SysXCHG: Refining Privilege with Adaptive System Call Filters

Alexander J. Gaidis
Brown University
Providence, RI, USA
agaidis@cs.brown.edu

Vaggelis Atlidakis
Brown University
Providence, RI, USA
eatlidak@cs.brown.edu

Vasileios P. Kemerlis
Brown University
Providence, RI, USA
vpk@cs.brown.edu

ABSTRACT

We present the design, implementation, and evaluation of SysXCHG: a system call (syscall) filtering enforcement mechanism that enables programs to run in accordance with the principle of least privilege. In contrast to the current, hierarchical design of seccomp-BPF, which does not allow a program to run with a different set of allowed syscalls than its descendants, SysXCHG enables applications to run with "tight" syscall filters, uninfluenced by any future-executed (sub-)programs, by allowing filters to be dynamically exchanged at runtime during execve[at]. As a part of SysXCHG, we also present xfilter: a mechanism for fast filtering using a process-specific view of the kernel's syscall table where filtering is performed. In our evaluation of SysXCHG, we found that our filter exchanging design is performant, incurring ≤ 1.71% slowdown on real-world programs in the PaSH benchmark suite, as well as effective, blocking vast amounts of extraneous functionality, including security-critical syscalls, which the current design of seccomp-BPF is unable to.

CCS CONCEPTS

• **Security and privacy → Systems security**; **Operating systems security**; **Software security engineering**.

KEYWORDS

Attack surface reduction, system call filtering, adaptive filtering

ACM Reference Format:
Alexander J. Gaidis, Vaggelis Atlidakis, and Vasileios P. Kemerlis. 2023. SysXCHG: Refining Privilege with Adaptive System Call Filters. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3623137

1 INTRODUCTION

Software has been progressively increasing in size and complexity. This trend can be attributed to several factors, including maintaining backwards compatibility with legacy code bases [82] and supporting a diverse set of features for growing, heterogeneous user-bases [39, 77, 81, 95]. The ramifications of this *software bloat* are twofold: (1) applications require an abundant set of OS services, delivered via the system call (syscall) API, to allocate memory, perform inter-process communication, make network connections, interact with the file system, *etc.*, and (2) attackers are provided with additional avenues for abusing applications [2, 80].

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0050-7/23/11.
https://doi.org/10.1145/3576915.3623137

Due to (1) and (2) above, the attack surface of an application (and the kernel) is enlarged [2], and hence an adversary has greater potential to exploit applications [18], after which they can also request a variety of kernel services to further elevate their privileges [61].

Moreover, userland programs have access to the entire syscall API by default, despite only requiring a fraction—across ≈30k binaries in Debian, the median syscall count per binary is 90 (≈26% of the total syscall API) and the 90th percentile is 145 (≈42%) [18]—, resulting in *over-privilege*, or the expansion of a program's functionality or abilities beyond what it requires for benign (i.e., non-adversarial) execution. This over-privilege exacerbates point (2) above, and it also allows an attacker to take full advantage of (1) by providing a rich post-exploitation arsenal, including privilege-escalation, by means of exploiting vulnerabilities in less-stressed syscalls [14, 15].

To limit the over-privilege of programs with respect to the syscall API, Linux introduced Seccomp BPF (seccomp-BPF) [99] to mediate access to syscalls at runtime. seccomp-BPF enables users to push BPF [69] programs into the kernel, which specify an action to take (e.g., "allow", "block", "log") for a given syscall based on its number and arguments. The majority of works regarding syscall filtering [8, 10, 18, 28, 29, 31, 51, 59, 76, 103, 104, 108] employ seccomp-BPF.

Problem #1: seccomp-BPF's hierarchical design. seccomp-BPF suffers from a hierarchical design inherently over-privileging processes by allowing access to syscalls not required for their execution. Specifically, seccomp-BPF filters cannot be removed once installed, and they are inherited across process creation and program execution. As a result, the set of syscalls allowed by a process must include those required for the current program's execution in addition to those required by *descendant* programs executed in the future via execve[at]. In general, this leads to over-privilege.

Solution #1: SysXCHG. To improve upon the aforementioned limitation(s) of seccomp-BPF, we present SysXCHG: a policy enforcement mechanism that allows *dynamic switching* of syscall filters at runtime in accordance with the principle of least privilege (PoLP) [88]. SysXCHG centers around a new concept called exec filters that are syscall filters, including, but not limited to, seccomp-BPF filters, which are *embedded* in binaries and installed automatically when the OS loads/executes the corresponding program. Using this primitive, SysXCHG introduces a new *exchange model* that differs from seccomp-BPF's default, over-privileged *inheritance model*, by exchanging any previously installed exec filters with the exec filters embedded in the soon-to-be-executed program. SysXCHG's exchange model allows a program to run with a syscall set specific to its own functionality, uninfluenced by any programs executed in the future. SysXCHG also accounts for developer-intended semantics and compatibility with legacy code bases by preventing any manual filters—i.e., those installed by a developer—from being removed, maintaining compatibility with the default inheritance model.

This allows for refinement of an application's syscall set beyond what is imposed by any `exec` filters. Additionally, to ensure that an attacker cannot tamper with `exec` filters, potentially increasing their privilege, `SysXCHG` binds `exec` filters to binaries with cryptographic signatures, guaranteeing immutability.

Problem #2: `seccomp-BPF`'s filter-install time. Recent work in syscall filtering [8, 10, 18, 28, 29, 31, 51, 59, 103, 104] primarily uses syscall numbers while filtering, which largely eliminates the need for most of `seccomp-BPF`'s argument-filtering infrastructure. To address this problem, kernel developers introduced a bitmap cache positioned at the entry-point of `seccomp-BPF`, which, in the best case, can quickly allow a syscall, preventing all installed filter programs from running in order to reach a decision. Alas, to update the bitmap cache when a filter is installed, the execution of the filter is *emulated* for each necessary syscall, resulting in a large increase to cold-start (i.e., no previous filters are installed) install times: ≈42% to ≈363% (§6.1.3).

Solution #2: `xfilter`. `SysXCHG` introduces an optimized installation and filtering mechanism called *express filter*, or `xfilter` for short, which addresses the weaknesses in `seccomp-BPF`'s filter installation procedure. Rather than relying on emulation, `xfilter` determines the "allow/deny" information statically and encodes it in a new type of filter that, when installed, removes the need for `seccomp-BPF` entirely when conducting syscall-number-based filtering. At a high-level, this new enforcement mechanism works by creating separate, process-specific views of the global syscall table, where filtering is performed by replacing syscall handlers with error stubs. Thus, in addition to *reducing* the filter installation time, when compared with `seccomp-BPF`, it also offers a *filtering shortcut*, removing the need for additional function calls in the hot-path of syscall entry that are required for filtering with `seccomp-BPF`.

Performance. We conducted a performance evaluation of `SysXCHG` to measure the overhead associated with: (1) using `exec` filters versus manual filters, (2) installing `seccomp-BPF` filters versus `xfilter` filters (`xfilters`), (3) using `seccomp-BPF` versus `xfilter` to filter, and finally, (4) exchanging both `seccomp-BPF` filters and `xfilters` at runtime. Regarding (1), the difference between the two filter types was negligible across SPEC CPU 2017 being ≤ 0.44% for `exec` and manual `seccomp-BPF` filters, and ≤ 0.36% for `exec` and manual `xfilters`. We evaluated (2) with a small microbenchmark that measured the time it takes for a filter-installing syscall to complete. This revealed that `xfilter` can reduce filter installation time, when compared with `seccomp-BPF`, between ≈76% and ≈97% over SPEC CPU 2017 and a set of real-world applications. We also used these benchmark programs to evaluate (3), where we found the overhead associated with `xfilter` (≈0% to 1.11%) was on par with, or better than, the overhead of `seccomp-BPF` (≈0% to 1.08%). Finally, to evaluate (4), we used the PaSH [46] benchmark suite, where we found that filter exchanging is performant regardless of whether `seccomp-BPF` or `xfilter` is employed. PaSH macrobenchmarks showed `seccomp-BPF` overheads ranging from ≈0% to 2.74%, and `xfilter` overheads ranging from ≈0% to 1.71%. Notably, `xfilter` outperformed `seccomp-BPF` in nearly all benchmarks.

Effectiveness. Finally, we examined the effectiveness of filter exchanging from a security standpoint, in both a quantitative and qualitative way, using the PaSH benchmark suite.

Our quantitative approach calculated the magnitude of additional syscalls a program must run with to support its execution, as well as any future executed programs, under the inheritance model. These additional syscalls we deem over-privilege, which `SysXCHG`'s exchange model completely removes from a given program's allowed syscall set. Notably, we found that the common, real-world programs found in the PaSH benchmarking suite can be grossly over-privileged under the currently employed inheritance model, in some cases allowing up to 113.73% more syscalls than what is needed. Our qualitative approach examines the functionality obtained by an attacker that has access to this additional set of syscalls, as well as whether any syscalls in the set are security-critical. We found that the majority of programs that execute others contain security-critical syscalls and a large body of functionality—both of which are the sole result of the inheritance model. Switching to `SysXCHG`'s exchange model remedies this problem, and importantly, it does so efficiently, with negligible performance overhead.

2 BACKGROUND AND RELATED WORK

2.1 Syscalls and Syscall Filtering

In contemporary OSes, when an application requires a *service* from the underlying kernel—such as spawning additional processes, allocating memory, accessing the filesystem, or making network connections—it makes a request through the *syscall API*. On the x86-64 architecture, syscalls are *software interrupts* that trap to kernel mode to perform some *privileged* operation before returning the result to userland. Importantly, Linux kernel v6.0.8 provides 361 syscalls—396 if legacy x86-32 sycalls are included—without any default restrictions on what syscalls a process can make.

Syscall interposition takes advantage of the well-defined interface boundary between the OS kernel and userland to attach additional functionality to the entry or exit path of syscalls. As meaningful changes to the system are performed via syscalls, interposing on this boundary provides visibility into how a process interacts with the system, which can inform modifications of the requested services. For example, syscalls can be intercepted and extended to provide new filesystem facilities [3], portable execution environments [37, 50], record-replay debuggers [87], multi-variant execution systems [53], intrusion detection systems [38, 60], and more [4, 30, 45, 52, 56, 91]. Importantly, syscall interposition can be used to *filter* syscalls according to a given policy, reducing a process' privilege by confining its set of allowed functionality (with regards to the syscall API) to what is necessary for its operation. Syscall filtering designs can be classified according to *where*/*how* policy decisions are made and enforced, and fall into three main categories. Namely, decision and enforcement either: (1) both occur in *user mode*; (2) both occur in *kernel mode*; or (3) are split between the domains in (1) and (2).

User Mode. A standard way both policy decisions and enforcement are carried-out in userland is with process tracing utilities provided by the kernel: namely, the `/proc` pseudo file system [64] or the `ptrace` syscall. These allow one user process (the *tracer*) to control the execution of one or more other processes (the *tracees*), providing a mechanism to filter syscalls. Prior work [34, 42] used these utilities to hook syscall entry and exit, examine syscall numbers and arguments, and modify/block syscalls.

This approach: (1) provides a solution that is easy to test and deploy (i.e., no superuser privilege or kernel recompilation is required); and (2) incurs no additional code (that is executed) in kernel mode, where vulnerabilities can have greater repercussions [49, 61]. Nonetheless, this approach has not achieved widespread adoption due to poor performance and security: syscall entry and exit hooks in the kernel trampoline execution to the tracer process requiring four additional context switches (two for both the entry and exit hooks); while both the tracer and the tracees execute at the same privilege level, thus making bypassing policy enforcement easier.

