responsible for the vulnerability.

KRAZOR detects exploits targeting such vulnerabilities in many cases (see Table 7). 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 our machine, and then attempted to elevate his privileges on the host using an exploit for the kernel, KRAZOR would detect the exploit. In particular, we 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 KRAZOR in the enforcement phase, and, by definition, it's not possible to write an exploit for a vulnerability without triggering the vulnerability.

Finally, we note that the AS_{CVE} metric results (in Table 3) provide figures for estimating KRAZOR's effectiveness in detecting exploits for past CVEs in a statistically significant manner. The following examples are 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). This vulnerability discovered by Jason A. Donenfeld [10] 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`. Although KRAZOR 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 KRAZOR does not instrument it). This means KRAZOR prevents this vulnerability.

`sk_run_filter` (CVE-2010-4158). This vulnerability is in the Berkeley Packet Filter (BPF) [42] 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

	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 7. Detection of previously exploited kernel vulnerabilities by KRAZOR (for each grouping). Legend: ✓: detected for all use cases, S: detected for `sshd`, M: detected for `mysqld`.

randomization). In our evaluation, KRAZOR 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 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 KRAZOR, even after grouping. However, unlike the three previous exploits, this vulnerability would also have been prevented 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.

6 Discussion

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

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

Analysis phase: grouping algorithms and trade-offs. Figure 4 depicts convergence and attack surface trade-offs for all four grouping methods explored in this work.

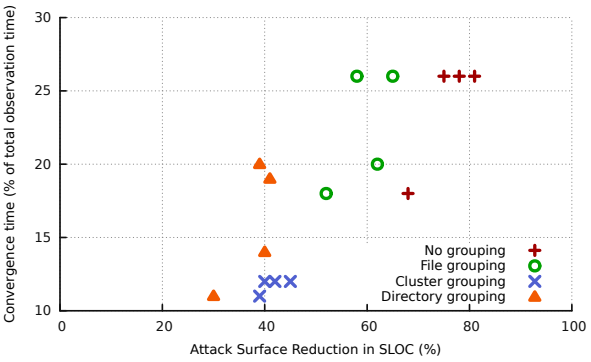

Fig. 4. Attack surface reduction and convergence rate for the evaluated applications, under different grouping methods.

The closer a data point is to the bottom right corner of this graph, the better the trade-off. For instance, we 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 KRAZOR, these trade-offs can be adapted to the workload and target service: for instance, in use-cases where the workload is less well defined, clustering grouping is a more attractive solution: it converges about twice as fast as the other algorithms.

False positives. In our evaluation of false-positives, we 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 we 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 positive: for all grouping methods and services, no false positives were observed for about a full year.

Performance trade-offs. The pre-learning phase contains a tunable parameter that sets the threshold for disable tracing of performance-sensitive functions. Because our results showed low performance overhead with good attack surface reduction, we did not tweak this parameter in our evaluations. However, we 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).

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 remarkable, because SLOC and cyclomatic complexity are a priori metrics (i.e., they aim to 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 [49]. We conclude the reduction observed is not merely in terms of lines of code, but really in the number of exploitable vulnerabilities.

Attack surface size and TCB. In absolute terms, our 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 [23]), the Fiasco microkernel (15K SLOC [20]) or Flicker (250 SLOC [43]), 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 [44]).

Hence, we 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.

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 [36]), and it would possible to expand KRAZOR with existing solutions. For example, Akeso [36] 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.

7 Conclusion

We presented a lightweight, per-application, run-time kernel attack surface reduction framework restricting 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 [11, 12] because of the rise of application sandboxes and the general increase in user-space hardening. The main goal of KRAZOR is to provide a way of reducing the kernel attack surface in a quantifiable and non-bypassable way. KRAZOR 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. Our evaluation shows that attack surface reduction is significant (from 30% to 80%) both in terms of lines of code and CVEs.

KRAZOR is implemented for the Linux kernel only, however the approach can be adapted to other operating systems: in particular, it assumes no source code access (apart from the use of kernel sources for the grouping algorithms and the attack surface quantification, which can be avoided in practice).

In its current state, KRAZOR 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. We envision the learning phase would be turned on when the server is tested prior to being put into production. In production, KRAZOR can detect many unknown kernel exploits and report, for example, to security incident and event management (SIEM) tools typically deployed nowadays.

Finally, this work further confirms that the notion of attack surface is a powerful way to quantify security improvements: it would not be possible to quantify improvements here with traditional TCB size measurements. We foresee that this notion can have wider application: for instance, the attack surface delimited thanks to KRAZOR could be used to steer source code analysis work preferably towards code that is reachable to attackers, and to prioritize kernel hardening efforts.

