usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

SHARD: Fine-Grained Kernel Specialization with Context-Aware Hardening

Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu, *Purdue University*

https://www.usenix.org/conference/usenixsecurity21/presentation/abubakar

This paper is included in the Proceedings of the 30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the 30th USENIX Security Symposium is sponsored by USENIX.

SHARD: Fine-Grained Kernel Specialization with Context-Aware Hardening

Muhammad Abubakar Adil Ahmad Pedro Fonseca Dongyan Xu
Department of Computer Science and CERIAS, Purdue University
{mabubaka, ahmad37, pfonseca, dxu}@purdue.edu

Abstract

With growing hardware complexity and ever-evolving user requirements, the kernel is increasingly bloated which increases its attack surface. Despite its large size, for specific applications and workloads, only a small subset of the kernel code is actually required. Kernel specialization approaches exploit this observation to either harden the kernel or restrict access to its code (debloating) on a per-application basis. However, existing approaches suffer from coarse specialization granularity and lack strict enforcement which limits their effectiveness.

This paper presents SHARD, a practical framework to enforce fine-grain kernel specialization. SHARD specializes at both the application and system call levels to significantly restrict the kernel code exposed to attackers. Furthermore, SHARD introduces context-aware hardening to dynamically enable code hardening during suspicious execution contexts. SHARD implements an instance of a context-aware hardening scheme using control-flow integrity (CFI), which provides near-native performance for non-hardened executions and strong security guarantees. Our analysis of the kernel attack surface reduction with SHARD as well as concrete attacks shows that SHARD exposes 181× less kernel code than the native kernel, an order of magnitude better than existing work, and prevents 90% of the evaluated attacks. Our evaluation shows that the average performance overhead of SHARD on real-world applications is moderate—10% to 36% on NGINX, 3% to 10% on Redis, and 0% to 2.7% on the SPEC CPU 2006 benchmarks.

1 Introduction

Operating system kernels have seen an exponential growth during the last two decades. The Linux kernel, for instance, grew from 2.4 million [15] lines of source code in 2001 to a staggering 27.8 million lines of source code in 2020 [14]. This growth is in large part a consequence of an increasingly diverse range of functions (e.g., supporting many devices) implemented by modern kernels. Unfortunately, because larger kernels increase the trusted computing base (TCB), systems have become increasingly vulnerable to attacks that exploit kernel defects to take complete control of the machine.

A promising approach to minimize any software codebase is by specialization through debloating [21, 31, 47, 50], which retains a small part of the codebase required for specific workloads and prevents the rest of the code from running. In the context of the kernel, debloating the kernel code for specific applications [30, 36], can reduce the kernel code to 8.89% of its native size and prevent attackers from exploiting many kernel vulnerabilities without hindering application functionality. However, because kernels are so large, even such kernel code reduction leaves vulnerable a significant part of the kernel, which can be exploited by attackers.

This paper proposes, SHARD, a practical framework for dynamic kernel specialization that implements fine-grained specialization. Unlike previous work that limits the granularity of specialization to the application level, SHARD goes significantly beyond by specializing the kernel at the system call level for each target application, which further constraints the amount of kernel code that an attacker can leverage. As a result, SHARD exposes 181× less kernel code, on average, than the native linux kernel, which is an order of magnitude better than existing work on kernel debloating [30].

At a high-level, SHARD first identifies the kernel code required to execute a system call by a specific application and then, during run-time, it ensures that only that kernel code is allowed to run when the application invokes the same system call. By profiling Linux with real-world applications, we concluded that in the majority of cases, two system calls share less than half of the kernel code that they execute. This low-overlap is expected because the kernel implements several classes of services (e.g., file operations, network operation, process management) using distinct code. Hence, fine-grained specialization, at the system call and application-level, significantly reduces the amount of kernel code exposed to attackers at any given point.

In addition to employing fine-grained specialization, SHARD also addresses the challenge of identifying the parts of the kernel that a system call, invoked by a specific application, should be allowed to execute, i.e., the kernel coverage of system calls. Dynamic profiling of applications [30,36–38,53,62] and static program analysis techniques [29,42,56,61] are common techniques used to identify the coverage of legitimate execution (e.g., code that does not subvert the control-flow of the kernel) but these techniques are either incomplete or unsound when applied to complex systems, such as the kernel. As a result of these limitations, prior specialization techniques compromise the security guarantees by either (a) only logging executions that reach unexpected code [30], instead of strictly enforcing specialization, which makes them ineffective at preventing attacks, or (b) overestimating the code that should be allowed to execute, which significantly increases the amount of code that attackers can use.

SHARD implements *context-aware hardening*, a new technique to address the limitations of program analysis and dynamic profiling techniques on complex code, such as kernels. Context-aware hardening *dynamically hardens* kernel code for suspicious executions, i.e., profiling or static analysis could not determine that the execution should be allowed or not. Because kernel code that falls under this class, even though representing more than half of the kernel, only rarely executes, context-aware hardening is a low-cost solution, unlike full-system hardening, that enables strict debloating enforcement.

Context-aware hardening allows SHARD to dynamically switch between hardened and non-hardened code according to the specialization policy *during* a system call execution. SHARD implements a specific context-aware hardening mechanism using fine-grained control-flow integrity (CFI) [20]. However, dynamic switching between CFI hardened and non-hardened code versions is challenging. First, CFI uses integer-based indexing at indirect call sites instead of function pointers, which must be consistent with non-hardened code versions to allow switching; therefore, non-hardened execution would also be impacted (i.e., up to 40% overhead [29]). Second, the switch from non-hardened to hardened code execution requires a special CFI check; since non-hardened code does not ensure CFI during the transition. SHARD deals with these challenges through a modified CFI instrumentation, which relies on function addresses, and a custom CFI check using Last Branch Record (LBR), ensuring secure transitions from non-hardened to hardened code execution.

SHARD relies on an offline analysis, to determine kernel coverage, and an online phase, during which the system is protected. During the offline analysis, SHARD analyzes the kernel to determine per-system call code coverage (i.e., required kernel code) for specific, benign application workloads. SHARD achieves this using two program analysis approaches — dynamic profiling, which may under-approximate coverage, and static analysis, which may over-approximate the coverage. During the online phase, SHARD uses a VMX-based monitor to transparently enforce kernel debloating and context-aware hardening. Importantly, SHARD does not require man-

	Spec.		**Protection**			**Kernel Instr.**	**Overhead**
	S	**A**	**Ratio**	**Strict**	**Type**		
Specialized hardening							
SplitKernel [39]	✗	✓	Full	N/A	Coarse CFI*	Manual	3-40%
ProxOS [52]	✗	✓	N/A	N/A	Isolation	Manual	200-2400%
Dynamic debloating							
FACECHANGE [30]	✗	✓	11.3×	✗	Debl.	Auto	0-40%
Multi-K [36]	✗	✓	11.3×	✗	Debl.	Manual	0-0.5%
Static debloating							
Kurmus et al [38]	✗	✓	4-5×	✓	Debl.	Manual	0%
Kuo et al [37]	✗	✓	6.5-7.5×	✓	Debl.	N/A	0%
SHARD [*this work*]	✓	✓	181×	✓	Debl. + Fine CFI*	Auto	3-36%

Table 1: Comparison of SHARD with prior kernel specialization work. Table compares the granularity of specialization ("Spec"), system call-level ("S") and application-level ("A"); Ratio, strictness and type of kernel protection; kernel instrumentation required; and the application overhead. SplitKernel [39] implements stack exhaustion and stack clearance checking, alongside coarse CFI, as hardening. SHARD implements Fine CFI according to the context-aware policy.

ual modifications to the kernel source code, instead it employs compile-time instrumentation to transparently introspect kernel state required by the specialization policies.

We evaluated SHARD's effectiveness on two popular applications, the Redis key-value store and the NGINX web server. Our evaluation shows that SHARD reduces, on average, the number of kernel instructions accessible to 0.49% for Redis and 0.60% for NGINX, compared to the native Linux kernel. Similarly, the number of ROP gadgets is reduced to 0.55% and 0.60% respectively. In addition, SHARD protects the kernel against 90% of the attack scenarios in our experiments by preventing the execution of the vulnerable code or the exploit payload. We found that the average overhead of SHARD is only 3-10% across the `redis-benchmark` test suite for Redis and 10-36% across varying request sizes for NGINX, despite reducing the code by 181×, an order of magnitude better than previous work and strictly enforcing specialization. Finally, on the SPEC CPU integer workloads, we observe a small overhead of only 0-2.7%.

This paper makes the following main contributions:

- Fine-grained specialization, a kernel specialization scheme that operates at the system call and application level to increase specialization effectiveness.
- Context-aware hardening, a general approach to selectively harden code during system calls to provide strict and efficient specialization enforcement.
- The design of SHARD, the first fine-grained specialization framework for commodity unmodified kernels.
- An evaluation of SHARD on real-world applications and real-world exploits and vulnerabilities.

The rest of the paper is organized as follows: §2 provides background on kernel specialization and motivates SHARD. §6 describes the threat model of SHARD. §7 and §8 describe the design and implementation of SHARD. §9 provides a secu-

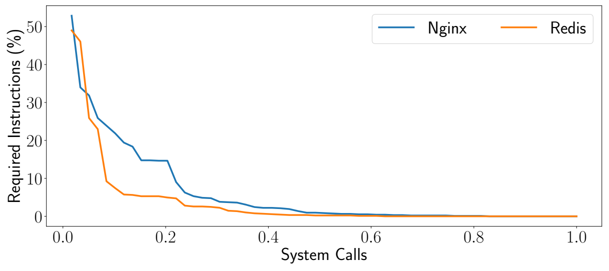


Figure 1: Distribution of instructions executed by each system call made by Nginx and Redis. Numbers are normalized to the total number of instructions required by each application and system calls are sorted from highest to lowest.

rity analysis. §10 discusses the performance evaluation.§11, §12 and §13 discuss limitations, related work, and conclude.

2 Background on Kernel Specialization

Kernel specialization approaches to improve system security rely on either hardening [39, 52] or debloating (i.e., minimizing) [30, 36, 36, 38, 62]. Hardening approaches generate two versions of the kernel and during run-time ensure that untrusted applications (i.g., target applications) use the hardened version of the kernel while trusted application use the native kernel version, without performance overhead.

Debloating approaches only allow the execution of kernel code that a certain application, or group of applications, requires. The remaining code of the kernel is either completely removed statically [37, 38, 53], retained in the binary but obfuscated [30], or made inaccessible at run-time [36, 62]. By minimizing the accessible code, debloating reduces the attack surface for code reuse attacks [47, 50], i.e., attacks reusing existing code sequences such as ROP gadgets [49], and can generally reduce vulnerabilities in software [21, 36, 37].