Dune [5] provides a unique approach to filtering syscalls in userland. By exposing privileged hardware features to user processes, Dune is able to divide a program's address space into two halves operating at different privilege levels. When the main program running in the lesser privileged half invokes a syscall, control flow is trampolined to a syscall handler in the more privileged half (by manipulating page tables to map the custom syscall handler on demand) where an appropriate action can be taken. The main disadvantage of Dune is its reliance on hardware-assisted virtualization.

Kernel Mode. Moving policy decision-making and enforcement into the kernel allows for optimal performance as well as greater visibility and control over the system. Several implementations [11, 23, 48, 70, 104] use kernel modules to extend the kernel's syscall handling functionality, while others [6, 31, 59, 62, 76, 83, 93] modify the kernel's source code directly. Most kernel-based syscall filtering approaches [8, 10, 18, 28, 29, 31, 51, 59, 76, 103, 104, 108] rely on Linux's Seccomp BPF infrastructure [99].

Hybrid. The final category splits policy decision-making and enforcement across different domains. Systrace [79] and Janus [25] are two influential hybrid schemes that extend the kernel to query a user-mode policy daemon to inform how the kernel should handle a given syscall. A related design, Ostia [26], uses a kernel module to prevent a given application from performing syscalls that access sensitive resources, instead trampolining control flow back to userland where a daemon process safely performs the syscall on the application's behalf. This "delegating" hybrid architecture has also been used to perform syscall filtering in PKU-based memory isolation systems, such as Jenny [89]. Finally, Onoue et al. [74] proposed a hybrid scheme that conducts policy enforcement in a hypervisor and policy deliberation in an isolated, purpose-specific virtual machine. Overall, while hybrid designs can minimize the amount of code added to sensitive areas, their designs often require multiple context switches, resulting in suboptimal performance [25, 26, 79].

2.2 Seccomp BPF

Seccomp (SECure COMPuting) [44] was added to Linux in v2.6.12 to restrict the syscalls available to a process to only `read`, `write`, `exit`, and `sigreturn`; thus confining a process to pure computation and basic I/O (with pre-existing file descriptors). Subsequently, Linux v3.5 introduced `seccomp-BPF` [99], substantially broadening seccomp's abilities by allowing BPF programs [69] to define *arbitrary* syscall filtering policies. After a process installs a `seccomp-BPF` filter, with the `seccomp` or `prctl` syscalls, a hook-point in the kernel's syscall entry path subjects all further syscalls made by the process, *and its children*, to the policy set by the filter.

For each syscall made, the kernel passes the syscall's context (i.e., syscall number, architecture, instruction pointer at the time of the syscall, and register arguments) to the BPF program, which determines the appropriate action (e.g., kill the process making the syscall, return an error value, log the syscall, or allow the syscall).

Multiple BPF programs can be installed simultaneously, forming a *stack*, where the most recently-installed program is executed first. To determine the action for a given syscall when multiple filter programs are installed, the syscall is filtered through every program and the *least permissive* action is used. Since filter programs are typically large sequences of conditionals—`seccomp-BPF` uses cBPF (classic BPF), which only allows forward-directed branches [43]—, executing every BPF program for every syscall can result in sub-par performance [93]. However, programs can be optimized by employing a *skip list* [18], and recently, Linux v5.11 added a *bitmap cache* to `seccomp-BPF`'s architecture, where syscalls that are allowed based solely on (syscall) number are added to a bitmap, which is queried to determine if the BPF program(s) need(s) to run [106]. Recent work [8, 10, 18, 28, 29, 31, 51, 59, 103, 104] generates filters mostly using syscall numbers, greatly benefiting from the bitmap cache.

Unfortunately, `seccomp-BPF` *suffers from over-privilege* due to its *hierarchical* and *append-only* design: a process inherits filters across `[v]fork` and `clone`, and retains filters when the process' image is replaced with another via `execve[at]`. More specifically, a field in the Linux kernel's `struct task_struct` maintains a set of `seccomp-BPF` programs, which remains unchanged when a new program is executed or is copied to a new `struct task_struct` when a new process is forked. Given this *inheritance model*, additional filters can only be installed to *further constrain* the allowed syscall set; thus, to prevent breaking an application, processes must initially allow all of the syscalls required throughout their (and their children's) lifetime. This set may contain syscalls the current program does not require, but are included for a succeeding program, resulting in over-privilege of the current program [18, §3.2].

2.3 Syscall Filtering Policies

execve Semantics. The `execve` and `execveat` syscalls are used to execute a program, replacing the current, calling program with a new one (in addition to initializing new stack, heap, and data segments). Importantly, `execve[at]` does not create a new process—the job of `clone` and `[v]fork`—, rather, it only changes the process' image, retaining the "shell" of the process, which includes, in part, its `PID`, open file descriptors, and any installed `seccomp-BPF` filters (§2.2). In terms of syscall filtering, `execve[at]` poses a design challenge as policies of the current and new programs may differ with respect to the functionality they are blocking. Some syscall filtering schemes, such as `seccomp-BPF` [99], only allow for *privilege reduction*, creating a situation where a program's installed policy must be a superset of all subsequently-executed programs—which can later refine the policy—, potentially resulting in over-privilege of any program that calls `execve[at]`. Other schemes [11, 42, 74, 79] allow a new policy to replace the current policy when a new program is executed, possibly expanding the privilege of the new program. Nevertheless, these designs do not verify the integrity of installed policies, allowing an adversary to tamper with policies and maliciously give a program additional privileges.

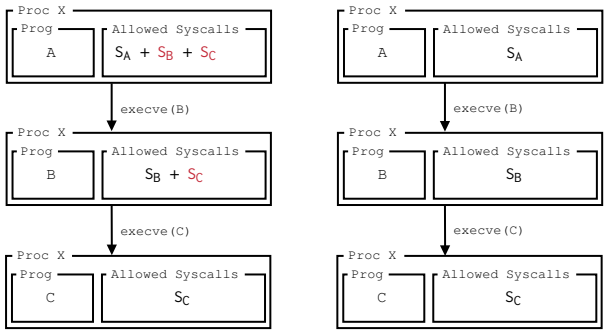
Policy Storage. *Storage* and *integrity* of syscall policies are fundamental security considerations for the enforcement mechanisms discussed in Section 2.1. In general, policies are stored in one of three ways: (1) as separate files, or modules, from the binary they enforce [6, 11, 18, 23, 25, 26, 34, 42, 59, 70, 74, 79], (2) encoded in the binary as an implicit policy [48, 83], or (3) attached to the binary [10, 62], creating a policy-bearing executable in the spirit of DuVarney et al. [20]. While there is more book-keeping involved in schemes that use separate policy files, neither approach is inherently more secure—both suffer from the inability to detect *policy tampering*. Currently, there has been no scheme that incorporates integrity checking of policies before they are applied at runtime.

2.4 Integrity Measurement Architecture

The Linux Integrity Measurement Architecture (IMA) was first added to the kernel as a part of the integrity subsystem, in v2.6.30, to detect file corruption [47]. The initial components, largely based on the work of Sailer et al. [86], allow a remote host to *verify* file integrity using a hardware (e.g., TPM-based) root of trust. Linux v3.3 added the extended verification module (EVM) to IMA, which detects modifications to a file's *metadata* (e.g., inode number; file owner, group, and mode; and LSM-related extended attributes), offering enhanced protection. EVM computes an HMAC, or signed hash, over a given file's metadata and stores it in the `security.evm` extended attribute. `security.evm` can then later be used to verify the integrity of the file's extended attributes. IMA was again extended in Linux v3.7 with an enforcement mechanism known as *appraisal* that provides integrity checks for (local) file systems. Offline, file hashes are signed and stored in a `security.ima` extended attribute of the corresponding file. After rebooting the system into appraisal's enforcement mode, opening a file that matches the installed IMA policy will be subject to verification of the digital signature in `security.ima`. Attempting to open a file that was unintentionally or maliciously altered after *measurement* (i.e., hashing and signing) will result in a failed integrity check and access denial.

3 THREAT MODEL

Adversarial Capabilities. We assume an adversary targeting userland applications (primarily, but not exclusively, written in C/C++ and/or ASM) that possess the ability to issue system calls. The attacker is able to exercise vulnerabilities in an application's main executable or dependent, shared libraries, modules, add-ons, *etc.*, to consequently exploit the victim program and make *arbitrary* syscalls. Importantly, we do not constrain (1) the *types of vulnerabilities* (ab)used by the attacker—e.g., spatial/temporal memory errors [71, 72, 98]; logic errors, such as missing authorization/-authentication checks [16, 94]; or discarded returned (error) values [35, 105]—, nor (2) the applied *exploitation technique*—e.g., code injection, code reuse, {data, block}-oriented programming, data-only attacks [12, 19, 33, 40, 41, 90, 92, 107]. Ultimately, we assume an attacker that can achieve arbitrary code execution and/or issue arbitrary syscalls with the intent to increase their privilege [61] or request services from the OS [18]. More formally, an attacker is able to repeatedly (if required) invoke any syscall, with arbitrary arguments of their choosing, at arbitrary times. Overall, the adversarial capabilities we consider are on par with the state-of-the-{art, practice} regarding syscall filtering [9, 13, 18, 27–29, 59, 75].



(a) Inheritance Model **(b) Exchange Model**

Figure 1: Syscall filtering models (inheritance vs. exchange). S_P represents the syscall set required for a program P's benign execution. Over privilege is highlighted in red.

Hardening Assumptions. We require a Linux kernel with support for `seccomp-BPF` [99] and a digital signature mechanism to sign binaries, such as IMA [47], and we assume the two can not be disabled. Further, we assume that target applications contain benign code. Standard userland hardening schemes—e.g., non-executable memory [65], stack/heap canaries [17], ASLR [22], code diversification [54, 58, 101], CFI [1, 24, 100, 109], CPI [57], and protection against data-only attacks [78]—are orthogonal to our scheme: we neither preclude nor depend on them. Finally, we deem side-channel [36] and microarchitectural attacks [9, 21] out of scope. Given the above, an attacker is still able to take full control of a userland process to execute arbitrary syscalls in an attempt to either: (1) elevate their privileges by finding and exercising vulnerabilities in syscalls [61], or (2) maliciously request unintended services from the OS under the guise of a benign process [18] (e.g., use the `setxattr` syscall to set a file's extended attributes).

4 DESIGN

The goal of `SysXCHG` is to reduce a program's allowed set of syscalls to those required for operation, in accordance with the *PoLP* [88]. The effect of this is twofold: (1) it reduces the potential of an attacker escalating their privileges by exploiting vulnerabilities in less-stressed syscalls [61], and (2) it reduces the functionality available to an attacker after they have taken control of a process [18]. Modern approaches to filtering syscalls [8, 10, 18, 28, 29, 31, 51, 59, 76, 103, 104, 108] typically rely on `seccomp-BPF`, despite its inherent hierarchical design, leading to *over-privilege* (§2.2). The over-privilege of this inheritance model is exemplified in Figure 1a: program `A` must allow all syscalls in its syscall set, S_A, as well as those for programs `B` and `C` which are executed subsequently. This results in `A` allowing S_A, S_B, and S_C, despite only needing S_A for its own execution—a gross violation of PoLP. When program `B` is executed, it can refine the set of allowed syscalls, getting rid of S_A, but it must keep S_C, resulting in another violation.