Bibliography

[1] Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A., Young, M.: MACH: A New Kernel Foundation for UNIX Development. In: Proceedings of the USENIX Summer Conference (1986)
[2] Acharya, A., Raje, M.: MAPbox: using parameterized behavior classes to confine untrusted applications. In: Proceedings of the 9th conference on USENIX Security Symposium-Volume 9 (2000)
[3] Boyd-Wickizer, S., Zeldovich, N.: Tolerating malicious device drivers in linux. In: Proceedings of the 2010 USENIX conference on USENIX annual technical conference. Berkeley, CA, USA (2010)
[4] Canali, D., Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M., Kirda, E.: A quantitative study of accuracy in system call-based malware detection. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis. New York, NY, USA (2012)
[5] Carbone, M., Cui, W., Lu, L., Lee, W., Peinado, M., Jiang, X.: Mapping kernel objects to enable systematic integrity checking. In: Proceedings of the 16th ACM conference on Computer and communications security. New York, NY, USA (2009)
[6] Castro, M., Costa, M., Martin, J.P., Peinado, M., Akritidis, P., Donnelly, A., Barham, P., Black, R.: Fast byte-granularity software fault isolation. In: [40]
[7] Chanet, D., Sutter, B.D., Bus, B.D., Put, L.V., Bosschere, K.D.: 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). New York, NY, USA (2005)
[8] Criswell, J., Lenharth, A., Dhurjati, D., Adve, V.: Secure virtual architecture: A safe execution environment for commodity operating systems. In: Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07). New York, NY, USA (2007)
[9] Dan, A., Mohindra, A., Ramaswami, R., Sitaram, D.: Chakravyuha: A sandbox operating system for the controlled execution of alien code. Tech. rep., IBM TJ Watson research center (1997)
[10] Donenfeld, J.A.: Linux local privilege escalation via suid /proc/pid/mem write (2012), http://blog.zx2c4.com/749
[11] Esser, S.: iOS Kernel Exploitation. http://media.blackhat.com/bh-us-11/Esser/BH_US_11_Esser_Exploiting_The_iOS_Kernel_Slides.pdf (2011)
[12] Evans, C.: Pwnium 3 and Pwn2Own Results. http://blog.chromium.org/2013/03/pwnium-3-and-pwn2own-results.html (2012)
[13] Feng, H.H., Kolesnikov, O.M., Fogla, P., Lee, W., Gong, W.: Anomaly detection using call stack information. In: Proceedings of the 2003 IEEE Symposium on Security and Privacy. Washington, DC, USA (2003)
[14] Forrest, S., Hofmeyr, S., Somayaji, A.: The evolution of system-call monitoring. In: Proceedings of the 2008 Annual Computer Security Applications Conference. Washington, DC, USA (2008)
[15] Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A sense of self for unix processes. In: Proceedings of the 1996 IEEE Symposium on Security and Privacy. Washington, DC, USA (1996)
[16] Gao, D., Reiter, M.K., Song, D.: Gray-box extraction of execution graphs for anomaly detection. In: Proceedings of the 11th ACM conference on Computer and communications security. New York, NY, USA (2004)
[17] Garfinkel, T.: Traps and pitfalls: Practical problems in system call interposition based security tools. In: NDSS (2003)
[18] Goldberg, I., Wagner, D., Thomas, R., Brewer, E.A.: 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)
[19] Google: Seccomp sandbox for linux (2009)
[20] Hartig, H., Hohmuth, M., Feske, N., Helmuth, C., Lackorzynski, A., Mehnert, F., Peter, M.: The nizza secure-system architecture. In: Collaborative Computing: Networking, Applications and Worksharing, 2005 International Conference on (2005)
[21] Heiser, G., Leslie, B.: The okl4 microvisor: convergence point of microkernels and hypervisors. In: Proceedings of the first ACM asia-pacific workshop on Workshop on systems. New York, NY, USA (2010)
[22] Herder, J.N., Bos, H., Gras, B., Homburg, P., Tanenbaum, A.S.: Construction of a highly dependable operating system. In: Proceedings of the Sixth European Dependable Computing Conference. Washington, DC, USA (2006)
[23] Herder, J.N., Bos, H., Gras, B., Homburg, P., Tanenbaum, A.S.: Minix 3: a highly reliable, self-repairing operating system. SIGOPS Oper. Syst. Rev. 40(3) (2006)
[24] Herder, J.N., Bos, H., Gras, B., Homburg, P., Tanenbaum, A.S.: 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. Washington, DC, USA (2008)