Previous debloating work enforces kernel specialization either at compile-time [37, 38, 53] or run-time [30, 36, 62]. Both approaches rely on an *analysis* phase to identify relevant kernel code for a set of applications, by executing the applications under representative workloads or by using static analysis techniques, such as control-flow graph analysis. After analysis, compile-time approaches statically compile a custom configured kernel containing only the required kernel features. While run-time approaches create multiple versions of the kernel (e.g., one kernel version for each target application) and dynamically switch the system's *kernel-view* whenever the executed application changes.

2.1 Limitations of Existing Approaches

Despite extensive work on specialization techniques [30, 36–38, 53, 62], existing kernel specialization techniques, as summarized in Table 1, are limited to coarse specialization and do not provide strict debloating enforcement, which seriously limits their effectiveness.

Coarse specialization. Existing kernel hardening and debloating specialization approaches are coarse because they only create a single kernel-view for the entire application. As a result, existing approaches do not prevent a system call invocation from using the kernel code that should only be accessed through other system calls by the application. Hence, they have a low *protection ratio* (i.e., the ratio of baseline to exposed instructions) that unnecessarily exposes a large quantity of code for attack purposes.

To demonstrate the security impact of this limitation, we devise an experiment employing single-view kernel specialization for two popular applications, the NGINX [16] web server and the Redis [17] key-value store. In this experiment, the applications can only access the required kernel code, as determined through dynamic profiling of application workloads (refer to §7.2 for the profiling details). Figure 1 shows what portion of the entire profiled kernel code is executed by the system calls invoked by NGINX and Redis. The results show that in both applications 80% of the system calls utilize less than 15% of the profiled kernel code at a time. This result demonstrates that further restricting which code can execute given the application profile *and* the system call context can significantly reduce the available code for attacks.

Limited debloating enforcement. Kernel debloating techniques require an accurate analysis phase (refer to §2) to provide strict debloating enforcement within the kernel. However, program analysis techniques are not complete and accurate on complex code, such as kernel code which extensively uses aliasing [22, 29, 42, 44, 56, 61], so they either under-estimate or over-estimate the kernel code required by the target applications. Existing schemes that under-estimate do *not strictly enforce debloating* [30, 36] but instead log suspicious executions, which does not prevent attacks and is hard to diagnose. In contrast, existing schemes that over-estimate allow strict enforcement but offer reduced debloating ratio and hence, limited effectiveness [37, 38].

In general, existing schemes analyze the kernel for debloating specialization either using (a) static call graph generation or (b) dynamic workload-based profiling. The static technique constructs a call graph of the kernel and identifies the kernel code that is reachable for each system call. However, this technique fails to precisely resolve indirect call sites (e.g., function pointers) and data-dependent paths; therefore, it over-estimates the required kernel code and might allow illegitimate executions during run-time.

On the other hand, dynamic profiling executes a representative application workload (e.g., test suites and benchmarks) and traces all kernel code executed by the workload. However, the profiled code coverage of such workloads is only 6% to 73% of the application's code [36]. Therefore, at run-time, an application might trigger a system call path that was not profiled but is legitimate. Existing approaches do not provide strict enforcement in such cases (i.e., only log suspicious execution paths for offline analysis [30]). Hence, a *potentially reachable code path* is exploitable.

3 Fine-grained System Call Specialization

SHARD employs fine-grained specialization by providing different kernel-views depending on the application running *and* the currently executing system call. Since the kernel implements unique system calls for distinct services, such as process management and device I/O, system calls providing orthogonal services do not share much code with each other. Hence, by specializing the kernel-view for an application at each system call, the amount of kernel code exposed to the attacker at any point is significantly reduced which further restricts the attacker's ability to construct ROP chains and exploit vulnerabilities.

To validate fine-grained specialization, consider the assembly instruction overlap between the top 10 system calls with largest coverage, invoked by NGINX and Redis during profiling, shown in Table 2. We observe that system calls providing distinct services do not share much kernel code. For example, in the case of Redis, `read`, which implements file and network I/O operations, shares only 6.8% (4.9k out of 72.1k) of its instructions with `exit_group`, which exits all process threads. Similarly, in case of NGINX, `recvfrom` which receives network packets, shares only 9.6% (5.1k out of 53k) of its instructions with `write`, which writes to local file. Given the disparity in kernel code coverage across system calls, system call-level specialization provides an opportunity to significantly reduce exposure to attacks.

However, a system call-only specialization (i.e., agnostic to the application) would not have a good protection ratio either. For example, consider the `write` system call, which shares less than 33% (22.9k out of 68.2k) of its instructions across NGINX and Redis. The reason is that NGINX only uses `write` for file I/O while Redis uses `write` for both file and network I/O. Therefore, the execution profile of `write` under Redis also includes networking functions that are not required by NGINX. Since system call-only specialization must support both NGINX and Redis, it would provide access to all instructions executable by `write` across both applications. Hence, ignoring the application dimension would inflate the attack surface in many scenarios.

4 Context-aware Hardening

SHARD employs *context-aware hardening* to address the uncertainty of whether code is reachable from a particular system call. In particular, SHARD analyzes the kernel using both static analysis techniques and dynamic workload-based profiling, to determine the accessibility of kernel code per-system call. Then, SHARD enforces hardening (e.g., control-flow integrity [20]) when it is unsure if the kernel should be allowed to execute a certain piece of code in the current context.

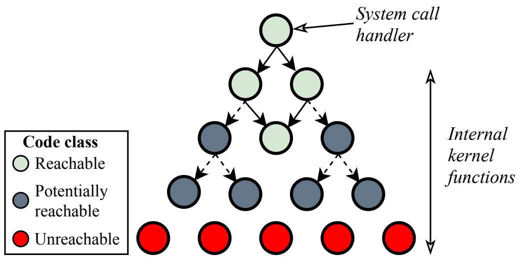


Figure 2: Classes of kernel functions relative to a system call.

SHARD classifies code into three, disjoint categories at the level of functions for each system call (as shown in Figure 2). In particular, the *reachable* nodes are the kernel functions executed during dynamic workload-based profiling (e.g., benchmarks and test suites). The *potentially reachable* nodes are the kernel functions that static analysis indicates might be reachable from a certain system call. Furthermore, static analysis can conclude (accurately) that some functions are not reachable from a certain system call; therefore, those are labeled *unreachable*.

SHARD does not harden the kernel when a system call only executes *reachable* functions. The reason is that our profiling accurately concludes that these functions are accessible by the currently invoked system call. Furthermore, *reachable* constitutes a very small portion of the kernel's code — only 0.49% and 0.60% of the native kernel's instructions are *reachable*, on average, for Redis and NGINX, respectively (Table 3). Therefore, they provide very few ROP gadgets (as we show in §9.1) and can be more easily tested for correctness.

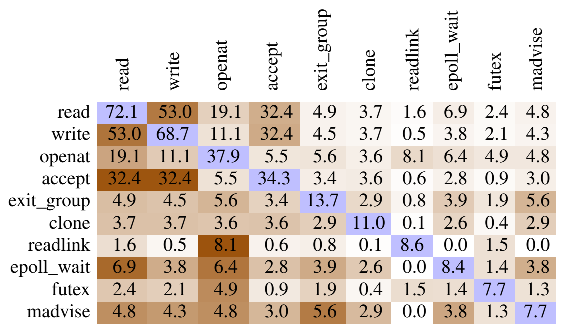
However, SHARD hardens the kernel when it detects an execution that transitions from *reachable* to *potentially reachable*, since SHARD cannot accurately conclude that *potentially reachable* code is accessible from the system call, , i.e., a *potentially reachable code path*. Therefore, hardening significantly raises the bar for attacks on the system through such executions. Furthermore, SHARD restricts access to *unreachable* functions since they should never be executed during benign kernel execution of the invoked system call.

The context-aware hardening technique employed by SHARD is fine-grained *control-flow integrity* (CFI) [20]. CFI ensures that all control-flow transfers, at run-time, adhere to a program's statically-analyzed control-flow graph (CFG). As shown by prior work [33], CFI can effectively prevent control-flow hijacks. Note that other techniques can be applied to implement context-aware hardening (as we discuss in §11).

5 System Model

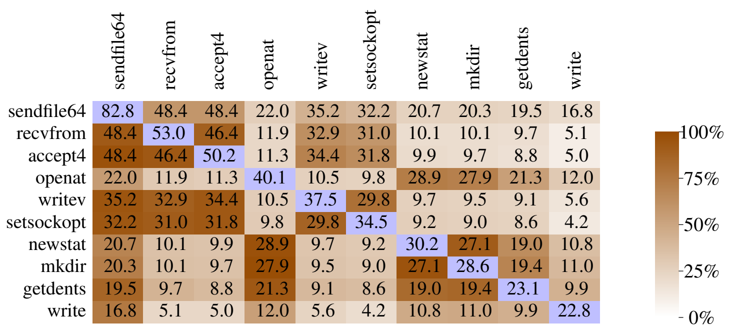
This section describes the scenario envisioned for SHARD.

Untrusted application. We assume a service provider (e.g., a website owner) needs to provide a service to many untrusted



	read	write	openat	accept	exit_group	clone	readlink	epoll_wait	futex	madvise
read	72.1	53.0	19.1	32.4	4.9	3.7	1.6	6.9	2.4	4.8
write	53.0	68.7	11.1	32.4	4.5	3.7	0.5	3.8	2.1	4.3
openat	19.1	11.1	37.9	5.5	5.6	3.6	8.1	6.4	4.9	4.8
accept	32.4	32.4	5.5	34.3	3.4	3.6	0.6	2.8	0.9	3.0
exit_group	4.9	4.5	5.6	3.4	13.7	2.9	0.8	3.9	1.9	5.6
clone	3.7	3.7	3.6	3.6	2.9	11.0	0.1	2.6	0.4	2.9
readlink	1.6	0.5	8.1	0.6	0.8	0.1	8.6	0.0	1.5	0.0
epoll_wait	6.9	3.8	6.4	2.8	3.9	2.6	0.0	8.4	1.4	3.8
futex	2.4	2.1	4.9	0.9	1.9	0.4	1.5	1.4	7.7	1.3
madvise	4.8	4.3	4.8	3.0	5.6	2.9	0.0	3.8	1.3	7.7

(a) Redis



	sendfile64	recvfrom	accept4	openat	writev	setsockopt	newstat	mkdir	getdents	write
sendfile64	82.8	48.4	48.4	22.0	35.2	32.2	20.7	20.3	19.5	16.8
recvfrom	48.4	53.0	46.4	11.9	32.9	31.0	10.1	10.1	9.7	5.1
accept4	48.4	46.4	50.2	11.3	34.4	31.8	9.9	9.7	8.8	5.0
openat	22.0	11.9	11.3	40.1	10.5	9.8	28.9	27.9	21.3	12.0
writev	35.2	32.9	34.4	10.5	37.5	29.8	9.7	9.5	9.1	5.6
setsockopt	32.2	31.0	31.8	9.8	29.8	34.5	9.2	9.0	8.6	4.2
newstat	20.7	10.1	9.9	28.9	9.7	9.2	30.2	27.1	19.0	10.8
mkdir	20.3	10.1	9.7	27.9	9.5	9.0	27.1	28.6	19.4	11.0
getdents	19.5	9.7	8.8	21.3	9.1	8.6	19.0	19.4	23.1	9.9
write	16.8	5.1	5.0	12.0	5.6	4.2	10.8	11.0	9.9	22.8

(b) NGINX

Table 2: Instruction overlap across system calls for Redis and Nginx configurations. Numbers represent thousands of instructions. Colors represent the intersection size relative to the overall number of instructions used by the *row* system call. Diagonal represents the instruction coverage of each system call. Only the highest coverage system calls for each configuration are shown.