In this example, only C runs with the least amount of syscalls required for benign execution. Notably, we assume that $S_A \not\supseteq \{S_B, S_C\}$ and $S_B \not\supseteq S_C$; i.e., a program's syscall set is not a superset of its descendants'. We found this to predominantly be the case in practice (§6.2.1). Hence, the scenario presented in Figure 1a holds true regardless of the exact means used for extracting/specifying S_A, S_B, and S_C; that is, irrespective of how "strict" or "loose" these sets are (due to the specifics of the underlying techniques used for extracting/specifying them), seccomp-BPF always results in over-privilege when execve is involved [18].

Our design of SysXCHG offers an alternative to the strict inheritance model of seccomp-BPF: the *exchange model*. Under the exchange model, programs are associated with one or more exec *filters*, which are installed during execve[at], replacing any exec filters the old program (i.e., the one that invoked execve[at]) previously installed, as shown in Figure 1b. To maintain developer-intended semantics, any installed *manual filters*—i.e., those a developer installed with seccomp or prctl—adhere to the inheritance model and are never exchanged throughout the lifetime of the process. This allows for the exec filters to reduce the set of allowed syscalls down to a safe/better approximation for each individual program, while the manual filters define filtering rules that last the lifetime of the process, potentially spanning the execution of multiple programs. Therefore, with SysXCHG, a program that requires fewer syscalls than its predecessors does not need to allow a union of its syscall set and the syscall sets of all programs it executes in the future, as would be the case with seccomp-BPF's inheritance model. Finally, to prevent an attacker from tampering with exec filters, we embed them in their respective (ELF) binaries and sign/appraise these binaries using the Linux IMA subsystem (§2.4).

4.1 Exec Filters

SysXCHG defines two *types* of filters: (1) manual filters which are intentionally installed by a developer with prctl or seccomp, and (2) exec filters which are installed automatically when a program is executed with execve[at]. We associate a given exec filter with a corresponding binary by embedding the filter in the latter. Later, when the binary is executed, the kernel extracts the exec filter and applies it, enabling enforcement. The exec filter is central to the design of SysXCHG as it enables its filter exchanging paradigm (§4.2). The remainder of this subsection details the design of exec filters according to the standard inheritance model of seccomp-BPF. Subsequently, in Section 4.2, we elaborate on the usefulness of exec filters and how they empower filter exchanging.

Extraction and Embedding. In an initial, offline phase of SysXCHG shown in steps ① and ② of Figure 2, a syscall policy for a given ELF binary is extracted, formatted into a seccomp-BPF program, and embedded in the binary. Since our design focuses on syscall filtering enforcement, it is *agnostic* to the technique(s) used for extracting a program's syscall set (step ①). Although, preferably, the method of choice should derive *complete* syscall sets, with few false positives (i.e., syscalls the program does not use) to prevent program breakage and reduce over-privilege. The policy that results from syscall extraction is then fed to a program that crafts a corresponding seccomp-BPF program and embeds it in the target binary (step ②). Specifically, the length and contents of the BPF

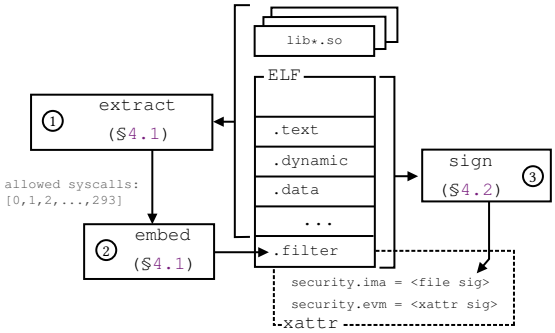


Figure 2: Offline components of SysXCHG's design: syscall filter extraction and embedding, as well as binary signing.

program (i.e., struct sock_fprog) are stored in a new ELF section, dubbed .filter, which is marked read-only and non-allocatable. In the case that multiple policies should be applied to a binary, either: (1) the policies can be merged into a single seccomp-BPF program, or (2) multiple seccomp-BPF programs (and their lengths) can be generated and stored in the .filter section consecutively. Additionally, we do not place any requirements on the type (filtering based on syscall number, arguments, *etc.*) of seccomp-BPF programs stored in the .filter section and used as exec filters.

Ideally, all of a system's binaries would carry exec filters that automatically constrain their privilege when executed. Consequently, our embedding design is largely motivated by compatibility with commercial off-the-shelf (COTS) binaries: we use binary (ELF) rewriting [102] to add the .filter section, avoiding any requirement on source code or toolchain modifications, and we do not require the enforced binary to load or run any additional code to install filters and attain protection, as is the case with most modern enforcement schemes [10, 18]. Our design can be employed by vendors distributing software or end users, with the latter protecting individual binaries or integrating our design into package managers to protect binaries as a step in their install process [54].

Inheritance Model Enforcement. Once filter extraction and embedding are complete, the system can enter an online phase where enforcement occurs. During this phase, a SysXCHG-aware kernel searches all executed ELF binaries for a .filter section while parsing the program and setting up its address space. If the section is found, its contents are extracted and installed normally using the standard seccomp-BPF install pathway. Namely, when the inheritance model is used, there is no difference in the installation procedure between exec and manual filters: they both act to only further restrict the set of allowed syscalls. Additionally, syscalls that are allowed based solely on number (and architecture) are added to a bitmap cache (§2.2) and filter programs are just-in-time (JIT) compiled (if specified by the kernel's configuration) [18].

4.2 Filter Exchanging

The main component of SysXCHG's design is the ability to *dynamically* switch syscall filters, at runtime, in a secure manner. In contrast to the inheritance model (§2.2, §4.1), this exchange model authorizes a process' allowed syscall set to grow or shrink according to the requirements of the currently executing program.

At a high-level, the filter exchanging design handles `exec` and manual filters differently, using `exec` filters to specify the maximum privilege of a program and manual filters to further refine the privileges based on developer intention (if needed). Notably, the design decisions that enable filter exchanging mainly apply to the kernel (with the exception of binary signing); the description of `exec` extraction and embedding in Section 4.1 remains unchanged. Given that, the remainder of this subsection details filter exchanging enforcement before moving on to discuss the security ramifications of allowing a process to potentially (but safely) increase its privileges.

Exchange Model Enforcement. Unlike the inheritance model, where both `exec` and manual filters are considered equal and are installed to the same filter list in the kernel's `seccomp-BPF` infrastructure, the exchange model *segregates* the two filter types, maintaining two distinct filter lists. The list of manual filters is inherited by child processes, preserved across `execve[at]`, and append-only. In contrast, the list of `exec` filters is inherited by child processes, but it is reset (i.e., cleared) and repopulated on every invocation of `execve[at]`. Thus, manual filters can be thought of as working on a *process-level* granularity, while `exec` filters work on a *program-level* granularity. Preserving `seccomp-BPF`'s current inheritance model with manual filters is important as it allows for compatibility with legacy software and the intentions of developers.

When the system is in "steady state," after syscall filter extraction and embedding, a given binary can either carry an `exec` filter (in a `.filter` section) or not. We consider the latter case equivalent to the binary carrying an `exec` filter that allows all syscalls—i.e., no filtering should be performed. When a program is executed, the kernel first extracts any `exec` filters embedded in the binary. Next, the kernel removes all `exec` filters the previous program installed from the process' `exec` filter list, and installs the new `exec` filters. In the case of a binary carrying no `.filter`, the previous `exec` filters are removed and none are installed in their place.

After installation, when a syscall is made, it is first filtered through the bitmap cache of the manual filters, followed by the bitmap cache of the `exec` filters, exiting early if both of the bitmap caches allowed the syscall. If the filtering result is still undecided, all of the manual filters are executed followed by all of the `exec` filter programs, and the most restrictive action is taken. Despite being maintained and administered separately, the execution of the two filter types is largely transparent to userland, appearing as a single list of filters, with one caveat: `seccomp-BPF` executes multiple filters in the reverse order that they are installed, so the presence of `exec` filters may alter this ordering.

In summary, the exchange model improves upon the inheritance model by giving each program only the syscalls they need to run correctly without consideration of process and program hierarchies. Ideally, a program's `exec` filters would define a set of syscalls required for (benign) execution. Additional filtering, via manual filters, can further constrain the allowed set of syscalls; e.g., using configuration files to guide additional syscall set refinement [29].

Security Considerations. Exchanging filters when new programs are executed could allow an adversary to *increase* their privileges (i.e., the syscalls they can invoke). Namely, an adversary could do one of two things after exploiting the current program (assuming no manual filters are installed): (1) create a new, arbitrary program with no `exec` filter and execute it, or (2) modify or strip an existing binary's embedded `exec` filter before executing it. Both cases result in an attacker-controlled process with an attacker-controlled syscall set, potentially allowing full access to the syscall API. `SysXCHG` addresses this privilege escalation by cryptographically binding policies to binaries with digital signatures, as shown in step ③ of Figure 2. We designed our system to be compatible with IMA/EVM, as this is a particularly convenient means for robust signature checking and readily available in the Linux kernel. However, our design is not restricted to only IMA/EVM.; `SysXCHG` is agnostic to the underlying method used for binary signing and verification.

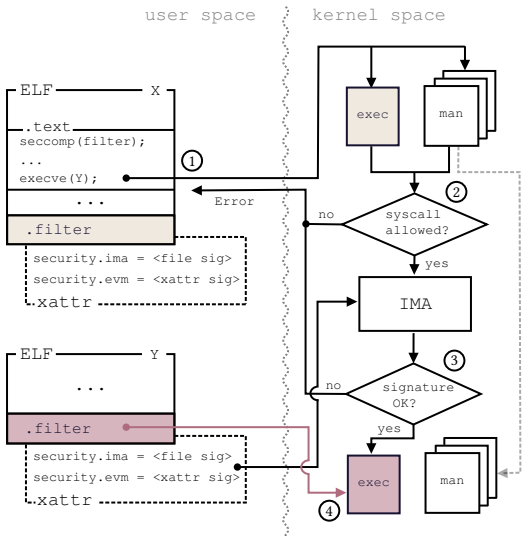


Figure 3: `SysXCHG` enforcement under the exchange model.

Offline, binaries are *measured* (i.e., hashed) and the measurements are signed with a private key and stored in the file's `security.ima` extended attribute. Further, a signature of the file's extended attributes is stored in `security.evm`. When the system is booted into appraisal mode (§2.4), opening an integrity-protected file triggers verification of the signatures in `security.{ima, evm}` using a public key from the system's keyring. If a file or its extended attributes have been tampered with, or lack a signature, verification will fail, and opening the file will be prohibited.

Recap. We now provide a summary of `SysXCHG`'s filter exchanging model using Figure 3 as reference. When a program `X` issues an `execve` syscall to run program `Y` (step ①), the system traps to the kernel where the `seccomp-BPF` infrastructure checks whether the syscall (i.e., `execve`) is allowed, using the installed `exec` and manual filters (step ②). If the `execve` is blocked, the appropriate blocked action is taken, as specified by the installed filters. If `execve` is allowed, the kernel's IMA subsystem checks whether the requested program (i.e., `Y`) is signed (step ③). Given a valid set of signatures in the file's `security.{ima, evm}` extended attributes, the kernel can perform the filter exchange, extracting `Y`'s `exec` filter(s) from its `.filter` section and replacing the list of previously installed `exec` filter(s) with new one(s) (step ④). Finally, the execution of `Y` can commence. Importantly, all manual filters previously installed (e.g., by `X`) are preserved across the call to `execve`.