[25] Hofmeyr, S.A., Forrest, S., Somayaji, A.: Intrusion detection using sequences of system calls. J. Comput. Secur. 6(3) (1998)
[26] Hohmuth, M., Peter, M., Härtig, H., Shapiro, J.S.: Reducing tcb size by using untrusted components: small kernels versus virtual-machine monitors. In: Proceedings of the 11th workshop on ACM SIGOPS European workshop. New York, NY, USA (2004)
[27] Jaeger, T., Edwards, A., Zhang, X.: Consistency analysis of authorization hook placement in the linux security modules framework. ACM Trans. Inf. Syst. Secur. 7(2) (2004)
[28] Kemerlis, V.P., Portokalidis, G., Keromytis, A.D.: kguard: lightweight kernel protection against return-to-user attacks. In: Proceedings of the 21st USENIX conference on Security symposium. Berkeley, CA, USA (2012)
[29] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: sel4: formal verification of an os kernel. In: [40]
[30] Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Automating mimicry attacks using static binary analysis. In: Proceedings of the 14th conference on USENIX Security Symposium - Volume 14. Berkeley, CA, USA (2005)
[31] Kruegel, C., Mutz, D., Valeur, F., Vigna, G.: On the detection of anomalous system call arguments. Computer Security–ESORICS 2003 (2003)
[32] Kurmus, A., Gupta, M., Pletka, R., Cachin, C., Haas, R.: A comparison of secure multi-tenancy architectures for filesystem storage clouds. In: Middleware (2011)
[33] Kurmus, A., Sorniotti, A., Kapitza, R.: Attack Surface Reduction For Commodity OS Kernels. In: Proceedings of the Fourth European Workshop on System Security (2011)
[34] Kurmus, A., Tartler, R., Dorneanu, D., Heinloth, B., Rothberg, V., Ruprecht, A., Schröder-Preikschat, W., Lohmann, D., Kapitza, R.: Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In: Proceedings of the 20th Network and Distributed System Security Symposium (2013)
[35] Lee, C., Lin, J., Hong, Z., Lee, W.: An application-oriented linux kernel customization for embedded systems. Journal of information science and engineering 20(6) (2004)
[36] Lenharth, A., Adve, V.S., King, S.T.: 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. New York, NY, USA (2009)
[37] Liedtke, J.: On μ-kernel construction. In: Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95) (1995)
[38] Ma, W., Duan, P., Liu, S., Gu, G., Liu, J.C.: Shadow attacks: automatically evading system-call-behavior based malware detection. J. Comput. Virol. 8(1-2) (2012)
[39] Mao, Y., Chen, H., Zhou, D., Wang, X., Zeldovich, N., Kaashoek, M.F.: Software fault isolation with api integrity and multi-principal modules. In: Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11). New York, NY, USA (2011)
[40] Matthews, J.N., Anderson, T.E. (eds.): Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09). New York, NY, USA (2009)
[41] McCabe, T.: A complexity measure. Software Engineering, IEEE Transactions on SE-2(4) (1976)
[42] McCanne, S., Jacobson, V.: 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. Berkeley, CA, USA (1993)
[43] McCune, J.M., Parno, B.J., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: an execution infrastructure for tcb minimization. SIGOPS Oper. Syst. Rev. 42(4) (2008)
[44] Murray, D.G., Milos, G., Hand, S.: Improving xen security through disaggregation. In: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments. New York, NY, USA (2008)
[45] Mutz, D., Valeur, F., Vigna, G., Kruegel, C.: Anomalous system call detection. ACM Trans. Inf. Syst. Secur. 9(1) (2006)
[46] Provos, N.: Improving host security with system call policies. In: Proceedings of the 12th conference on USENIX Security Symposium-Volume 12 (2003)
[47] Sculley, D.: Web-scale k-means clustering. In: Proceedings of the 19th international conference on World wide web. New York, NY, USA (2010)
[48] Seshadri, A., Luk, M., Qu, N., Perrig, A.: 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. New York, NY, USA (2007)
[49] Shin, Y., Williams, L.: Is complexity really the enemy of software security? In: Proceedings of the 4th ACM workshop on Quality of protection. New York, NY, USA (2008)
[50] Smalley, S., Vance, C., Salamon, W.: Implementing SELinux as a Linux security module. Tech. rep., NAI Labs Report (2001)
[51] Spengler, B., PaX team: grsecurity kernel patches. `www.grsecurity.net` (2003)
[52] Swift, M.M., Martin, S., Levy, H.M., Eggers, S.J.: 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". New York, NY, USA (2002)
[53] Tan, K.M.C., McHugh, J., Killourhy, K.S.: Hiding intrusions: From the abnormal to the normal and beyond. In: Revised Papers from the 5th International Workshop on Information Hiding. London, UK, UK (2003)
[54] Wagner, D., Dean, D.: Intrusion detection via static analysis. In: Proceedings of the 2001 IEEE Symposium on Security and Privacy. Washington, DC, USA (2001)
[55] Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In: Proceedings of the 9th ACM conference on Computer and communications security. New York, NY, USA (2002)
[56] Watson, R.N.M.: Exploiting concurrency vulnerabilities in system call wrappers. In: Proceedings of the first USENIX workshop on Offensive Technologies. Berkeley, CA, USA (2007)
[57] Watson, R.N.M.: A decade of os access-control extensibility. Commun. ACM 56(2) (2013)
[58] Wright, C., Cowan, C., Morris, J., Smalley, S., Kroah-Hartman, G.: Linux security module framework. In: Ottawa Linux Symposium. vol. 8032 (2002)