Type	Redis	NGINX
Reachable	0.49%	0.60%
Potentially reachable	45.52%	44.35%
Unreachable	53.99%	55.05%

Table 3: The number of kernel instructions in each of the three classes (Figure 2). The profiling details are provided in §9.3.

clients. The clients access the service by sending requests to a client-facing application (i.e., *untrusted application*), such as a web server or database application, installed on the service provider's machine. However, the service provider does not trust their clients. There can be many reasons for clients to attack the system, such as stealing information related to other clients, taking control of the machine to corrupt the service, compromising other services on the same machine, or hiding evidence of attacks. We assume that controlling the client-facing application process is not enough because the application is sandboxed (e.g., Native Client [60], Linux containers [45]), hence, the adversary needs to control the system's *kernel* to attack the provider. For presentation purposes, we assume only one client-facing application but SHARD works with groups of applications as well.

Trusted applications. The service provider may also need to run trusted supporting services (e.g., back-end encryption engine for a database) on the same machine that do not accept input from the adversarial clients and are sandboxed from direct attacks by the untrusted application.

Kernel. We assume the service provider has access to the source code of the kernel; therefore, they can statically and dynamically analyze the kernel and instrument it.

6 Threat Model

Attacker Capabilities. An adversary may control all client-facing applications and the libraries used by these applications to mount attacks against the kernel. In particular, the adversary may invoke any system call, using any parameters and at any time, from client-facing applications.

The adversary is capable of launching control-flow hijacks against the system's kernel. Such attacks redirect the program's control-flow to an arbitrary location by reusing the code in the memory (i.e., system kernel in our case). The requirements [51] for such attacks are (a) the existence of an out-of-bounds or dangling pointer vulnerability that can overwrite a code pointer, such as a function pointer or return address and (b) the ability to execute an exploit payload (e.g., through ROP [49] or JOP gadgets [23]).

Kernel Assumptions. The system kernel is benign (i.e., written by honest developers) but may contain bugs (e.g., memory-safety violations). We make the following standard CFI assumptions [29] about the kernel:

- Kernel uses NX protection or similar [2] to prevent writes to kernel executable memory, thus code-injection attacks are not possible unless protections are disabled.
- The kernel boots in a trusted state, therefore, the initial kernel image is not corrupted and does not contain malicious code.

Out-of-scope. We assume that the SHARD framework and the hardware is trusted and beyond the control of the adversary. Side-channel attacks (e.g., cache attacks) and micro-architectural leaks, although important, are not specific to the kernel. Furthermore, such channels could be disabled by firmware patches [13] or software solutions [63]. Finally, the adversary does not have physical access to the machine, therefore, hardware attacks are out-of-scope.

7 Design of SHARD

This section provides a description of SHARD including a design overview (§7.1) and a description of the offline analysis (§7.2), kernel instrumentation (§7.3), and run-time kernel specialization and hardening enforcement (Figure 7.4).

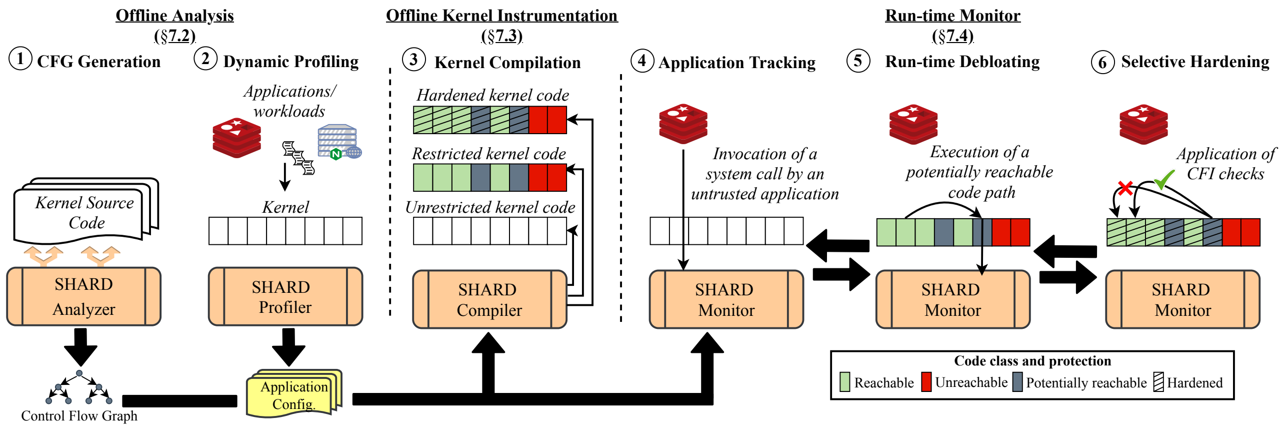


Figure 3: Workflow of SHARD.

7.1 Overview

The SHARD framework consists of an offline analysis phase to generate specialized configurations for each target application and an online phase that enables, during run-time, kernel (debloating) specialization and context-aware hardening based on the generated configurations. Figure 3 demonstrates the workflow of SHARD.

During the offline analysis (①∼②), SHARD first creates a static control-flow graph of the kernel to identify the *unreachable* code for each system call (①). Then, SHARD dynamically profiles the target application to identify the kernel code required and commonly used by the application, i.e., *reachable* code (②). The remaining kernel code is labeled *potentially reachable*. Using the offline analysis information, SHARD creates per-application *configurations* and instruments the kernel code (③) for the online phase.

During the online phase (④∼⑥), SHARD installs a VMX security monitor to enforce specialization policies. The SHARD monitor performs three tasks: (a) track the context switches involving the untrusted application and its system call invocations, (b) specialize the kernel-view of the untrusted application on each system call, and (c) implement kernel context-aware hardening using control-flow integrity [20] during a system call if, and when, it executes *potentially reachable* code.

SHARD detects context switches to and from the target application and system call invocation using lightweight kernel instrumentation on context switch functions and system call handlers (④). On each system call invoked by the untrusted application, SHARD transparently replaces the kernel's code pages based on the application's configuration, as determined by the offline analysis (⑤). This step debloats the kernel (i.e., disables the *unreachable* code) and allows SHARD to detect kernel transitions to *potentially reachable* code. On detecting a transition to *potentially reachable* code, SHARD similarly replaces the kernel's code pages with hardened versions (⑥). Finally, when the kernel execution returns from *potentially reachable* code to *reachable* code, SHARD replaces the hardened code pages with the previous specialized code pages.

7.2 Offline Analysis

This section describes how SHARD generates a target application's kernel configuration, which outlines the kernel code required by the application on a per-system call basis. To strike a balance between efficiency and effectiveness, SHARD generates configurations using function-level granularity, i.e., SHARD determines which kernel functions are executed for a given system call. SHARD implements two main analysis stages: (a) static control-flow graph generation and (b) dynamic profiling using application workloads.

Static control-flow graph generation. SHARD statically analyzes the kernel to create a control-flow graph (CFG) of the kernel. In particular, the CFG differentiates the *reachable* + *potentially reachable* kernel code from the *unreachable* kernel code for each system-call. Note that while the CFG over-estimates the *potentially reachable* code (as mentioned in §2.1), it is sound when it determines that code is unreachable. The CFG is generated once per-kernel version, regardless of the target application.

SHARD leverages a two-layered type analysis algorithm [42] to generate the CFG. This two-layered analysis exploits the kernel's extensive use of `struct` types for function pointer storage, to significantly increase precision over previous approaches [55]. It matches indirect call sites which load function pointers from a field within a `struct`, to functions stored to that field of that `struct` for precisely identifying the number of potential targets for the call site. In addition, SHARD also uses the generated control-flow graph to enforce control-flow integrity in context-aware hardening (§7.4).

Dynamic profiling using application workloads. SHARD executes the target applications using representative workloads (e.g., benchmarks or test suites) to identify the *reachable* kernel code during each invoked system call.

The dynamic profiling takes place in a benign environment. SHARD uses lightweight compile-time instrumentation

```
SyS_sysinfo:
    ; shadow stack prologue
    SUB $8, %gs:0       ; Increment shadow stack
    MOV %gs:0, %rax     ; Copy return address
    MOV (%rsp), %rcx    ; into current shadow
    MOV %rcx, (%rax)    ; stack
    ...
    ...
    ; shadow stack epilogue
    MOV %gs:0, %rcx    ; Compare current
    CMP %rcx, (%rsp)   ; shadow stack against
    JNE abort          ; return address
    ADD $8, %rcx
    MOV %rcx, %gs:0    ; Decrement shadow stack

    CMP %fs:0, %rcx    ; Check If we should
    JNE ret            ; disable CFI
    UD2                ; Exit to KVM
    ret:
        RET            ; Allow return
    abort:
        UD2            ; Kill process
```

Figure 4: SHARD's instrumentation for the shadow stack. `shadow_stack` refers the `%gs` register which is randomized on each hardening instance. The base of the shadow stack is stored in the `%fs` register to check if the shadow stack is empty.

to generate a kernel version that supports offline dynamic profiling. The instrumentation ensures that the kernel traps, on each kernel function (not previously-logged for a certain system call) when the untrusted application executes, into SHARD's profiler (using UD2 instructions) which executes in VMX root mode. Hence, SHARD can record the (a) system calls invoked by the application, and (b) kernel functions used by the system calls.