4.3 Performance Considerations

Recent work in syscall filtering with `seccomp-BPF` [10, 18, 28, 29, 103] has focused on extracting and applying policies based solely on syscall numbers. Number-based filters are able to take advantage of `seccomp-BPF`'s recent bitmap cache optimization (§2.2), avoiding the overhead of executing (potentially many) BPF programs with each syscall. That being said, the bitmap cache has two main performance shortcomings. First, when installing a `seccomp-BPF` program, the program must be *emulated* for all syscalls that were previously allowed (i.e., their number-index corresponds to a 1 in the bitmap) to determine if the resulting action is cacheable (i.e., `SECCOMP_RET_ALLOW`). Second, while the bitmap cache may prevent `seccomp-BPF` programs from running, *multiple* function calls are required to reach the seccomp kernel infrastructure and test the cache. Any additional code executed on such a hot path will result in performance degradation. To address these shortcomings, `SysXCHG` includes a feature called *express filter* (`xfilter`).

Initially, we designed `xfilter` as a "fast path" for enforcing number-based syscall policies with minimal kernel additions (when compared with the size and complexity of `seccomp-BPF`). While doing so, we realized that `xfilter` does not incur additional overhead while filtering *and* it benefits other settings performance-wise, such as filter install time. Thus, `xfilter` offers three benefits: (1) it reduces the syscall filtering footprint in the kernel, (2) it combats the aforementioned deficiencies in `seccomp-BPF`, and (3) it optimizes number-based syscall filtering.

`xfilter` facilitates process-level filtering by providing each process with a different *view* of the kernel's syscall table, where handlers are replaced to provide filtering functionality. More specifically, every execution thread maintains a different view of the syscall table, which is used in place of the global syscall table when a syscall is invoked by the process. If no filter is installed, all handlers in the process' (i.e., `struct task_struct`'s) syscall table will be set to their default value, as defined by the global syscall table. In contrast, when a filter is applied to a process'/thread's syscall table, the handlers of filter-specified syscalls will be replaced with pointers to functions that perform filtering functionality.

`xfilters` are specified by a bit vector that can be computed prior to running a program, and they are installed with `prctl` or as an `exec` filter, embedded in an ELF binary's `.filter` section. In both cases, `xfilter` offers an optimization of filter *installation time* over `seccomp-BPF`: `xfilter` can determine the allowed set of syscalls prior to a program running, whereas `seccomp-BPF` must resort to emulating filters *at runtime* to enable/populate its bitmap cache optimization. In brief, `xfilter` facilitates applying filters quickly.

`xfilter` & Inheritance Model. `xfilter` strives to have similar default behavior as `seccomp-BPF`. Specifically, `xfilter` allows for the "stacking" of multiple filters, always choosing the most restrictive action for a given syscall; e.g., if one filter allows a syscall and another blocks it, the syscall's handler will correspond to the blocked syscall handler. Additionally, installed filters are inherited by a process' children across invocations of `[v]fork` and `clone`. Filters are also preserved across `execve[at]` by default, implementing the inheritance model that only allows privileges to shrink.

`xfilter` & Exchange Model. To provide filter exchanging support, `xfilter` marks which syscalls were blocked by `exec` filters installed automatically during `execve[at]`, and which were blocked by manual filters installed with `prctl`.

Compatibility with `Seccomp-BPF`. Our design of `xfilter` does not preclude the use of `seccomp-BPF`; a process can use both simultaneously. This allows for both filter interoperability and a separation of concerns, where filtering performance can be optimized: `seccomp-BPF` can be used to perform argument-based filtering and `xfilter` can perform number-based filtering, removing the need for `seccomp-BPF`'s bitmap cache.

Since `seccomp-BPF` filters currently provide more filtering functionality than `xfilter`, we do not anticipate that `xfilter` will entirely replace `seccomp-BPF`. Rather, we expect `xfilter` and `seccomp-BPF` to operate in tandem, adapting to certain environments and developer needs. In addition, future iterations of `xfilter` could support argument filtering via attaching small, targeted BPF programs to individual syscalls by adding a new dispatching handler at their slot in the per-process syscall table view that would run specific BPF program(s), validating arguments. This would stay in-line with the design philosophy of `xfilter` and provide an optimization to argument filtering—i.e., only small, packed BPF programs specifically associated with the currently invoked syscall are run instead of a generic `seccomp-BPF` program (or multiple) that apply to all syscalls. We plan on investigating this in the future.

An example of the interplay between `seccomp-BPF` and `xfilter` is given in Figure 4. In transition ① a `seccomp-BPF exec` filter is installed when program `B` is executed. Subsequently (②), a `seccomp-BPF`-based manual filter is installed with the `seccomp` syscall; once this manual filter is installed, it persists throughout the lifetime of the process. When another program is executed that contains a `seccomp-BPF exec` filter (③) *all* currently installed `exec` filters are removed, and the new one is installed in their stead. The same is true of `exec xfilters`, as in ④: the previously installed `seccomp-BPF exec` filter is removed, and the new `exec xfilter` is installed. Transitions ④ – ⑥ perform the same task as ① – ③ with `xfilters` instead of `seccomp-BPF` filters, illustrating that we maintain identical semantics across the two designs.

5 IMPLEMENTATION

System Support. We implemented `SysXCHG` atop the Linux kernel v6.0.8, which has support for syscall filtering with `seccomp-BPF` (including its bitmap cache optimization) and file integrity checking with IMA measurement and appraisal.

Syscall Policies. While our design of `SysXCHG` is *agnostic* to the method used to extract a syscall policy from a binary, our implementation uses `sysfilter` [18]. `sysfilter` is a *binary* analysis framework that *statically* extracts a *complete* set of syscalls a program requires for benign execution. We chose this framework because it is: (1) able to work on commercial off-the-shelf (COTS) binaries (i.e., no source code is required); (2) complete, preventing program breakage by forming a safe over-approximation of the allowed set of syscalls; and (3) scalable, as policies can be extracted on the order of seconds. Abhaya [75], Confine [27], Chestnut [10], *etc.*, can easily be used in lieu of `sysfilter`, if necessary.

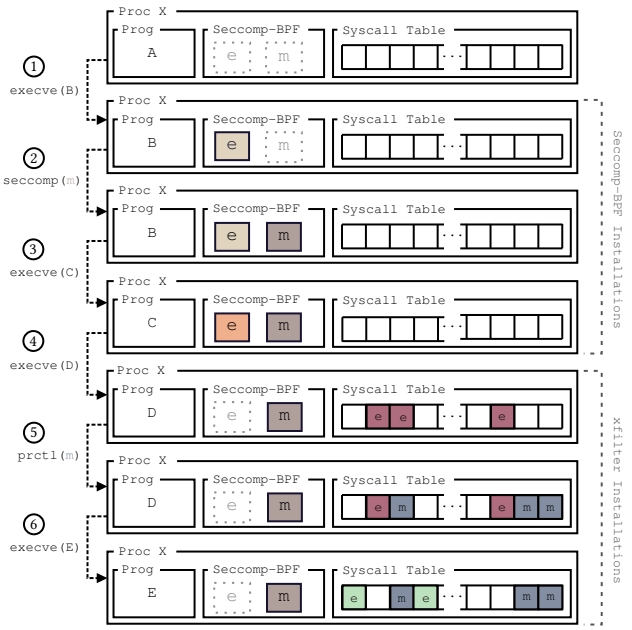


Figure 4: Installation semantics of `exec` and manual filters for both `seccomp-BPF` and `xfilter` (multi. program executions).

IMA Policy. To ensure the integrity of binaries prior to their execution, we validate the signatures stored in their `security.{ima, evm}` with IMA appraisal. Signature checking is performed with public keys (signed by the kernel's builtin, trusted keys) loaded onto the system's `.{ima, evm}` keyrings at boot.

To enable IMA appraisal, the system is rebooted with the kernel command-line option `ima_appraise=enforce`, and a policy is written to `ima/policy` in the `securityfs` pseudo file system. SysXCHG uses the following minimal policy: `appraise func=BPRM_CHECK fgroup=997 appraise_type=imasig`. This policy mandates that appraisal occurs for files that are about to be executed (`BPRM_CHECK` stands for "binary program check") *and* are owned by a group ID (GID) equal to 997. Further, `appraise_type=imasig` specifies that the file must have a digital signature in its `security.ima` extended attribute, not just a hash of the file. This policy removes the obligation to measure and appraise all files on a given system—files that are not executed or do not match the specified GID are not subject to appraisal—improving performance and flexibility without hampering SysXCHG's security guarantees. An approach similar to this is how we envision SysXCHG to be used in practice: it will be applied selectively in critical program-execution "chains."

We additionally enable EVM by writing to the `evm` pseudo file in `securityfs`; unlike IMA, EVM requires no command-line options to activate. Thus, the net result of this is that any program that is executed is ensured to be tamper-free having been approved during a trusted, offline phase.

5.1 Offline Tooling

We created a tool called `sysembed` to generate `exec` filters and add them to ELF binaries as a new section: `.filter`. It consists of ≈ 330 LOC (Python, Bash, and C). The inputs to the tool include the binary file to be protected, a JSON file specifying the syscall numbers to include in the filter (e.g., the output of `sysfilter` [18]), and the type of filter to generate: either `seccomp-BPF` or `xfilter`. For `seccomp-BPF`, `sysembed`: (1) extracts the syscall numbers from the JSON file, (2) constructs a `seccomp-BPF` program using a skip list-based approach (as in `sysfilter`), (3) writes the `BPF` program as a `struct sock_fprog` to a temporary file, before (4) using `objcopy` to embed the filter as an additional section in the input binary. We use the `-set-section-flags` option of `objcopy` to specify that the section should be read-only and not mapped into the program's address space. The process of embedding an `xfilter` is similar, only differing in the filter generation stage, creating a bit vector of allowed/blocked syscalls rather than a `BPF` program. After a filter is embedded in a binary, we use the command-line tool `evmctl` [63] to add IMA and EVM signatures to the binary's extended attributes.

We automated the steps of the offline tooling process above, in addition to generating syscall sets with `sysfilter`, in ≈ 100 LOC of Bash. Thus, (with or without our script) minimal developer effort is required to enable SysXCHG protection for a given binary. The offline process is typically performed once for a given binary; although if the binary is subsequently modified, enforcement will need to be performed again. In the best case, where modifications do not affect the binary's syscall set, nor its embedded filter(s), only updating the binary's signature is required. In the worst case, all offline steps must be performed again. While enforcing the benchmark applications (§6.1), we observed that embedding filters and signing binaries are performant while extracting syscall sets is a bottleneck. However, since SysXCHG's design makes no assumptions about the syscall extraction method, different tools can be substituted with `sysfilter` to further optimize performance if necessary.

5.2 Kernel Modifications

Our prototype of SysXCHG extends Linux kernel v6.0.8 with ≈ 600 LOC in C, added mainly to its `seccomp-BPF` infrastructure, ELF execution/loading, `prctl` syscall, and syscall handling pathway. Roughly half of the additions (≈ 300 LOC) went towards implementing `xfilter`, while the other half went towards extending `seccomp-BPF` with SysXCHG.

Handling exec Filters. We modified the kernel's binary loading code to search for the existence of a `.filter` section in ELF files while setting up a new program's address space. If the kernel finds a `.filter` section, it extracts it, formats it according to the filter type (e.g., `seccomp-BPF` versus `xfilter`), and installs it as specified by the enabled install model (i.e., inheritance versus exchange). Although our explanation of `exec` filters describes a single ELF filter section called `.filter`, in practice we use separate section names to signify different `exec` filter types to simplify our implementation. To implement the former, a small header of one byte can be prepended to filters in the `.filter` section denoting up to 256 unique filter types and flags (e.g., a `seccomp-BPF` filter with flag `SECCOMP_FILTER_FLAG_SPEC_ALLOW` to disable speculative store bypass mitigations for the filter program).

seccomp-BPF. The changes we made to the seccomp-BPF infrastructure to handle exec filters and both the inheritance and exchange models are minimal, resulting in ≈ 150 lines of additional code. For the inheritance model, after the kernel extracts a seccomp-BPF exec filter from an ELF binary, it installs it using the standard seccomp-BPF installation pathway. This simplicity is the result of the inheritance model making no distinction between exec and manual filters once they are in the kernel's internal representation. In contrast, to support the exchange model, SysXCHG creates separate lists of installed filters for exec and manual filters, as well as separate bitmap caches. When a new exec filter is installed, SysXCHG first clears the exec filter list and corresponding bitmap cache before repopulating them with the new program's filter.

xfilter. We represent xfilters in our implementation as a bit vector. As with seccomp-BPF filters, xfilters can be installed either manually with prctl, or automatically, as exec filters.

Every process in the kernel is represented by a struct task_struct, which we modify to include an additional struct that contains a pointer to a view of a task-specific syscall table as well as a reference count. In the case that no xfilters are installed, the syscall table pointer refers to the global syscall table. When an xfilter is installed for the first time, a copy of the global syscall table is created, and the task's syscall table pointer is made to point to the copy, where filter installation will occur. If the task's syscall table pointer does not refer to the global syscall table (i.e., an xfilter was previously installed), then its reference count is checked. If the count is one—i.e., only this process uses this syscall table—, filtering can proceed on the current syscall table without copying. If the count is greater than one—i.e., multiple processes share this syscall table as a result of [v]fork or clone—, the current syscall table is copied, and filter installation is done on the copy.

In the inheritance model, when a filter is installed on a syscall table, the function pointers in it (that correspond to blocked syscalls) are replaced with a pointer to a function that handles the illegal syscall. Similar to seccomp-BPF, exec and manual filters are treated equally in the inheritance model: they both can only reduce the set of allowed syscalls. Our implementation achieves this by never allowing a process to restore syscall handlers in its syscall table once they have been replaced with the illegal syscall handler.

In the exchange model, exec and manual filters are treated differently. To differentiate which entries in a given syscall table are blocked by exec and manual filters, we employ different illegal syscall handlers for each, called exec_blocked_syscall and man_blocked_syscall, respectively. When a manual filter is installed, we replace the handlers of any blocked syscalls with the man_blocked_syscall handler. In contrast, when an exec filter is installed, for each blocked syscall, we check whether it was previously blocked by a manual filter using pointer equality to compare the given syscall's handler with the address of man_blocked_syscall. No action is taken in the event that the syscall was previously blocked by an manual filter. In all other cases, the illegal syscall handler for exec filters, exec_blocked_syscall, can replace the current handler. Likewise, when a given syscall is allowed by an exec filter, we can restore the syscall's original handler (if it was previously blocked) iff a manual filter did not block it. Thus, this mechanism allows differentiation of filter types and ultimately enables filter exchanging for filters of type xfilter.

Table 1: Performance results for SPEC CPU 2017 using the inheritance model.

Benchmark	Seccomp-BPF		xfilter	
	Manual	Exec	Manual	Exec
600.perlbench_s	≈0%	≈0%	≈0%	≈0%
602.gcc_s	≈0%	0.44%	≈0%	0.33%
605.mcf_s	≈0%	0.37%	≈0%	0.29%
620.omnetpp_s	≈0%	0.34%	≈0%	≈0%
623.xalancbmk_s	≈0%	≈0%	0.11%	0.36%
625.x264_s	≈0%	0.12%	≈0%	0.09%
631.deepsjeng_s	0.02%	0.02%	0.38%	0.02%
641.leela_s	≈0%	0.02%	≈0%	0.05%
657.xz_s	≈0%	≈0%	≈0%	≈0%
619.lbm_s	0.14%	≈0%	0.05%	≈0%
638.imagick_s	0.09%	0.04%	0.04%	≈0%
644.nab_s	0.02%	0.07%	≈0%	≈0%

6 EVALUATION

Testbed. We evaluated SysXCHG on a host equipped with an Intel Xeon W-2145 8-core (16-thread) processor and 64GB of DDR4 memory, running Debian v11 (bullseye) Linux with kernel v6.0.8. In order to reduce benchmarking noise and assist with reproducibility, the CPU was configured to run at a fixed frequency of 3.7GHz in the C0 C-state, with dynamic voltage and frequency scaling (Intel Turbo Boost, Intel SpeedStep) disabled. To match common, real-world settings, we enabled both simultaneous multithreading and ASLR, and we built all binaries as position-independent (-f{PIC, PIE}, -pie). Finally, when installing seccomp-BPF filters (via prctl, seccomp, or as an exec filter) the SECCOMP_FILTER_FLAG_SPEC_ALLOW flag was always set to disable speculative store bypass mitigations (i.e., SysXCHG filters are considered to be non-adversarial).

6.1 Performance

6.1.1 Inheritance Model. We first evaluated the performance of SysXCHG under the inheritance model (i.e., no filter exchanging) by running a set of real-world and synthetic benchmarks. Regarding the latter, we chose the SPEC CPU 2017 benchmark suite to measure the *performance impact* of the two different filter installation methods (for both seccomp-BPF filters and xfilters): (1) manual filter installation via invoking prctl/seccomp in the constructor of a dynamic shared object (DSO), which the main binary loads (as in sysfilter [18]); and (2) exec filter installation, which uses the kernel to extract and install embedded filters (§4.1). Notably, under the inheritance model, manual and exec filters are treated equally *after* installation: seccomp-BPF maintains both manual and exec filters in the same append-only, kernel-resident list of filters, and xfilter only allows syscalls to be removed from a process' view of the syscall table. The results of this experiment are reported in Table 1 (≈0% corresponds to < 0.01%). In general, the SPEC CPU 2017 benchmark programs only invoke execve[at] once at the beginning of the benchmark, resulting in ≈1 filter installation. Hence, the contribution of *filter installation* to the overall performance impact in these benchmarks is minimal; we examine installation more thoroughly in Section 6.1.3.

Table 2: Performance results for real-world applications using the inheritance model (exec filters only).

Benchmark	Seccomp-BPF	xfilter
Nginx (1KB)	1.08%	0.07%
Nginx (100KB)	0.54%	1.10%
Nginx (1MB)	1.07%	1.11%
Redis (GET)	0.53%	0.26%
Redis (SET)	0.53%	0.26%
MariaDB	0.80%	0.50%
SQLite	≈0%	≈0%

Similarly, for the real-world application benchmark results (Table 2), start-up time (incl. filter installation) is not measured at all, and thus we omit manual filter installation and default to exec filters only. Overall, our results regarding the inheritance model demonstrate that xfilter performs equal to (or better than) seccomp-BPF when filtering is being performed. For SPEC CPU 2017 (Table 1), overheads range from ≈0% to 0.36% for xfilter, and ≈0% to 0.44% for seccomp-BPF. The real-world applications (Table 2) exhibit a slightly higher overhead, ranging from ≈0% to 1.11% for xfilter and ≈0% to 1.08% for seccomp-BPF. Table 1 shows that there is a *negligible* difference between manual and exec installation times, suggesting a performant exec filter implementation.

SPEC CPU 2017. We use the 12 C/C++ benchmarks from the SPEC CPU 2017 SPECspeed Integer and Floating Point suites [7] to evaluate the runtime slowdown of SysXCHG for both xfilter and seccomp-BPF filter types. For each enforcement variant in Table 1 (col. 2–5), we averaged the results of 10 iterations of the ref workload, and report the performance impact as a percentage atop a baseline that performs no syscall filtering. The results show negligible performance degradation in the range of 0%–0.14%, and 0%–0.44%, for seccomp-BPF manual, and exec filter installation methods, respectively; and 0%–0.38%, and 0%–0.36%, for xfilter manual, and exec filter installation methods, respectively. This demonstrates the performance *improvement* that can indeed be achieved with xfilter acting as a filtering shortcut.

Real-world Applications. To understand SysXCHG's performance impact in a realistic setting, we benchmarked four popular real-world applications: Nginx (v1.22.1) [73], Redis (v6.0.16) [85], MariaDB (v10.5.18) [68], and SQLite (v3.34.1) [97]. Results, averaged over 20 runs, are shown as a percentage atop a baseline that performs no syscall filtering in Table 2.

• *Nginx*: We measured the effects of xfilter and seccomp-BPF filters on the throughput of Nginx using the wrk [32] benchmarking tool to generate simultaneous, continuous HTTP requests to Nginx over a 1-minute period for three different file sizes (filled with random data): 1KB, 100KB, and 1MB. To ensure I/O did not mask SysXCHG's overhead, we ran Nginx and wrk on the same host and connected them over the loopback (lo) virtual network interface. We also tuned the benchmark for each file size to maximize CPU utilization. For all of the file sizes, wrk ran with 8 threads of execution, and with each thread making 64 simultaneous HTTP requests. For the 1KB and 100KB file sizes, Nginx ran with 8 worker threads, while for the 1MB file size it ran with 4.

The overhead of seccomp-BPF filtering ranges from 0.54%–1.08%, while the overhead of xfilter ranges from 0.07%–1.11%. Notably, our results show the worst-case. If the CPU is not saturated, I/O will mask any performance degradation resulting from SysXCHG.

• *Redis*: We used the memtier_benchmark [84] to generate a realistic workload for Redis consisting of a 1-minute stream of SET and GET requests (1 : 10 ratio) on a 32-byte object. Both memtier_benchmark and Redis ran on the same host and performed I/O over lo. We ran memtier_benchmark with 3 threads, each simulating 16 clients making simultaneous requests to drive the execution of a single redis-server thread, and we verified that these settings maximized the CPU utilization of the redis-server thread. The throughput degradation of xfilter and seccomp-BPF is *negligible* at 0.26% and 0.53%, respectively; xfilter is (slightly) better.

• *MariaDB*: We evaluated the throughput reduction in MariaDB using a simulated OLTP workload (oltp_read_write) generated by sysbench [55]. Similar to the other networked benchmarks, MariaDB and sysbench ran on the same host and communicated over lo. Each benchmark run lasted 5 minutes, wherein sysbench made simultaneous requests using 6 threads on a database consisting of a single table of 2M rows. We largely followed MariaDB's recommended settings for benchmarking with sysbench [67], with a small number of modifications to ensure CPU utilization was high and I/O was not shadowing results. Specifically, we used 75% of our machine's RAM (48GB) for the buffer pool, 512MB for the log buffer (≈11MB per GB of the buffer pool), and 12GB for the log file size (25% of the buffer pool). The impact of xfilter and seccomp-BPF on MariaDB's throughput is *inconsequential* at 0.50% and 0.80%, respectively. Again, xfilter is (slightly) better than seccomp-BPF.

• *SQLite*: We used SQLite's Speedtest benchmark [96] to measure the runtime increase of xfilter and seccomp-BPF across a series of standard database operations. We used an in-memory database for our evaluation, and the benchmark's default options. Our results show the performance overhead of SysXCHG is *unnoticeable*.