SHARD labels exception and interrupt handlers as *reachable* code, for each system call, since they might execute at any time. SHARD determines the exception and interrupt handler coverage the same way it determines system call coverage. Since SHARD relies on compile-time instrumentation, our current implementation does not specialize kernel code written in assembly and hence considers it *reachable*.

7.3 Offline Kernel Instrumentation

After analysis, SHARD compiles three versions of each kernel code page, UNRESTRICTED, RESTRICTED, and HARDENED, using the unmodified kernel's source code. The UNRESTRICTED version (§7.3-(a)) enables all kernel functions and is used only by trusted applications. The RESTRICTED version (§7.3-(b)) enables only the *reachable* code relative per-system call. The HARDENED version (§7.3-(c)) contains both the *reachable* and *potentially reachable* code, and is shown only to untrusted applications. Furthermore, SHARD ensures that functions are address-aligned across the three versions of code pages by padding them with NOP instructions. Therefore, different versions of the same code page are interchangeable without impairing the kernel's correctness.

UNRESTRICTED code pages. The system runs various applications that are trusted (refer to §5). Therefore, SHARD compiles UNRESTRICTED code pages that do not restrict or harden the kernel's code to allow native execution of trusted

```
check_cfi:
    MOV 0x10(%rdi), %rax      ; Load pointer into RAX
    MOV  %rax, %rcx
    SHR $0xc, %rcx            ; Move frame number into RCX
    CMP $0x7ff,%rcx           ; Check if
    JA abort                  ; frame >= TOTAL_FRAMES
    MOV  $TAB(,%rcx, 8), %rcx ; Move table[frame] to RCX
    TEST %rcx, %rcx           ; Check if table[frame]
    JE abort                  ; is set
    MOV %rax, %rdx
    AND $0XFFF, %edx          ; Move offset into RDX
    CMP 0x0, (%rcx, %rdx, 1)  ; Check if table[frame]
    JE abort                  ; [offset] is set
    CALLQ *%rax               ; Make indirect call
    ..
    abort:
        UD2
```

Figure 5: SHARD's CFI instrumentation at indirect call sites.

applications.

However, UNRESTRICTED code pages are still minimally instrumented to track context switches to untrusted applications as well as padded with NOP instructions to align code with the RESTRICTED and HARDENED versions. In particular, SHARD instruments the kernel's (a) context switch function (e.g., `__switch_to` in Linux) and (b) common system call handler (e.g., `do_syscall_64` in Linux), to notify its runtime monitor when untrusted applications execute and invoke a system call, respectively. The notification of system calls is enabled only during the execution of untrusted applications.

RESTRICTED code pages. Based on SHARD's dynamic profiling (§7.2-(b)), SHARD compiles RESTRICTED frames for each system call required by the untrusted application. Such code pages contain only the *reachable* kernel functions required by a specific system call invoked by the application, while the remaining code (i.e., *potentially reachable* and *unreachable*) is replaced with undefined (UD2) instructions.

HARDENED code pages. SHARD compiles HARDENED code pages with both *potentially reachable* and *reachable* code enabled and hardened. These code pages are used when SHARD detects during runtime the execution of *potentially reachable* code. Since, such execution is possibly malicious, SHARD ensures that all enabled kernel code, i.e., both *reachable* and *potentially reachable*, is hardened until the execution returns from the *potentially reachable code path*. SHARD requires a single HARDENED version of each kernel code page (unlike RESTRICTED versions which are application and system call-specific) since the hardening checks (explained below) protect the execution within the kernel, irrespective of the system call and application.

SHARD enforces control-flow integrity (CFI) in HARDENED code pages, ensuring all control flow transfers adhere to the control-flow graph (CFG) generated in §7.2-(a). Importantly, unlike prior system [39], SHARD ensures fine-grained CFI by checking whether the destination of an indirect control-flow transfer is valid from that specific code location. In particular, SHARD enforces CFI on forward indirect control-flow transfers using a technique that is based on Restricted Pointer Indexing (RPI) [29, 56], while protecting backwards return transfers using the shadow stack.Hence, SHARD's hardening prevents both ROP and JOP attacks.

Note that SHARD's contribution isn't the hardening mechanisms or implementations, which are from existing work. In particular, SHARD's contribution lies in the efficient, context-aware application of hardening mechanisms. The following paragraphs provide details about SHARD's instrumentation related to RPI and shadow stack.

Restricted Pointer Indexing (RPI). Traditional RPI uses integer-based indexing into a call target table (refer to [29] for details) for indirect control-flow transfers. However, such indexing would raise compatibility issues when passing function pointers from UNRESTRICTED and RESTRICTED to HARDENED code pages, because the former use function addresses (natively used by the compiler). A naive solution would be to modify RESTRICTED and UNRESTRICTED pages to use integer-indexing as well. However, such approach would incur considerable overhead, up to 40% [29], for code pages that otherwise would execute at near-native speed. Therefore, SHARD uses a modified version of RPI which uses function addresses to ensure that non-hardened code versions are not impacted.

Figure 5 illustrates SHARD's RPI instrumentation to enforce control-flow integrity at indirect call sites. In particular, SHARD maintains two call target tables for reference. Each valid target address from an indirect call site contains a corresponding reference in a first call table, which references an entry in a second call table. The call target tables are populated with valid targets using the kernel's control-flow graph (generated in §7.2-(a)) and then marked as *read-only* to avoid tampering at run-time.

The first target table contains an entry for each kernel code frame (i.e., 2048 entries in Linux's case), indicating if a branch to the target kernel code frame is allowed or not, from the indirect call site. The second table contains an entry for each offset in a frame (i.e., a 4 KB frame has 4096 offsets), indicating whether a branch to such an offset of the kernel frame is allowed or not. On each indirect control transfer, the instrumentation asserts that the corresponding entries exist in both tables, otherwise, the control-flow does not follow CFI and the program is terminated by SHARD.

Shadow stack. Shadow stack stores a backup copy of the stack to prevent an adversary from returning to a different address during execution. Each program thread is allocated a separate shadow stack. SHARD uses randomization to hide the shadow stacks and prevent malicious modification. In particular, SHARD uses the segment register (`%gs`) to randomize the shadow stack [25] on each context-aware hardening. However, randomization-based shadow stack protection is not fundamental to SHARD's design; hence, other techniques (e.g., memory protection [57]) can be adopted by SHARD.

Figure 4 shows SHARD's instrumentation for the shadow stack. At the start of each function, SHARD's instrumentation stores the return address in the shadow stack. Then, on the subsequent return, the instrumentation asserts whether

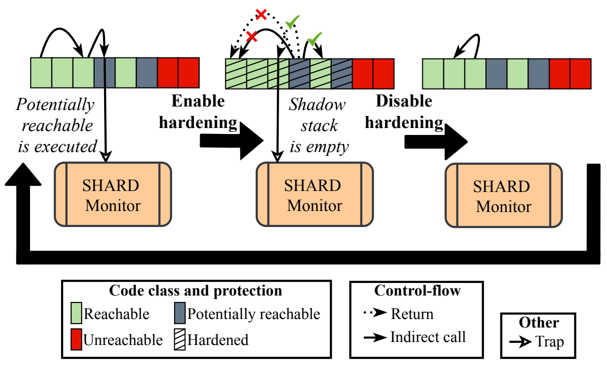


Figure 6: SHARD context-aware hardening cycle.

the return address stored in the native stack and the shadow stack's return address are consistent. Furthermore, SHARD's instrumentation tracks when the shadow stack is empty, i.e., the *potentially reachable code path* has completed execution. In particular, on each return, the instrumentation checks whether the shadow stack pointer is pointing towards the shadow stack's base (stored in `%fs` segment register). If yes, the instrumentation triggers a `UD2` to inform SHARD.

Shadow stack implementation using segment registers can potentially suffer from time-of-check-to-time-of-use (TOCTTOU) attacks [28], i.e., the return address is correct at the time of validation but is modified before the return instruction. However, an extensive study on shadow stacks [24] suggests that exploiting this race is non-trivial since it requires highly precise timing. Nevertheless, mitigations exist against this problem [24], at slightly higher performance costs.

7.4 Run-time Monitor

During the online phase, the SHARD monitor executes in VMX root mode to track the execution of untrusted applications, as well as enforce debloating and context-aware hardening.

Workflow. SHARD monitor operates in four major stages.

1. Initial kernel-view. The SHARD monitor enables the UNRESTRICTED version for all kernel code pages to allow the unrestricted execution of trusted applications and detect the execution of the untrusted application.

2. Debloating enforcement. The monitor is notified through kernel instrumentation on (a) context switches to untrusted applications and (b) system call invocations by untrusted applications. On system call invocations by the untrusted application, the monitor switches all kernel code pages to RESTRICTED, based on the specific system call and application configuration (generated in §7.2) to enforce debloating by allowing only *reachable* code to execute.

3. Hardening enforcement. During the execution of RESTRICTED pages, a triggered `UD2` signals that the kernel tried to execute an *unreachable* or *potentially reachable* kernel code. If the kernel tried to execute *unreachable* code, the monitor terminates the application since such execution cannot be legitimate. On the other hand, if the attempt was towards a *potentially reachable code path*, SHARD enforces context-aware hardening by (a) implementing an initial CFI check using the CPU Last Branch Record (LBR) and (b) switching the kernel-view to HARDENED (illustrated in Figure 6). The initial CFI check ensures that the first control-flow transfer from *reachable* to *potentially reachable* is valid.

4. Disabling hardening. Lastly, the monitor disables hardening, i.e., switches from HARDENED code pages to RESTRICTED, when the system returns to the *reachable* code from where it triggered the hardening (refer to the shadow stack implementation in Figure 4).

Transparent and efficient kernel-view switch. The SHARD monitor uses a VMX feature, Extended Page Tables (EPT) [32], to achieve transparent and efficient switching between different versions of the kernel code pages. In particular, the monitor uses the EPT to redirect the guest memory view of the system from one (host) physical page to another.

Since the kernel is huge and spans many code pages (e.g., 2048 code pages in our Linux kernel), updating the EPT entries individually for each page would be costly. Therefore, SHARD updates the EPT at the page directory-level, i.e., 512 pages at once, to change the kernel-view. For efficiency, during initialization, the SHARD monitor statically creates page tables for each system call using the configuration of each application (generated in §7.2). Then, on system call invocations, the monitor updates the page directory entries to point towards these already-crafted page tables.

LBR-based control-flow integrity check. At an `UD2`-trap, during the execution of RESTRICTED code pages, although SHARD changes the code versions to HARDENED, the current control-flow transfer (that raised the trap) would be unprotected without an additional check. In particular, while HARDENED code page versions enforce CFI during their execution, SHARD should enforce the same while the system transitions from RESTRICTED to HARDENED versions.