6.1.2 Exchange Model. We next evaluated the performance of SysXCHG's exchange model (§4.2) using the suite of benchmarks from PaSH [46], and both seccomp-BPF and xfilter. Specifically, we used nearly the entire suite of PaSH benchmarks, excluding only correctness tests (i.e., the POSIX test suite) and PaSH-specific micro-benchmarks. The included benchmarks represent real-world use-cases of processes that invoke execve[at] multiple times throughout their lifetime and that of their children, providing insights about the performance overhead contributed by *filter exchanging* and *signature verification*. In other words, while the PaSH benchmark suite was originally created to measure shell script performance, the programs within execute other programs in a *hierarchical* manner that is conducive to evaluating filter exchanging. In terms of signature verification, prior to running the benchmarks, we signed all executed (filter-carrying) binary programs with IMA and EVM signatures, and during benchmarking, we appraised the binaries before their execution using trusted X.509 public key certificates to verify both the IMA and EVM signatures. Further, we chose the smallest dataset for all benchmarks to reduce the chance of I/O masking any performance overhead. The results of 20 runs of this experiment are shown in Table 3 as a percentage increase atop a vanilla baseline (≈0% corresponds to $< 0.01\%$).

Table 3: Performance results for PaSH (exchange model).

Benchmark	Seccomp-BPF	xfilter
Common Unix One-liners		
bi-grams	6.02%	3.30%
diff	6.57%	0.84%
nfa-regex	7.21%	0.90%
set-diff	7.81%	0.69%
shortest-scripts	1.42%	0.34%
sort-sort	7.11%	0.36%
sort	6.38%	1.08%
spell	1.77%	0.37%
top-n	2.41%	≈0%
wf	2.54%	0.82%
Bell Labs Unix50	0.40%	0.39%
COVID-19 Transit Analytics	0.87%	0.78%
Natural-Language Processing	0.95%	0.51%
NOAA Weather Analysis	0.96%	0.88%
Wikipedia Web Indexing	0.34%	0.16%
Video Processing	0.43%	0.40%
Audio Processing	0.03%	0.05%
Program Inference	0.29%	0.12%
Traffic Log Analysis	1.11%	0.30%
PCAP Log Analysis	0.25%	0.31%
Genomics Computation	≈0%	0.18%
Encryption	0.29%	0.10%
Compression	≈0%	≈0%
AUR Package Compilation	2.74%	1.71%

Overall, our results demonstrate that performing filter exchanging (including verifying signed binaries) using `seccomp-BPF` filters and `xfilters` results in *marginal* runtime overhead for the programs we consider macrobenchmarks (i.e., every benchmark except *Common Unix One-liners*), ranging from ≈0% to 2.74% for `seccomp-BPF` and ≈0% to 1.71% for `xfilter` (col. 2–3; Table 3). Exchanging using `xfilter` is, on average, more performant than `seccomp-BPF`, which is clearly highlighted by the microbenchmark *Common Unix One-liners*, where `seccomp-BPF` overheads get as high as 7.81%, while `xfilter` only gets as high as 3.39%.

• *Common Unix One-liners*: This PaSH microbenchmark consists of a set of popular shell scripts that contain common Unix idioms, which can be run with a small dataset of ≈1MB. The programs all spawn multiple children that subsequently `execve[at]` to set up pipelines to process the input data, all of which run on the order of milliseconds. We consider these workloads informative of the time it takes to exchange filters during `execve[at]` in addition to filtering syscalls. Rows 2–11 in Table 3 detail the results of each individual one-liner for `xfilter` and `seccomp-BPF` filter exchanging. Overall, `xfilter` consistently *outperforms* `seccomp-BPF`; overheads associated with `xfilter` range from ≈0% to 3.30%, while those of `seccomp-BPF` are greater, ranging from 1.42% to 7.81%.

• *Others*: The remainder of the PaSH benchmarks (rows 12–25 in Table 3) provides insights about the overhead associated with other common, real-world programs that are subject to multiple filter exchanges. While the overhead of exchanging `seccomp-BPF` filters

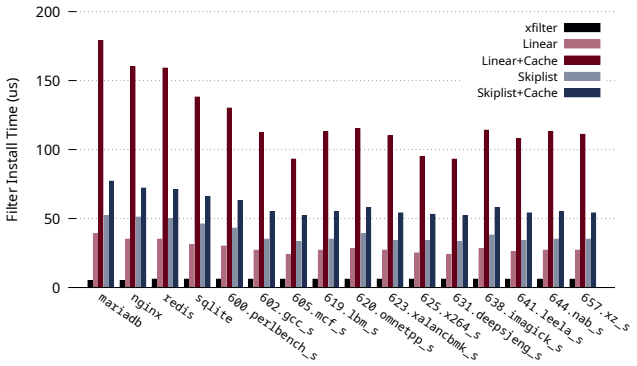


Figure 5: `xfilter` and `seccomp-BPF` filter installation times across real-world applications and SPEC CPU 2017.

and `xfilters` is small in both cases, being ≤ 2.84% and ≤ 1.71%, respectively, `xfilter` generally performs *better* than `seccomp-BPF`. In all but 3 cases, `xfilter` outperforms `seccomp-BPF`, and in those 3 cases, the difference between the two is negligible (≤ 0.19%).

6.1.3 Installation Time. To determine the *installation speedup* we achieve with `xfilter` over `seccomp-BPF`—both with and without the bitmap cache optimization—we constructed a microbenchmark to measure filter installation time of SPEC CPU 2017 and the real-world applications. The microbenchmark uses `rdtscp` to measure the time it takes to complete a manual filter installation. In order to reduce benchmarking noise, we pin the microbenchmark to a "quiet" CPU thread before timing its critical section.

Figure 5 shows the results of our microbenchmark across 10,000 installation iterations of `xfilter` as well as `seccomp-BPF` with and without the bitmap cache optimization, and with linear and skiplist-based filtering approaches [18]. Knowing the set of allowed and blocked syscalls prior to installing a filter, as is the case with `xfilter`, can greatly reduce filter installation time, ranging from ≈76% to ≈97%. Additionally, disabling `seccomp-BPF`'s bitmap cache significantly decreases installation time ranging from ≈29% to ≈36% for skiplist-based filters and ≈73% to ≈78% for linear-based filters. This reduction is the result of removing the `BPF` program emulation for every syscall that is required to populate the bitmap cache.

6.2 Effectiveness

The primary objective of `SysXCHG`, as described in Section 4, is to provide a syscall filtering enforcement mechanism that reduces both the attack surface of the kernel and access to unnecessary OS functionality (from the userland). The main component of `SysXCHG`'s design that enables this is filter exchanging, which performs syscall filtering in accordance to the PoLP. The remainder of this section evaluates how *effective* `SysXCHG` is at realizing these two claims using the PaSH [46] benchmark suite, as shown in Table 4 (that describes the `execve[at]` relationships in the PaSH benchmarks). For example, the *Common Unix One-liners* benchmark `shortest-scripts` (row 6) initially runs `bash` (depth = 1; col. 2) which spawns multiple child processes that `execve` other programs, contributing to over-privilege described in col. 7 (§6.2.1) and col. 8–9 (§6.2.2).

Table 4: Over-privilege of programs executed in the PaSH benchmark suite. Over-privilege percentage (col. 7) is calculated via $\frac{y-x}{x} \cdot 100$ where x = Exchange (col. 6) and y = Inheritance (col. 5).

Benchmark	Depth	Root	No. Desc.	Total Syscalls		Over-privilege		
				Inheritance	Exchange	Pct.	Critical	Functionality
Common Unix One-liners								
bi-grams	1	bash	11	122	84	45.24%	✓	stdio {c,d,r}path chown fattr id proc
diff	1	bash	6	109	84	29.76%	✓	stdio {c,d,r}path fattr id proc
nfa-regex	1	bash	3	100	84	19.05%	✓	stdio rpath id proc
set-diff	1	bash	7	109	84	29.76%	✓	stdio {c,d,r}path fattr id proc
shortest-scripts	1	bash	8	108	84	28.57%	✓	stdio rpath fattr id proc
↪	2	xargs	1	52	51	1.96%	✗	stdio
↪	2	xargs	1	60	51	17.65%	✓	stdio {c,tmp}path fattr proc
sort-sort	1	bash	3	102	84	21.43%	✓	stdio id proc
sort	1	bash	2	102	84	21.43%	✓	stdio id proc
spell	1	bash	7	106	84	26.19%	✓	stdio id proc
top-n	1	bash	5	115	84	36.90%	✓	stdio chown fattr id proc
wf	1	bash	4	103	84	22.62%	✓	stdio id proc
Bell Labs Unix50	1	bash	12	124	84	47.62%	✓	stdio rpath chown fattr id proc vminfo unix inet
COVID-19 Transit Analytics	1	bash	6	119	84	41.67%	✓	stdio chown fattr id proc vminfo unix inet
Natural-Language Processing	1	bash	14	128	84	52.38%	✓	stdio {c,r,w,tmp}path chown fattr id proc vminfo unix inet
NOAA Weather Analysis	1	bash	13	142	84	69.05%	✓	stdio chown {c,r}path fattr flock id proc protexec vminfo unix inet
↪	2	xargs	1	109	51	113.73%	✓	stdio {c,r,tmp}path fattr flock id proc protexec unix inet
↪	2	sh	1	77	68	13.24%	✗	stdio {c,tmp}path chown fattr
Wikipedia Web Indexing	1	bash	16	146	84	73.81%	✓	stdio {c,d,r,w,tmp}path chown fattr id proc protexec
Video Processing	1	bash	3	137	84	63.10%	✓	stdio {c,r}path chown fattr id proc protexec unix inet
Audio Processing	1	bash	3	173	84	105.95%	✓	stdio {c,r,w}path chown fattr flock id proc protexec unix inet
Program Inference	1	bash	2	129	84	53.57%	✓	stdio {r,w,tmp}path chown fattr id proc protexec
Traffic Log Analysis	1	bash	8	120	84	42.86%	✓	stdio rpath chown fattr id proc vminfo unix inet
PCAP Log Analysis	1	bash	6	135	84	60.71%	✓	stdio rpath chown fattr id proc protexec
↪	2	sh	1	83	68	22.06%	✓	rpath id proc stdio
Genomics Computation	1	bash	8	127	84	51.19%	✓	stdio rpath chown fattr id proc protexec
Encryption	1	bash	3	124	84	47.62%	✓	stdio{c,r}path chown fattr id proc protexec unix inet
Compression	1	bash	3	112	84	33.33%	✓	stdio {c,r}path chown fattr id proc
AUR Package Compilation	1	bash	69	176	84	109.52%	✓	stdio {c,d,r,w,tmp}path chown fattr id proc protexec vminfo unix inet settime
…	…	…	…	…	…	…	…	…
↪	2	sh	2	138	68	102.94%	✓	stdio {c,d,r,w,tmp}path chown fattr id proc protexec unix inet
…	…	…	…	…	…	…	…	…
↪	10	make	3	123	74	66.22%	✓	stdio {c,d,r,w}path chown fattr id proc
↪	11	collect2	1	58	46	26.09%	✓	stdio {c,tmp}path fattr

Two of the children the initial bash process spawns execute xargs (depth = 2; rows 7–8), which further execute other programs, justifying their place in the table. Col. 4 corresponds to the number of different programs execve[at]'d from each "root" (col. 3) program. At a high-level, any program that executes another has the potential to violate the PoLP in the inheritance model.

Overall, SysXCHG's exchange model is *extremely effective*, offering a huge improvement over the currently deployed inheritance model. For the PaSH benchmark suite, the inheritance model can result in up to 109.54% additional syscalls for a program atop what the program needs for isolated execution (i.e., no consideration for subsequently executed programs). This huge increase, which includes a plethora of functionality as well as critical syscalls, drops to 0% when employing the exchange model.

6.2.1 Kernel Attack Surface Reduction. An adversary (§3) that has taken control of a userland process—and thus has the ability to make arbitrary syscalls with arbitrary arguments—may seek to escalate their privileges by attacking the kernel, exploiting vulnerabilities in less-stressed syscalls [61]. Since it is infeasible to deduce a priori whether a given syscall and set of arguments can result in a vulnerability, we consider the *magnitude* of the reduction in a process' set of allowed syscalls a viable metric for determining SysXCHG's ability to reduce the attack surface of the kernel.

We applied this metric to PaSH and detail our findings in col. 7 of Table 4. For a given root program (col. 3), we calculated the total number of syscalls that must be allowed by filtering mechanisms under the inheritance model to provide correct, non-adversarial execution of itself and any descendant programs it executes (col. 5).

In addition to the number of syscalls allowed, we also show the corresponding percentage increase of syscalls (col. 7) atop what the exchange model requires (col. 6)—in this case, the exchange model represents the minimum number of syscalls the given root program requires for benign execution when run in isolation (i.e., without considering any descendant programs).

To calculate the percentage increase, we use the formula: $\frac{y-x}{x} \cdot 100$ where x = Exchange (col. 6) and y = Inheritance (col. 5). As an example, consider the root of *Audio Processing* (i.e., `bash`). When run under the currently-employed inheritance model, the root must allow 173 syscalls to accommodate its descendants, despite only requiring 84 for its own execution.

The exchange model enables the root to securely run with its minimum required set of 84 allowed syscalls, while the inheritance model mandates the root allow 173 syscalls, a 105.95% increase in over-privilege, which is significant.

6.2.2 Privilege Reduction. In addition to our quantitative approach to evaluate SysXCHG's effect on the kernel's attack surface, we also take a qualitative approach to evaluate the ability of SysXCHG's exchange model to *limit dangerous functionality* that is normally available to an attacker when the inheritance model is employed.

Given the set of syscalls that the inheritance model allows, and which the exchange model blocks (i.e., the syscall set allowed by the inheritance model minus the set allowed by the exchange model), we determine the *additional abilities* the attacker has at their disposal as well as whether any of these syscalls are *security critical*. To determine the former, we use the classifications (i.e., "promises") established by OpenBSD's `pledge` [66], which restricts the syscalls a process can make based on OS *functionality*. To determine the latter, we use the definition of critical syscalls (and their functional equivalents; e.g., `open`⇝`openat`) by Ghavamnia et al. [28], which considers a given syscall as critical if it shows up in real-world exploits. Together, these two metrics provide a qualitative view of the danger of the inheritance model, which is abated by SysXCHG.

Attacker Abilities. There are 165 unique syscalls across all PaSH benchmarks that are available to an attacker as a result of the inheritance model. Out of these, 82 map directly to a syscall in OpenBSD that is classified in a `pledge` promise. For the remaining 83 syscalls, we manually classified them according to similar semantics (e.g., `fsync` is classified as `stdio`, so we manually classified `fdatasync` as `stdio` as well).

Using this classification of syscalls to abilities, we categorized the type(s) of functionality available to an attacker under the inheritance model, as shown in column 9 of Table 4. In other words, by using SysXCHG's exchange model in place of the inheritance model, an attacker's capabilities would be limited in the areas of functionality described in column 9.

Critical Syscalls. We examined each row of Table 4 to determine whether any syscalls that are available to a process due to the inheritance model are security critical according to the classification of Ghavamnia et al. [28]. Our results are shown in column 8 of Table 4. The vast majority of programs run under the inheritance model in the PaSH benchmark suite include security critical syscalls, providing an adversary with dangerous abilities, and motivating the necessity of the exchange model.

7 CONCLUSION

We presented the design and implementation of SysXCHG: a syscall filtering enforcement mechanism that enables programs to run in accordance with the PoLP. We introduced a new primitive, the `exec` filter, which is embedded in the binary it applies to and is automatically installed upon `execve[at]`. Using `exec` filters we detailed the main thrust of the paper: the exchange model, which contrasts with `seccomp-BPF`'s currently employed inheritance model, by allowing a program to exchange a previously installed `exec` filter with its current one. We still maintained developer-intended semantics with manual filters, which do not have the ability to be uninstalled or exchanged. To combat security concerns of an adversary tampering with `exec` filters and increasing their privilege, we sign filter-carrying binaries using Linux's IMA subsystem. Additionally, we identified a shortcoming of `seccomp-BPF`'s filter installation procedure, which we remedied with the design of `xfilter`: a process-specific view of the kernel's syscall table.

In our evaluation of SysXCHG we: (1) measured the performance overhead of `exec` filters, filter exchanging, and `xfilter`; and (2) assessed the effectiveness of our design from a security standpoint. Our performance evaluation demonstrated the effects of filter exchanging are negligible, resulting in $\leq$ 2.74% and $\leq$ 1.71% for `seccomp-BPF` and `xfilter`, respectively, for the PaSH macrobenchmarks. However, the PaSH microbenchmark revealed that `xfilter` performs significantly better ($\leq$ 3.30%) than `seccomp-BPF` ($\leq$ 7.81%). Further, we showed that `xfilter` offers a substantial speedup for filter installation, ranging from ≈76% to ≈96%. Lastly, our effectiveness evaluation determined that programs that execute others are grossly over-privileged under the inheritance model. In contrast, the design of SysXCHG's exchange model removes all of the over-privilege associated with the inheritance model, providing a performant and effective scheme.

Availability

Our prototype implementation of SysXCHG is available at: https://gitlab.com/brown-ssl/sysxchg

ACKNOWLEDGMENTS

We thank our shepherd, Alexios Voulimeneas, and the anonymous reviewers for their valuable feedback. This work was supported in part by the CIFellows 2020 program, through award CIF2020-BU-04, and the National Science Foundation (NSF), through award CNS-2238467. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the US government, NSF, or CRA.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *ACM Conference on Computer and Communications Security (CCS)*. 340–353.

[2] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2019. Nibbler: Debloating Binary Shared Libraries. In *Annual Computer Security Applications Conference (ACSAC)*. 70—-83.

[3] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman. 1998. UFO: A Personal Global File System Based on User-Level Extensions to the Operating System. *ACM Transactions on Computer Systems (TOCS)* 16, 3 (1998), 207–233.

[4] Paul-Antoine Arras, Anastasios Andronidis, Luís Pina, Karolis Mituzas, Qianyi Shu, Daniel Grumberg, and Cristian Cadar. 2022. SaBRe: load-time selective

binary rewriting. *International Journal on Software Tools for Technology Transfer (STTT)* 24, 2 (2022), 205–223.
[5] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 335–348.
[6] Massimo Bernaschi, Emanuele Gabrielli, and Luigi V. Mancini. 2000. Operating System Enhancements to Prevent the Misuse of System Calls. In *ACM Conference on Computer and Communications Security (CCS)*. 174–183.
[7] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-generation Compute Benchmark. In *ACM/SPEC International Conference on Performance Engineering (ICPE)*. 41–42.
[8] Alexander Bulekov, Rasoul Jahanshahi, and Manuel Egele. 2021. Saphire: Sandboxing PHP Applications with Tailored System Call Allowlists. In *USENIX Security Symposium (SEC)*. 2881–2898.
[9] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium (SEC)*. 249–266.
[10] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. 2021. Automating Seccomp Filter Generation for Linux Applications. In *ACM Cloud Computing Security Workshop (CCSW)*. 139–151.
[11] Suresh N. Chari and Pau-Chen Cheng. 2003. BlueBoX: A Policy-Driven, Host-Based Intrusion Detection System. *ACM Transactions on Information and System Security (TISSEC)* 6, 2 (2003), 173–200.
[12] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-Oriented Programming without Returns. In *ACM Conference on Computer and Communications Security (CCS)*. 559–572.
[13] Microsoft Corporation. 2016. Seccomp security profiles for Docker. https://github.com/microsoft/docker/blob/master/docs/security/seccomp.md
[14] The MITRE Corporation. 2014. CVE-2014-0038. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0038
[15] The MITRE Corporation. 2017. CVE-2017-8824. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-8824
[16] The MITRE Corporation. 2021. CVE-2021-44228. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44228
[17] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *USENIX Security Symposium (SEC)*, Vol. 98. 63–78.
[18] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. 2020. sysfilter: Automated System Call Filtering for Commodity Software. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 459–474.
[19] Solar Designer. 1997. Getting around non-executable stack (and fix). https://seclists.org/bugtraq/1997/Aug/63.
[20] Daniel C. DuVarney, V. N. Venkatakrishnan, and Sandeep Bhatkar. 2003. SELF: A Transparent Security Extension for ELF Binaries. In *ACM New Security Paradigms Workshop (NSPW)*. 29–38.
[21] Catherine Easdon, Michael Schwarz, Martin Schwarzl, and Daniel Gruss. 2022. Rapid Prototyping for Microarchitectural Attacks. In *USENIX Security Symposium (SEC)*. 3861–3877.
[22] Stephanie Forrest, Anil Somayaji, and David H. Ackley. 1997. Building Diverse Computer Systems. In *Workshop on Hot Topics in Operating Systems (HotOS)*. 67–72.
[23] Timothy Fraser, Lee Badger, and Mark Feldman. 2000. Hardening COTS Software with Generic Software Wrappers. In *IEEE DARPA Information Survivability Conference and Exposition (DISCEX)*, Vol. 2. 323–337.
[24] Alexander J. Gaidis, Joao Moreira, Ke Sun, Alyssa Milburn, Vaggelis Atlidakis, and Vasileios P. Kemerlis. 2023. FineIBT: Fine-grain Control-flow Enforcement with Indirect Branch Tracking. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.
[25] Tal Garfinkel. 2003. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Network and Distributed System Security Symposium (NDSS)*.
[26] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. 2004. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Network and Distributed System Security Symposium (NDSS)*.
[27] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. 2020. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 443–458.
[28] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal System Call Specialization for Attack Surface Reduction. In *USENIX Security Symposium (SEC)*. 1749–1766.
[29] Seyedhamed Ghavamnia, Tapti Palit, and Michalis Polychronakis. 2022. C2C: Fine-Grained Configuration-Driven System Call Filtering. In *ACM Conference on Computer and Communications Security (CCS)*. 1243–1257.
[30] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, and Thomas E. Anderson. 1998. SLIC: An Extensibility System for Commodity Operating Systems. In *USENIX Annual Technical Conference (ATC)*.
[31] Adrien Ghosn, Marios Kogias, Mathias Payer, James R Larus, and Edouard Bugnion. 2021. Enclosure: Language-Based Restriction of Untrusted Libraries. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 255–267.
[32] Will Glozer. 2021. wrk – a HTTP benchmarking tool. https://github.com/wg/wrk.
[33] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out Of Control: Overcoming Control-Flow Integrity. In *IEEE Symposium on Security and Privacy (S&P)*. 575–589.
[34] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. 1996. A Secure Environment for Untrusted Helper Applications Confining the Wily Hacker. In *USENIX Security Symposium (SEC)*.
[35] Ivan Gotovchits, Rijnard Van Tonder, and David Brumley. 2018. Saluki: Finding Taint-style Vulnerabilities with Static Property Checking. In *Workshop on Binary Analysis Research (BAR)*.
[36] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-level Caches. In *USENIX Security Symposium (SEC)*. 897–912.
[37] Philip J. Guo and Dawson Engler. 2011. CDE: Using System Call Interposition to Automatically Create Portable Software Packages. In *USENIX Annual Technical Conference (ATC)*.
[38] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. 1998. Intrusion Detection Using Sequences of System Calls. *Journal of Computer Security* 6, 3 (1998), 151–180.
[39] Gerard J. Holzmann. 2015. Code Inflation. https://spinroot.com/gerard/pdf/Code_Inflation.pdf
[40] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. In *IEEE Symposium on Security and Privacy (S&P)*. 969–986.
[41] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. 2018. Block Oriented Programming: Automating Data-Only Attacks. In *ACM Conference on Computer and Communications Security (CCS)*. 1868–1882.
[42] Kapil Jain and R. Sekar. 2000. User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement. In *Network and Distributed System Security Symposium (NDSS)*.
[43] Jake Edge. 2015. A seccomp overview. https://lwn.net/Articles/656307/.
[44] Jonathan Corbet. 2005. Securely renting out your CPU with Linux. https://lwn.net/Articles/120647/.
[45] Michael B. Jones. 1993. Interposition Agents: Transparently Interposing User Code at the System Interface. *ACM Special Interest Group in Operating Systems (SIGOPS)* 27, 5 (1993), 80–93.
[46] Konstantinos Kallas, Tammam Mustafa, Jan Bielak, Dimitris Karnikis, Thurston H.Y. Dang, Michael Greenberg, and Nikos Vasilakis. 2022. Practically Correct, Just-in-Time Shell Script Parallelization. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 769–785.
[47] Dmitry Kasatkin, David Safford, and Mimi Zohar. 2010. An Overview of The Linux Integrity Subsystem.
[48] Guarav S. Kc and Angelos D. Keromytis. 2005. e-NeXSh: Achieving an Effectively Non-Executable Stack and Heap via System-Call Policing. In *Annual Computer Security Applications Conference (ACSAC)*.
[49] Vasileios P. Kemerlis. 2015. *Protecting Commodity Operating Systems through Strong Kernel Isolation*. Ph. D. Dissertation. Columbia University.
[50] The Linux Kernel. 2023. Syscall User Dispatch. https://docs.kernel.org/admin-guide/syscall-user-dispatch.html.
[51] Sungjin Kim, Byung Joon Kim, and Dong Hoon Lee. 2021. Prof-gen: Practical Study on System Call Whitelist Generation for Container Attack Surface Reduction. In *IEEE International Conference on Cloud Computing (CLOUD)*. 278–287.
[52] Taesoo Kim and Nickolai Zeldovich. 2013. Practical and Effective Sandboxing for Non-root Users. In *USENIX Annual Technical Conference (ATC)*. 139–144.
[53] Koen Koning, Herbert Bos, and Cristiano Giuffrida. 2016. Secure and Efficient Multi-variant Execution Using Hardware-assisted Process Virtualization. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 431–442.
[54] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P Kemerlis, and Michalis Polychronakis. 2018. Compiler-assisted Code Randomization. In *IEEE Symposium on Security and Privacy (S&P)*. 461–477.
[55] Alexey Kopytov. 2021. sysbench. https://github.com/akopytov/sysbench.
[56] Eduardo Krell and Balachander Krishnamurthy. 1992. COLA: Customized Overlaying. In *USENIX Winter Technical Conference*. 3–7.
[57] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea nd R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 147–163.