Therefore, the SHARD monitor implements a custom CFI check for such control-flow transfers using the Last Branch Record (LBR). In particular, the LBR stores information about the 32 most recent *taken* branches by the processor [32]. The stored information includes the source and target addresses of the branches. Using this information, SHARD ensures that the control-flow transfer's target address is a valid target for its source (using the CFG generated in §7.2-(a)). If it is not, SHARD terminates the program, otherwise, switches the RESTRICTED versions of the kernel's code pages to their HARDENED versions.

8 Implementation

SHARD's implementation consists of a static analyzer, a dynamic profiler, an LLVM instrumentation pass, and a run-time monitor. Table 4 lists the lines of source code for each component of the implementation. SHARD's source code is available at https://github.com/rssys/shard.

Component	Lines of code
Static analyzer	2047
Dynamic profiler	171
Offline kernel instrumentation	822
Run-time monitor	1842
Total	4882

Table 4: SHARD components' lines of code.

The static analyzer uses the two-layer type analysis algorithm [42], which, to the best of our knowledge, is the current state-of-the-art in kernel CFG generation. The analysis algorithm divides indirect calls based on whether they load function pointers from a `struct` or not. For the former case, all call pointers loaded from a particular field of a structure are matched with all functions stored to that field. Such functions are identified using taint analysis. For the latter, the analysis uses traditional signature-matching approach [54]. The static analyzer resolves an indirect call site to 7 targets, on average, in the kernel's CFG. Using the CFG, we create and populate control-flow integrity target tables (refer to §7.3-(c)). On average, we require only two tables (i.e., a frame table and an offset table) for each indirect call site in our kernel.

Furthemore, we wrote an LLVM-5 [41] instrumentation pass to instrument the kernel and create different types of code page versions (refer to §7.3). It supports the Linux kernel v4.14 with modules built-in and can be extended to work on any kernel that compiles to the LLVM IR (e.g., BSD). It can also be extended to work for dynamically-loaded modules, similar to prior work [30].

Finally, we implement the dynamic profiler (refer to §7.2) and the run-time monitor (refer to §7.4) in the KVM module. The run-time monitor reserves a random 400KB memory region within the guest for shadow stacks. The reserved region should be configured based on the maximum number of threads that the target program executes (i.e., 1KB for each thread's shadow stack). Note that SHARD also randomizes the base of a shadow stack (on each hardening instance); hence, an attacker must continuously guess the shadow stack's location, even if they guess the base address of the reserved memory. Please refer to existing sources [24, 64] for a full entropy analysis of randomization-based shadow stack protection, as well as its limitations and other approaches.

9 Security Evaluation

SHARD's goal is to restrict the attacker capabilities to conduct control-flow hijacks by reducing the amount of kernel code exposed and employing context-aware hardening through CFI. Therefore, we quantify and provide an analysis of SHARD's attack surface in §9.1. Furthermore, we analyze the number

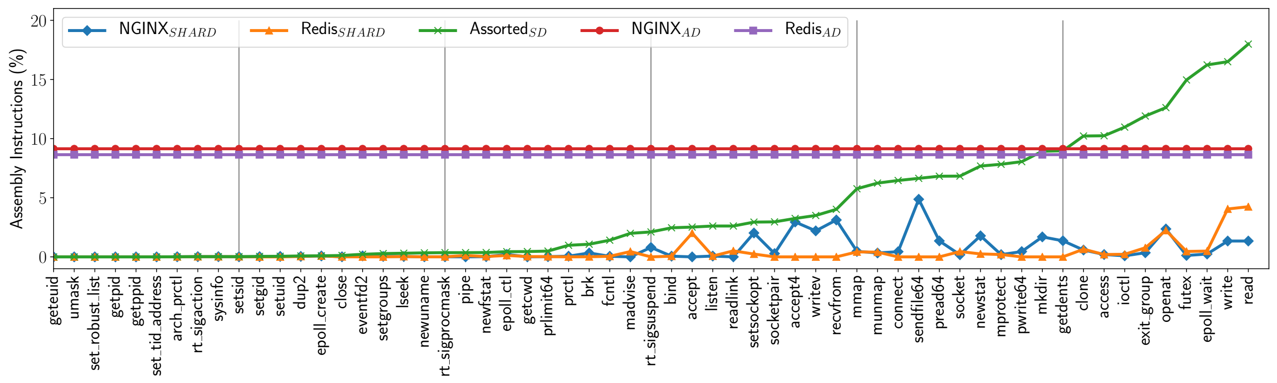


Figure 7: Attack surface reduction (debloating) across system calls. Numbers represent the assembly instructions available relative to the native kernel for each system call. XYZ_{SHARD} and XYZ_{AD} refers to application XYZ running with SHARD and application-only (existing) debloating respectively. *Assorted*$_{SD}$ refers to system call-only debloating using NGINX, Redis, and LTP workloads.

of ROP and JOP gadgets exposed by SHARD in §9.2. Finally, we show how SHARD's reduced attack surface and hardening prevents actual kernel attacks in §9.3.

9.1 Attack Surface Reduction

SHARD restricts the attack surface to the *reachable* code. In particular, SHARD disables the *unreachable* code at every system call, while it hardens the *potentially reachable* code through control-flow integrity (CFI).

In the following, we show the attack surface in terms of *reachable* assembly instructions. Furthermore, we compare SHARD's exposed attack surface against both existing application-only kernel debloating (i.e., debloating at the level of each application and not system call) and system call-only debloating (i.e., debloating at the level of each system call and not application).

Setup and methodology. We use two popular real-world applications, NGINX [16] web server and Redis [17] key-value store. To dynamically profile these applications, we used the `ab` [1] and `redis-benchmark` suites, respectively. In particular, we used `ab` with a range of file sizes from 1KB to 128KB and `redis-benchmark` with default settings.

We determine the attack surface of application-only kernel debloating (*NGINX*$_{AD}$ and *Redis*$_{AD}$ in Figure 7) through dynamic profiling of the test applications. Furthermore, to estimate the attack surface of system call-only debloating (*Assorted*$_{SD}$ in Figure 7), we calculate the upper bound of the kernel code required for each system call by combining the dynamic profiles of NGINX, Redis, and the Linux Test Project (LTP) [40]. Note that our assorted workload might not consider all kernel functions required by each system call, however, we expect that it provides a good approximation.

Finally, we determine the attack surface of SHARD (*NGINX*$_{\text{SHARD}}$ and *Redis*$_{\text{SHARD}}$ in Figure 7) by determining the *reachable* code at each system call through dynamic profiling of the test applications.

Results. Figure 7 shows the number of instructions of assembly code, differentiated by each system call invoked by the test applications. Our analysis reveals that for half the system calls, SHARD exposes between $0-0.2\%$ of assembly instructions in the Linux kernel. Even in the worst case, only 4.87% of the kernel's instructions are available to the attacker.

In contrast to SHARD, the coarse debloating employed by previous (application-only) kernel debloating systems, reveals a constant and large attack surface, which represents the cumulative sum of all kernel code that an application requires during execution. Furthermore, while system call-only debloating alternative performs similar to SHARD for simpler system calls such as `setuid`, `dup2`, which only execute a few internal kernel functions, it performs much worse for more complex system calls (e.g., `read`). The reason is that complex system calls implement multiple functions, using many kernel functions, most of which are not required by a specific application.

9.2 ROP and JOP Gadget Analysis

This section analyzes the ROP and JOP gadgets exposed by SHARD as well as system call and existing application-only kernel debloating approaches. Similar to assembly instructions, SHARD only allows the attacker to construct ROP and JOP gadgets using *reachable* code. Note that reduction in ROP and JOP gadgets is not a comprehensive metric for reduction in attacks since a few gadgets are enough for meaningful exploits [58,59]. Nevertheless, such analysis aids in SHARD's comparison with existing approaches [21,31,36,46,47,50] that also provide such gadget analysis.

Setup and methodology. The evaluation setup, methodology, and applications are the same as §9.1.

Results. Table 5 shows the absolute number of ROP and JOP gadgets exposed under all applications and debloating types considered, across system calls. On average, SHARD shows a reduction (compared to the native Linux kernel) of 149× and

	Min	Max	Median	Avg	Factor
Native kernel	339017	339017	339017	339017	1×
NGINX$_{AD}$	33614	33614	33614	33614	10×
Redis$_{AD}$	32090	32090	32090	32090	11×
Assorted$_{SD}$	0	67260	8783	15757	22×
NGINX$_{\text{SHARD}}$	0	16689	440	2273	149×
Redis$_{\text{SHARD}}$	0	14605	519	1854	183×

Table 5: ROP and JOP gadgets exposed by SHARD and other approaches across system calls. Only systems that specialize across system calls have non-constant values. All numbers were obtained using the ROPGadget tool [12]. Factor refers to the ratio between the native kernel and the system average.

183× considering NGINX and Redis, respectively, which is an order of magnitude better than existing application-focused and system call-only debloating.

9.3 Attack Evaluation and Analysis

This section describes how SHARD prevents control-flow hijacks, which require kernel vulnerabilities and exploit payloads, through an attack analysis.

Setup and methodology. We consider five diverse exploit payloads which have previously been evaluated by others [35, 43, 58, 59]. Furthermore, we randomly selected a list of Linux vulnerabilities.

Table 6 provides an overview of the exploit payloads (P_1 - P_5). P_1 elevates the privileges of a user process, giving root privilege to the process. P_2 disables the separation between kernel and user processes, which allows an adversary to execute user code in kernel space. Lastly, P_3, P_4. and P_5 allow the attacker to inject malicious code in the kernel by disabling NX protections, i.e., make writable memory executable or executable memory writable.

Table 7 provides an overview of the list of vulnerabilities considered (V_1 - V_{10}). These vulnerabilities include out-of-bounds access such as buffer overflows, use-after-free access for a dangling pointer, and double-free issues. These vulnerabilities are caused by kernel bugs in a diverse set of kernel functionality, including the ext4 file system, keyring facility, block layer, and networking module.

Finally, we use the same test applications (mentioned in §9.1) for attack evaluation.

Attack analysis. SHARD can prevent the execution of 4 out of the 5 considered payloads, for the NGINX and Redis configurations. In particular, P_3, P_4, and P_5 are either completely disabled (i.e., in *unreachable* code) or hardened using CFI (i.e., in *potentially reachable* code). SHARD also prevents P_1, which requires the execution of two kernel functions in succession, `prepare_kernel_cred` which creates root credentials and `commit_creds` which commits the credentials to grant the application root access. However, only `commit_creds` is *reachable* (in system calls `setuid`, `setgid`, and `setgroups`)

Payload	Dependencies	Protection		Prevented
		Unr	Hard	
P1: Privilege eleva. [58]	`commit_creds`, `prepare_kernel_cred`	✓	✗	✓
P2: Disable SMAP [59]	`native_write_cr4`	✗	✗	✗
P3: Set memory exec. [48]	`set_memory_x`	✓	✗	✓
P4: Set memory writ. [43]	`set_memory_rw`	✗	✓	✓
P5: Modify page table [43]	`lookup_address`	✓	✗	✓

Table 6: SHARD's protection against exploit payloads. "Unr" stands for *unreachable* and "Hard" represents *hardening*. For P_1, only `prepare_kernel_cred` is *unreachable* but since the exploit requires both functions, we classify it as *unreachable*.

while running NGINX or Redis. While an attacker can recreate the credentials using ROP gadgets, it would be very challenging because SHARD exposes few ROP gadgets (i.e., 175, 118, and 207, respectively) for these system calls. Finally, SHARD cannot prevent the execution of P_2 because it depends on `native_write_cr4`, a function required by interrupt handlers and, therefore, *reachable* from every system call (as mentioned in §7.2).

Regarding vulnerabilities, SHARD disables 5 out of 10 vulnerabilities considered because they are located in *unreachable* code for these applications. The remaining 5 vulnerabilities can be triggered since they exist in *reachable* or *potentially reachable* code. However, they cannot always be exploited as we explain in the next paragraph.

Considering control-flow hijacks, which require both a vulnerability *and* an exploit payload (as explained in §6), an attacker can attempt 50 concrete attacks using the considered 5 payloads (P_1 - P_5) and 10 vulnerabilities (V_1 - V_{10}). Because SHARD can prevent hijacks by either disabling the vulnerability or the exploit, SHARD prevents 90% (45 out of 50) of the attacks. In particular, SHARD is only susceptible to attacks using the payload P_2 and the exposed 5 vulnerabilities (V_3, V_6, V_8, V_9, and V_{10}), as both the payload and the vulnerabilities are *reachable* in these applications.

Our analysis indicates that SHARD can invalidate many exploit payloads and vulnerabilities, hence, it is highly effective at thwarting control-flow hijacks, despite low overhead (§10).

Defense validation. To validate our analysis, we attempted six control-flow hijacks using NGINX and Redis. For this, we used the exploit payload, P_1, and three vulnerabilities namely CVE-2016-0728 [18], CVE-2017-5123 [7], and CVE-2017-7308 [8]. We attempted each control-flow hijack by both overwriting a function pointer and a return address, i.e., six attacks in total. SHARD successfully prevented all six attacks because the payload was *unreachable* for both application; hence, jumps to the payload were caught by SHARD.

10 Evaluation

This section describes the experimental setup for SHARD (§10.1), evaluates its overhead through micro-benchmarks

CVE	Vulnerable Function	Unr	Prevented				
			P_1	P_2	P_3	P_4	P_5
V1: 2016-0728 [18]	`join_session_keyring`	✓	✓	✓	✓	✓	✓
V2: 2017-5123 [7]	`SyS_waitid`	✓	✓	✓	✓	✓	✓
V3: 2017-7308 [8]	`packet_set_ring`	✗	✓	✗	✓	✓	✓
V4: 2017-10661 [3]	`SyS_timerfd_settime`	✓	✓	✓	✓	✓	✓
V5: 2017-11176 [4]	`SyS_mq_notify`	✓	✓	✓	✓	✓	✓
V6: 2017-17052 [5]	`get_net_ns_by_id`	✗	✓	✗	✓	✓	✓
V7: 2018-7480 [10]	`blkcg_init_queue`	✓	✓	✓	✓	✓	✓
V8: 2018-10880 [6]	`ext4_update_inline_..`	✗	✓	✗	✓	✓	✓
V9: 2018-17182 [9]	`vmacache_flush_all`	✗	✓	✗	✓	✓	✓
V10: 2019-20054 [11]	`ext4_xattr_set_entry`	✗	✓	✗	✓	✓	✓

Table 7: SHARD's effectiveness against control-flow hijacks attacks using different vulnerabilities and payloads (Table 6). The code of some vulnerabilities is *unreachable* ("Unr").

(§10.2) and real-world applications (§10.3), and evaluates the impact of profiling accuracy (§10.4).

10.1 Experimental Setup

Machine specification. We conducted all our experiments on an Intel (R) Core (TM) i7-6500U CPU @ 2.50GHz with 4 MB of last-level cache, 8 GB of memory, and support for the Last Branch Record (LBR).

Kernel configuration. Our SHARD-protected kernel was Linux kernel v4.14, which ran inside a guest virtual machine (VM). The VM was allocated 4 GB of memory, 1 thread, and connected to the host with a 1 Gb/s virtual connection.

SHARD configuration. SHARD's monitor was installed on the KVM module of the host, running Linux kernel v4.15.

10.2 Micro-benchmarks

This section analyzes the memory footprint of SHARD and the overhead of SHARD monitor's operations.

Memory footprint. SHARD maintains various versions of instrumented kernel code pages (i.e., UNRESTRICTED, RESTRICTED, and HARDENED) and call target tables to enforce control-flow integrity (CFI) (refer to §7.2). Table 8 shows the memory overhead incurred by SHARD. Each application incurs a different overhead for RESTRICTED code page versions, based on the invoked system calls and kernel functions. The main memory overhead is caused by call target tables, maintained for each indirect kernel call site, to enforce CFI. Nevertheless, this memory consumption is negligible in comparison with the memory available in modern machines (usually tens of GBs).

Monitor overhead. The SHARD monitor performs 3 operations (refer to §7.4): (a) trap on context switches and system calls, (b) switch the EPT to enforce hardening and debloating, and (c) perform an LBR-based check for CFI during hardening. To ascertain the runtime overheads, we create a benchmark which executes a system call (i.e., `getgid`) in

Component	Required memory (MB)
Kernel code pages	
UNRESTRICTED	8.0
RESTRICTED (NGINX, Redis)	14.4 – 18.0
HARDENED	8.0
CFI tables	
Frame table	14.0
Offset table	34.0
Total	78.4 – 82.0

Table 8: Memory footprint of SHARD.

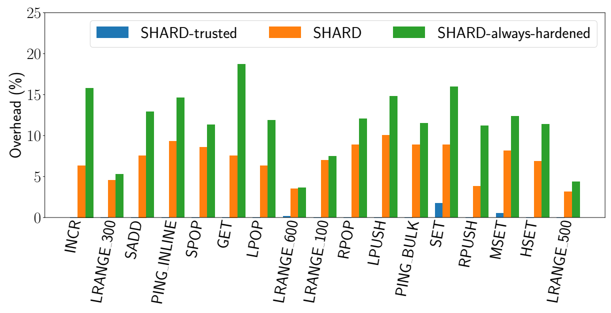


Figure 8: Performance overhead of `redis-benchmark`.

a loop for 10 million iterations. This is lightweight system call that only takes 0.43 μs on average to execute in the native kernel. We measure how long it takes for the benchmark to complete, while selectively enabling each operation, and comparing it against the native (non-monitored) execution.

Our results show that a trap at each system call adds an average overhead of 1.21 μs per-system call. Furthermore, switching the EPT involves updating 4 page directory entries (since our kernel is 8 MB and a page directory holds 2 MB of pages) and the `INVEPT` instruction, which adds 0.60 μs. Also, the SHARD monitor implements a CFI-check using LBR, which requires referencing the two call target tables and retrieving the latest entry in the LBR, taking 1.01 μs on average.

10.3 Real World Applications

This section evaluates SHARD's overhead while executing real-world widely-deployed applications, NGINX web server and Redis key-value store, that match our use-case scenario (refer to §5). Furthermore, we also evaluate SHARD with a well-known set of real-world workloads, SPEC CPU 2006.

Common settings and terminology. We profiled each application using the experiment workload. The client-server experiments (NGINX and Redis) were performed by sending requests from clients on the host machine. We ran each experiment 10 times and report the average overhead compared to a native (uninstrumented) Linux kernel.

In Figure 8 and Figure 10, "SHARD-trusted" refers to scenarios where SHARD does not enforce debloating or hardening (i.e., for trusted applications), "SHARD " means SHARD's overhead while enforcing debloating and context-aware hard-

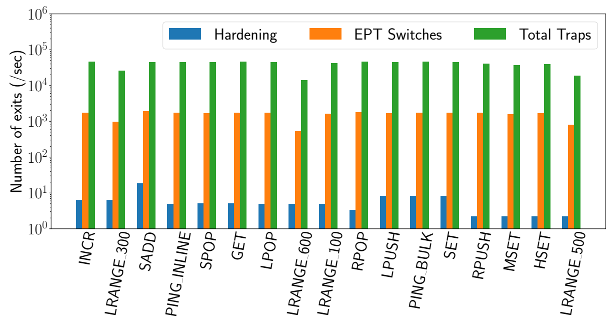


Figure 9: SHARD statistics while running `redis-benchmark`.

ening, and "SHARD-always-hardened" means SHARD's overhead while enforcing debloating and full-hardening on each system call. Note that SHARD-always-hardened can only be realized using SHARD's framework, i.e., it is not existing work, and is included for performance comparison.

Figure 9 and Figure 11 illustrate the overall statistics for SHARD, including number of exits and EPT switches (for debloating or context-aware hardening), related to NGINX and Redis, respectively.

Redis key-value store. We evaluate Redis using the official `redis-benchmark`. The benchmark ran with the default configuration, sending requests from 50 concurrent clients.

Figure 8 shows the overheads for the `redis-benchmark` tests. The average overhead across all the tests for SHARD is 6.83%. Considering the execution statistics (Figure 9), we notice more than 40,000 traps per-second in some tests. However, since the application invoked the same system calls (i.e., mostly `read` and `write`) successively, 96.15% of these traps did not require switching the EPT (for debloating or hardening). Switching the EPT requires invalidation of the instruction cache, which is costly to repopulate. Due to few such cases, the overhead remains low. Additionally, we noticed 29 average instances of hardening per-second. However, their overall impact on the execution was low since hardening was only enforced for small durations.

Moreover, SHARD-always-hardened incurs an additional overhead of 0.1-11% over SHARD (average increases to 11.49%). In particular, we observe a high overhead when the benchmark application invokes many system calls in a small span of time (e.g., for INCR and GET). In contrast, benchmark applications (e.g., LRANGE) that execute for longer periods and invoke system calls less frequently, exhibit less overhead for full-hardening. Finally, while running Redis as a trusted application (SHARD-trusted), we only observe an average overhead of 1.2%, because SHARD did not trap its execution. The negligible overhead is due to the lightweight instrumentation of UNRESTRICTED code pages (mentioned in §7.2) and demonstrates the performance benefits of specialization.