[58] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated Software Diversity. In *IEEE Symposium on Security and Privacy (S&P)*. 276–291.

[59] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. 2017. SPEAKER: Split-Phase Execution of Application Containers. In *International Conference of Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 230–251.

[60] Bo Li, Jianxin Li, Tainyu Wo, Chunming Hu, and Liang Zhong. 2010. A VMM-Based System Call Interposition Framework for Program Monitoring. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. 706–711.

[61] Yiwen Li, Brendan Dolan-Gavitt, Sam Weber, and Justin Cappos. 2017. Lock-in-Pop: Securing Privileged Operating System Kernels by Keeping on the Beaten Path. In *USENIX Annual Technical Conference (ATC)*. 1–13.

[62] Cullen Linn, Mohan Rajagopalan, Scott Baker, Christian S. Collberg, Saumya K. Debray, and John H. Hartman. 2005. Protecting Against Unexpected System Calls. In *USENIX Security Symposium (SEC)*. 239–254.

[63] Linux Integrity Project. 2020. evmctl - IMA/EVM signing utility. https://manpages.debian.org/bullseye/ima-evm-utils/evmctl.1.en.html.

[64] Linux Programmer's Manual. 2021. proc - process information pseudo-filesystem. https://man7.org/linux/man-pages/man5/proc.5.html.

[65] LWN.net. 2004. x86 NX support. https://lwn.net/Articles/87814/.

[66] System Calls Manual. 2022. pledge — restrict system operations. https://man.openbsd.org/pledge.2

[67] MariaDB. 2011. MariaDB Tools. https://github.com/MariaDB/mariadb.org-tools/blob/master/sysbench/run-sysbench.sh.

[68] MariaDB. 2023. MariaDB. https://mariadb.com.

[69] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX Winter Conference*.

[70] Terrence Mitchem, Raymond Lu, and Richard O'Brien. 1997. Using Kernel Hypervisors to Secure Applications. In *Annual Computer Security Applications Conference (ACSAC)*. 175–181.

[71] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. 245–258.

[72] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *ACM International Symposium on Memory Management (ISMM)*. 31–40.

[73] Nginx. 2023. Nginx. https://nginx.org.

[74] Koichi Onoue, Yoshihiro Oyama, and Akinori Yonezawa. 2008. Control of System Calls from Outside of Virtual Machines. In *ACM Symposium on Applied Computing (SAC)*. 2116–1221.

[75] Shankara Pailoor, Xinyu Wang, Hovav Shacham, and Isil Dillig. 2020. Automated Policy Synthesis for System Call Sandboxing. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*.

[76] Dinglan Peng, Congyu Liu, Tapti Palit, Pedro Fonseca, Anjo Vahldiek-Oberwagner, and Mona Vij. 2023. uSWITCH: Fast Kernel Context Isolation with Implicit Context Switches. In *IEEE Symposium on Security and Privacy (S&P)*. 2956–2973.

[77] Rob Pike and Brian Kernighan. 1984. Program Design in the UNIX Environment. *AT&T Bell Laboratories Technical Journal* 63, 8 (1984), 1595–1605.

[78] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P Kemerlis, and Michalis Polychronakis. 2020. xMP: Selective Memory Protection for Kernel and User Space. In *IEEE Symposium on Security and Privacy (S&P)*. 563–577.

[79] Niels Provos. 2003. Improving Host Security with System Call Policies. In *USENIX Security Symposium (SEC)*. 257–272.

[80] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *USENIX Security Symposium (SEC)*. 1733–1750.

[81] Anh Quach, Rukayat Erinfolami, David Demicco, and Aravind Prakash. 2017. A Multi-OS Cross-Layer Study of Bloating in User Programs, Kernel and Managed Execution Environments. In *ACM Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*. 65–70.

[82] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *USENIX Security Symposium (SEC)*. 869–886.

[83] Mohan Rajagopalan, Matti A. Hiltunen, Trevor Jim, and Richard D. Schlichting. 2006. System Call Monitoring Using Authenticated System Calls. *IEEE Transactions on Dependable and Secure Computing (TDSC)* 3, 3 (2006), 216–229.

[84] Redis. 2023. memtier_benchmark. https://github.com/RedisLabs/memtier{_}benchmark.

[85] Redis. 2023. Redis. https://redis.io.

[86] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert Van Doorn. 2004. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *USENIX Security Symposium (SEC)*. 223–238.

[87] Yasushi Saito. 2005. Jockey: A User-Space Library for Record-Replay Debugging. In *ACM International Symposium on Automated Analysis-Driven Debugging (AADEBUG)*. 69–76.

[88] Jerome H. Saltzer and Michael D. Schroeder. 1975. The Protection of Information in Computer Systems. *Proc. IEEE* 63, 9 (1975), 1278–1308.

[89] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *USENIX Security Symposium (SEC)*. 936–952.

[90] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *IEEE Symposium on Security and Privacy (S&P)*. 745–762.

[91] Albert Serra, Nacho Navarro, and Toni Cortes. 2000. DITools: Application-level Support for Dynamic Extension and Flexible Composition. In *USENIX Annual Technical Conference (ATC)*. 225–238.

[92] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*. 552–561.

[93] Dimitrios Skarlatos, Qingrong Chen, Jianyan Chen, Tianyin Xu, and Josep Torrellas. 2020. Draco: Architectural and Operating System Support for System Call Security. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 42–57.

[94] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. 2011. Rolecast: Finding Missing Security Checks When You Do Not Know What Checks Are. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 1069–1084.

[95] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. 2021. A Comprehensive Study of Bloated Dependencies in the Maven Ecosystem. *Empirical Software Engineering (EMSE)* 26, 3 (2021), 45.

[96] SQLite. 2023. Database Speed Comparison. https://www.sqlite.com/speed.html.

[97] SQLite. 2023. SQLite. https://www.sqlite.org.

[98] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal War in Memory. In *IEEE Symposium on Security and Privacy (IEEE S&P)*. 48–62.

[99] The Linux Kernel. 2023. Seccomp BPF (SECure COMPuting with filters). https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html.

[100] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *USENIX Security Symposium (SEC)*. 941–955.

[101] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 367–382.

[102] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 133–147.

[103] Yunlong Xing, Jiahao Cao, Kun Sun, Fei Yan, and Shengye Wan. 2022. The devil is in the detail: Generating system call whitelist for Linux seccomp. *Future Generation Computer Systems (FGCS)* 135 (2022), 105–113.

[104] Yunlong Xing, Xinda Wang, Sadegh Torabi, Zeyu Zhang, Lingguang Lei, and Kun Sun. 2023. A Hybrid System Call Profiling Approach for Container Protection. *IEEE Transactions on Dependable and Secure Computing (TDSC)* (2023).

[105] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. 2013. Chucky: Exposing Missing Checks in Source Code for Vulnerability Discovery. In *ACM Conference on Computer and Communications Security (CCS)*. 499–510.

[106] YiFei Zhu. 2020. seccomp: Add bitmap cache of constant allow filter results. https://lwn.net/Articles/834056/.

[107] Yves Younan, Wouter Joosen, and Frank Piessens. 2012. Runtime Countermeasures for Code Injection Attacks against C and C++ Programs. *ACM Computing Surveys (CSUR)* 44, 3 (2012), 1–28.

[108] Dongyang Zhan, Zhaofeng Yu, Xiangzhan Yu, Hongli Zhang, Lin Ye, and Likun Liu. 2022. Securing Operating Systems Through Fine-Grained Kernel Access Limitation for IoT Systems. *IEEE Internet of Things Journal (IoT-J)* 10, 6 (2022), 5378–5392.

[109] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical Control Flow Integrity and Randomization for Binary Executables. In *IEEE Symposium on Security and Privacy (S&P)*. 559–573.