NGINX web server. We used the apachebench, `ab` [1], to send 10,000 requests using 25 concurrent clients to an NGINX web server running a single worker thread.

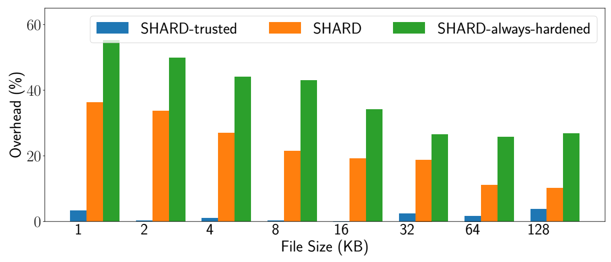


Figure 10: The performance overhead of NGINX across varying requested file sizes.

Figure 10 shows the end-to-end latency increase across different requested file sizes. We observe a higher SHARD overhead for NGINX, 22.21% on average. Unlike Redis, which successively calls the same system call, we observe (Figure 11) a high number of traps which incur EPT switches (i.e., NGINX invokes distinct system calls successively). Furthermore, while the overhead is high (up to 37%) for smaller file sizes, it is amortized over memory and I/O overhead as the file size increases. Note that NGINX showcases the worst-case scenario for SHARD's overhead, i.e., many distinct system calls per-second. In practice, we expect system calls to be small in number (as we show for SPEC below) or to be similar (as Redis). Also, we observe a very low number of hardening instances, showing that in many cases a good representative profiling workload ensures low run-time deviation.

The full-hardening enforcement of NGINX (SHARD-always-hardened) incurs an additional overhead of 8-20% over SHARD. In particular, the average performance overhead, with full-hardening enforcement, becomes 38.17%. Finally, running NGINX as a trusted application (SHARD-trusted) incurs only 1.59% average overhead, similar to Redis.

SPEC CPU 2006. We ran SHARD on the SPEC CPU 2006 integer suite, which includes 12 applications that range from file compression (`bzip2`) to gene sequencing (`hmmer`). All experiments used the reference workloads.

Table 9 shows the overhead caused by SHARD on SPEC applications, including the number of traps. In general, we observe very low overhead (between −0.37 and 2.73%) for these applications. The reason behind this is that while we see many traps at the SHARD monitor, they were dispersed over long-running tests. We expect such patterns to be common in many applications; for such applications SHARD's overhead will likely be very low as well.

10.4 Impact of Profiling Accuracy

This section demonstrates the impact of profiling (in)accuracy on the performance of SHARD. In particular, we illustrate SHARD's performance when profiled with a (a) different application, (b) different application workload, or (c) partial application workload.

Terminology. Related to Figure 12, Figure 13, and Figure 14,

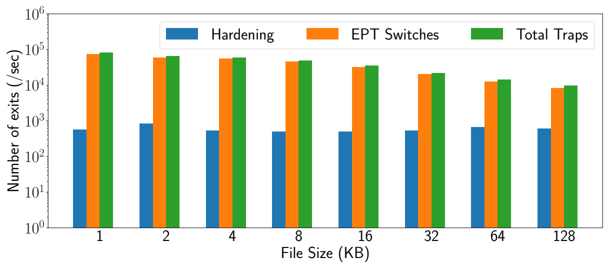


Figure 11: SHARD statistics while running NGINX.

Benchmark	Execution time		Total Traps	EPT Switches	Overhead
	Baseline (s)	SHARD (s)			
400.perl	306	307	195050	75070	0.32%
401.bzip2	436	442	109789	37386	1.38%
403.gcc	270	269	79805	27630	-0.37%
429.mcf	365	375	46804	30845	2.74%
445.gobmk	464	471	125006	79648	1.51%
456.hmmer	356	363	79813	28292	1.97%
458.sjeng	507	518	41770	27955	2.17%
462.libquantum	322	325	34986	25311	0.93%
464.h264ref	669	683	87162	41142	2.09%
471.omnetpp	381	390	46486	31215	2.36%
473.astar	440	442	44225	28441	0.45%
483.xalancbmk	237	240	123595	28655	1.27%

Table 9: SPEC CPU 2006 results. Table only shows numbers while running untrusted applications with SHARD.

SHARD refers to scenarios where profiling was accurate—SHARD was profiled using the same application and workload against which it was evaluated, whereas SHARD-Prof$_{abc}$ refers to scenarios where SHARD was profiled with a different application or workload or partial workload.

Profiling using different application. To evaluate the impact of a different application profile on performance, we generated a SHARD profile using Redis and ran NGINX with the generated profile. We used the `redis-benchmark` for profiling. For evaluation, we used `ab` to send 10,000 requests using 25 concurrent clients to an NGINX server with one worker thread (similar to §10.3).

Figure 12 shows the performance overhead of Redis profile (SHARD-Prof$_{redis}$) compared to accurate profiling (SHARD). As expected, SHARD-Prof$_{redis}$ performs considerably worse. In particular, we noticed a very high number of hardening instances with SHARD-Prof$_{redis}$ because NGINX and Redis profiles are highly-disjoint (as illustrated in §3). For example, retrieving 1KB files, SHARD-Prof$_{redis}$ incurs $\sim 24,000$ hardening instances per-second, compared to $\sim$ 300 hardening instances per-second with SHARD. Consequently, SHARD-Prof$_{redis}$ exhibits a much higher overhead (i.e., upto 89%).

Profiling using different application workload. Next, we evaluated the impact of profiled application workloads on application performance. In particular, we generated a SHARD NGINX profile using `ab`. Afterwards, we evaluated NGINX's performance using `wrk` [19]. During profiling, `ab` generated requests for files between 1 to 128 KB size using 25 concurrent clients. Then, during evaluation, `wrk` requested the same files using the same number (25) of clients.

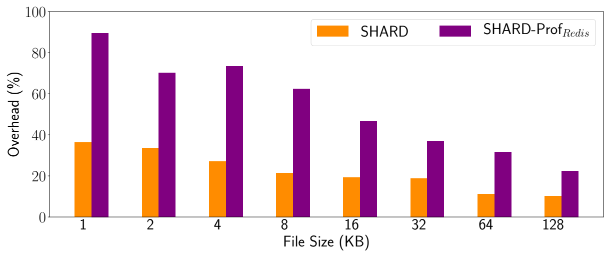


Figure 12: The performance overhead of NGINX when the system is profiled with the same (SHARD) and different (SHARD-Prof$_{redis}$) application.

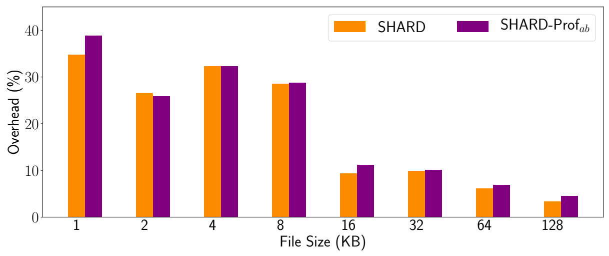


Figure 13: The performance overhead of NGINX when the system is profiled with the same (SHARD) and different application workload (SHARD-Prof$_{ab}$).

Figure 13 shows the performance overhead of the `ab` profile (SHARD-Prof$_{ab}$) compared to an accurate profile using `wrk` (SHARD). We notice that the specific profiled workload, related to an application, has little impact on the application's performance (i.e., less than 2% increase in performance overhead mostly for SHARD-Prof$_{ab}$). Hence, we conjecture that as long as the profiling workload for an application is comprehensive, the exact workload type is less important.

Profiling using partial application workload. Finally, we show the impact on application performance when SHARD is profiled using a partial set of application workloads. In particular, we generated a SHARD profile using half the `redis-benchmarks` and evaluated the performance using the rest. The benchmark applications in the profiling and evaluation sets were randomly chosen. Figure 14 shows the performance with complete (SHARD) and partial (SHARD-Prof$_{part}$) application workload profiles. We notice that SHARD-Prof$_{part}$ increases performance overhead only between $0-3\%$. Hence, our results suggest that a partial profile is also sufficient to offer high performance for an application.

11 Limitations and Discussion

Context-aware control-flow integrity (CFI) creates a narrow window of opportunity for an attacker that full CFI would not. In particular, while the attacker cannot execute an exploit

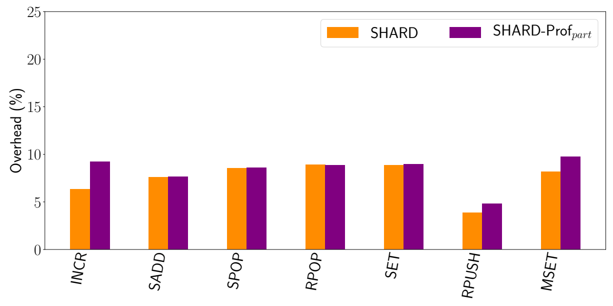


Figure 14: The performance overhead of Redis when the system is profiled with complete (SHARD) and partial set of `redis-benchmark` (SHARD-Prof$_{part}$).

payload directly with context-aware CFI (due to SHARD's hardening and debloating), the attacker can potentially make a malicious update to a function pointer and trick trusted applications (for which the kernel is not hardened or debloated by SHARD) to use the malicious function pointer. Although possible, we expect such attacks to be significantly difficult to perform for several reasons. In particular, the untrusted application is sandboxed (refer to §5); therefore, its interactions with the outside world are rigorously controlled. Furthermore, the attacker must both know the system call semantics of a trusted application and be able to trick the application to use the malicious function pointer in a specific scenario to conduct such attacks. We leave the investigation of these attacks to future work.

Moreover, other techniques can be applied to the kernel with SHARD to provide alternative or complementary context-aware hardening protection. For example, stack exhaustion and stack clearance checks can be applied to prevent attacks through the kernel's stack. These techniques, unlike CFI, are not subject to the limitations of selective hardening [39].

12 Related Work

Dynamic kernel debloating. SHARD is most closely related to previous work in application-driven, run-time kernel (debloating-based) specialization [30, 36, 62]. However, compared to these schemes, SHARD significantly reduces the attack surface by specializing at both the application and system call levels and strictly enforces debloating.

Static kernel debloating. Static, configuration-based specialization [37, 38, 53] is another approach for kernel minimization. Since such techniques statically determine the kernel-view, they provide good performance but with lower security guarantees (e.g., larger attack surface and/or non-strict enforcement).

Application specialization. Specializing of applications has been explored extensively, including for debloating purposes. Trimmer [50] employs static analysis techniques to identify reachable application code with respect to a particular user-provided input. Quach et al [47] statically identify library code needed by an application and use piece-wise compilation and loading to specialize the library-view of the application at run-time. Azad et al [21] and Razor [46], use dynamic profiling to identify and remove the code that is not needed by an application in a particular usage scenario. Finally, CHISEL [31] adopts a delta debugging approach to obtain a minimal program satisfying a set of test cases. Unlike these systems, specializing at the kernel requires addressing additional complexities (e.g., a very large codebase which is hard to accurately analyze statically or dynamically profile) to provide strict enforcement guarantees with low overhead.

Kernel CFI. Control-flow integrity [20] prevents control-flow hijacks by ensuring that control-flow transfers are only to valid targets. Previous work has applied CFI to protect privileged software, including kernels and hypervisors. HyperSafe [56] applies CFI to hypervisors. For CFI enforcement, they introduce a lightweight Restricted Pointer Indexing (RPI) approach. SHARD proposes a modified implementation of RPI which is compatible with function addresses.

KGuard [34] protects the kernel against return-to-user attacks by ensuring indirect control-flow transfers in the kernel cannot target user space addresses. KCoFI [26] uses the secure virtual architecture (SVA) [27] to enforce CFI on the system's kernel. However, their implementation incurs a high overhead, exceeding 100% in some scenarios. Furthermore, Ge et al. [29] apply fine-grained CFI to kernel software by using RPI. The instrumentation causes a high overhead of up to 50%. While SHARD's implementation of CFI is similar, it introduces a modified RPI instrumentation, compatible with function addresses, which allows near-native non-hardened execution. Additionally, SHARD enforces strict debloating, which completely removes many vulnerabilities; thereby, preventing a wide-range of attacks with a low overhead, unlike CFI-only schemes.

Specialized kernel hardening. To the best of our knowledge, the Split-Kernel [39] technique is the only previous effort in specialized kernel hardening. Both Split-Kernel and SHARD implement selective hardening of kernel execution by providing different kernel views to applications based on whether they are trusted or not. However, a major difference is that Split-Kernel fully hardens the kernel view (using coarse-CFI) of untrusted applications, which incurs a high overhead (40% on average on a web server). In contrast, SHARD avoids this overhead by hardening only *potentially reachable* code paths while allowing *reachable* code to execute unrestricted.

13 Conclusion

This paper presents SHARD, a run-time fine-grained kernel specialization system that combines debloating with context-aware hardening to prevent kernel attacks. SHARD achieves an order of magnitude higher attack surface reduction than prior work and implements strict enforcement. Furthermore, SHARD incurs an overhead of only 3-10% on Redis, 10-36% on NGINX, and 0-2.7% on the SPEC CPU benchmarks.

Acknowledgement

We thank our shepherd, Vasileios Kemerlis, and the anonymous reviewers for their valuable comments and suggestions. We also thank Mathias Payer for earlier discussion on software debloating and hardening. This work was supported in part by ONR under Grant N00014-17-1-2513. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the ONR.

References

[1] ab - apache http server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html/.

[2] Amd64 architecture programmer's manual volume 3: General-purpose and system instructions. https://www.amd.com/system/files/TechDocs/24594.pdf.

[3] Cve-2017-10661 detail. https://nvd.nist.gov/vuln/detail/CVE-2017-10661.

[4] Cve-2017-11176 detail. https://nvd.nist.gov/vuln/detail/CVE-2017-11176.

[5] Cve-2017-17052 detail. https://nvd.nist.gov/vuln/detail/CVE-2017-17052.

[6] Cve-2017-17052 detail. https://nvd.nist.gov/vuln/detail/CVE-2018-10880.

[7] Cve-2017-5123. https://security.archlinux.org/CVE-2017-5123.

[8] Cve-2017-7308 detail. https://nvd.nist.gov/vuln/detail/CVE-2017-7308.

[9] Cve-2018-17182 detail. https://nvd.nist.gov/vuln/detail/CVE-2018-17182.

[10] Cve-2018-7480 detail. https://nvd.nist.gov/vuln/detail/CVE-2018-7480.

[11] Cve-2019-20054 detail. https://nvd.nist.gov/vuln/detail/CVE-2019-20054.

[12] Jonathansalwan/ropgadget. https://github.com/JonathanSalwan/ROPgadget.

[13] L1 terminal fault / cve-2018-3615 , cve-2018-3620,cve-2018-3646 / intel-sa-00161. https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault.

[14] The linux kernel enters 2020 at 27.8 million lines in git but with less developers for 2019. https://www.phoronix.com/scan.php?page=news_item&px=Linux-Git-Stats-EOY2019#:~:text=The%20Linux%20Kernel%20Enters%202020,Less%20Developers%20For%202019%20%2D%20Phoronix&text=As%20of%20this%20morning%20in,in%20at%2027.8%20million%20lines!

[15] Linux kernel grows past 15 million lines of code. https://www.tomshardware.com/news/Linux-Linus-Torvalds-kernel-too-complex-code,14495.html.

[16] Nginx | high performance load balancer, web server, amp; reverse proxy. view-source:https://www.nginx.com/.

[17] Redis. redis.io.

[18] Vulnerability details : Cve-2016-0728. https://www.cvedetails.com/cve/CVE-2016-0728/.

[19] wrk - a http benchmarking tool. https://github.com/wg/wrk.

[20] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 2009.

[21] B. A. Azad, P. Laperdrix, and N. Nikiforakis. Less is more: Quantifying the security benefits of debloating web applications. In *Proceedings of the 28th USENIX Security Symposium (Security)*, 2019.

[22] J.-J. Bai, J. Lawall, Q.-L. Chen, and S.-M. Hu. Effective static analysis of concurrency use-after-free bugs in linux device drivers. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2019.

[23] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40, 2011.

[24] N. Burow, X. Zhang, and M. Payer. Shining light on shadow stacks. *arXiv preprint arXiv:1811.03165*, 2018.

[25] N. Burow, X. Zhang, and M. Payer. Sok: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 985–999, 2019.

[26] J. Criswell, N. Dautenhahn, and V. Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2014.

[27] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[28] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 555–566, 2015.

[29] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-grained control-flow integrity for kernel software. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016.

[30] Z. Gu, B. Saltaformaggio, X. Zhang, and D. Xu. FACE-CHANGE: application-driven dynamic kernel view switching in a virtual machine. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, (DSN)*, 2014.

[31] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik. Effective program debloating via reinforcement learning. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[32] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. December 2016.

[33] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM Conference on Computer and Communications Security (CCS)*, 2018.

[34] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis. kguard: lightweight kernel protection against return-to-user attacks. In *Proceedings of the 21st USENIX Security Symposium (Security)*, 2012.

[35] A. Konovalov. Blogger. https://googleprojectzero.blogspot.com/.

[36] H. Kuo, A. Gunasekaran, Y. Jang, S. Mohan, R. B. Bobba, D. Lie, and J. Walker. Multik: A framework for orchestrating multiple specialized kernels. *CoRR*, abs/1903.06889, 2019.

[37] H.-C. Kuo, J. Chen, S. Mohan, and T. Xu. Set the configuration for the heart of the os: On the practicality of operating system kernel debloating. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2020.

[38] A. Kurmus, R. Tartler, D. Dorneanu, B. Heinloth, V. Rothberg, A. Ruprecht, W. Schröder-Preikschat, D. Lohmann, and R. Kapitza. Attack surface metrics and automated compile-time OS kernel tailoring. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*, 2013.

[39] A. Kurmus and R. Zippel. A tale of two kernels: Towards ending kernel hardening wars with split kernel. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2014.

[40] P. Larson. Testing linux® with the linux test project. In *Ottawa Linux Symposium*, page 265, 2002.

[41] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[42] K. Lu, A. Pakki, and Q. Wu. Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences. In *Proceedings of the 28th USENIX Security Symposium (Security)*, 2019.

[43] A. Lyashko. Hijack linux system calls: Part iii. system call table. Blog] System Programming, Available at:.

[44] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna. DRCHECKER: A soundy analysis for linux kernel drivers. In *Proceedings of the 26th USENIX Security Symposium (Security)*, 2017.

[45] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014.

[46] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee. RAZOR: A framework for post-deployment software debloating. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1733–1750, Santa Clara, CA, Aug. 2019. USENIX Association.

[47] A. Quach, A. Prakash, and L. Yan. Debloating software through piece-wise compilation and loading. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 869–886, 2018.

[48] D. Rosenberg. Anatomy of a remote kernel exploit, 2011.

[49] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, 2007.

[50] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar. Trimmer: application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 329–339, 2018.

[51] L. Szekeres, M. Payer, L. T. Wei, and R. Sekar. Eternal war in memory. In *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*, 2014.

[52] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[53] R. Tartler, A. Kurmus, B. Heinloth, V. Rothberg, A. Ruprecht, D. Dorneanu, R. Kapitza, W. Schröder-Preikschat, and D. Lohmann. Automatic OS kernel TCB reduction by leveraging compile-time configurability. In *Proceedings of the 8th Workshop on Hot Topics in System Dependability, (HotDep)*, 2012.

[54] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, 2014.

[55] W. Wang, K. Lu, and P.-C. Yew. Check it again: Detecting lacking-recheck bugs in os kernels. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[56] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2010.

[57] Z. Wang, C. Wu, M. Xie, Y. Zhang, K. Lu, X. Zhang, Y. Lai, Y. Kang, and M. Yang. Seimi: Efficient and secure smap-enabled intra-process memory isolation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 592–607, 2020.

[58] W. Wu, Y. Chen, X. Xing, and W. Zou. KEPLER: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities. In *Proceedings of the 28th USENIX Security Symposium (Security)*, 2019.

[59] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 781–797, 2018.

[60] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P)*, 2009.

[61] T. Zhang, W. Shen, D. Lee, C. Jung, A. M. Azab, and R. Wang. Pex: A permission check analysis framework for linux kernel. In *Proceedings of the 28th USENIX Security Symposium (Security)*, 2019.

[62] Z. Zhang, Y. Cheng, S. Nepal, D. Liu, Q. Shen, and F. A. Rabhi. KASR: A reliable and practical approach to attack surface reduction of commodity OS kernels. In *Proceedings of the 21st International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2018.

[63] Z. Zhou, M. K. Reiter, and Y. Zhang. A software approach to defeating side channels in last-level caches. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS), 2016.*

[64] P. Zieris and J. Horsch. A leak-resilient dual stack scheme for backward-edge control-flow integrity. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 369–380, 2018.