Jenny: Securing Syscalls for PKU-based Memory Isolation Systems

David Schrammel, Samuel Weiser, Richard Sadek,
and Stefan Mangard, *Graz University of Technology*

https://www.usenix.org/conference/usenixsecurity22/presentation/schrammel

This paper is included in the Proceedings of the
31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Jenny: Securing Syscalls for PKU-based Memory Isolation Systems

David Schrammel
Graz University of Technology

Samuel Weiser
Graz University of Technology

Richard Sadek
Graz University of Technology

Stefan Mangard
Graz University of Technology

Abstract

Effective syscall filtering is a key component for withstanding the numerous exploitation techniques and privilege escalation attacks we face today. For example, modern browsers use sandboxing techniques with syscall filtering in order to isolate critical code. Cloud computing heavily uses containers, which virtualize the syscall interface. Recently, cloud providers are switching to in-process containers for performance reasons, calling for better isolation primitives. A new isolation primitive that has the potential to fill this gap is called Protection Keys for Userspace (PKU). Unfortunately, prior research highlights severe deficiencies in how PKU-based systems manage syscalls, questioning their security and practicability.

In this work, we comprehensively investigate syscall filtering for PKU-based memory isolation systems. First, we identify new syscall-based attacks that can break a PKU sandbox. Second, we derive syscall filter rules necessary for protecting PKU domains and show efficient ways of enforcing them. Third, we do a comparative study on different syscall interposition techniques with respect to their suitability for PKU, which allows us to design a secure syscall interposition technique that is both fast and flexible.

We design and prototype Jenny– a PKU-based memory isolation system that provides powerful syscall filtering capabilities in userspace. Jenny supports various interposition techniques (e.g., seccomp and ptrace), and allows for domain-specific syscall filtering in a nested way. Furthermore, it handles asynchronous signals securely. Our evaluation shows a minor performance impact of 0–5% for nginx.

1 Introduction

Today's software ecosystems are incredibly complex and depend on numerous interacting components, all of which increase the attack surface. An inadvertent mistake in a library could compromise the entire software stack [13, 50, 51]. From an attacker's perspective, the syscall interface plays an essential role in exploitation. Hence, safeguarding syscalls is a key defense strategy taken by both container environments [28] as well as process sandboxing engines found in modern browsers [26, 39]. However, running syscall filtering code in the kernel poses a security risk, as shown by dozens of privilege escalation vulnerabilities [43–46, 48, 49], and can typically only be applied on a *process level* but not constrain, e.g., a single library. Since cloud providers are also shifting from process isolation towards more fine-grained in-process sandboxing [10], syscall filtering needs to catch up.

Recent work [14, 27, 56, 61] pushes towards more efficient *intra-process* sandboxes that are backed by so-called Protection Keys for Userspace (PKU) [32], suggesting a significant performance improvement over traditional process-based sandboxing. Compared to SFI [57], they allow sandboxing unmodified binaries. Unfortunately, researchers [11] uncovered serious deficiencies in how the evaluated PKU-based sandboxes ERIM [61] and Hodor [27] safeguard the syscall interface. Also, Donky [56] only blocks memory-related syscalls, while SealPK [14] does not address syscall security at all. Until now, it is unclear whether syscall filtering can be solved both efficiently and securely for PKU sandboxes. Furthermore, nested PKU domains should be constrained appropriately by their parent domains, *i.e.*, a domain might want to disable any dangerous or unneeded syscalls for its sub-domain which runs untrusted 3rd-party code. E.g., the webserver nginx uses the compression library zlib, whch had arbitrary code execution vulnerabilities [40–42] in the past. Thus, nginx, and any of its modules, might want to constrain such libraries, to limit their exploitation potential. Independently of syscalls, PKU sandboxes face other security challenges such as secure signal handling.

In this work, we comprehensively investigate syscall filtering for PKU-based memory isolation systems. First, we identify novel syscall-based attacks that can bypass PKU isolation. We design appropriate syscall filters for protecting PKU domains against these syscall-based attacks.

Next, we provide a systematic study comparing different syscall interposition techniques with respect to their suitability for PKU on x86-64. We found that none of them match the

PKU needs, being either insecure or slow. We design a new syscall interposition technique that is both secure and fast.

We present Jenny, the first comprehensive PKU-based in-process isolation system offering dynamic syscall filtering in userspace. Filters can act on the same thread and also be nested across PKU domains. Jenny comes with filter rules for protecting PKU domains and also supports advanced filters such as file system protection. Jenny further supports different syscall interposition techniques. Moreover, Jenny is the first PKU system that supports secure signal handlers. Finally, we introduce novel multi-domain call gates needed to safeguard the PKU policy register on x86-64. Our evaluation shows a minor performance impact of 0–5% for nginx.

Contribution. We make the following contributions:

- We identify previously unknown syscall attacks on PKU-based isolation systems.
- We derive syscall filter rules necessary for protecting PKU-based isolation domains.
- We perform a comparative study of various syscall interposition techniques for their applicability with PKU and derive a new technique that is tailored for PKU.
- We design Jenny– the first comprehensive PKU-based isolation system that supports secure (same-thread) userspace syscall filtering, secure (async.) signal handling, and secure multi-domain PKU call gates for x86-64.
- We prototype and evaluate Jenny under different interposition techniques and filter rules, and open-source it [1].

Outline. Section 2 gives some background. Section 3 raises challenges, analyzes the syscall interface and derives filter rules. Section 4 handles syscall interpositioning. Section 5 presents our design, which Section 6 evaluates. Section 7 and 8 discuss limitations and related work, and we conclude in Section 9.

2 Background

Sandboxing is used to constrain potentially malicious or vulnerable code from affecting the rest of a system. While different forms of sandboxing exist (e.g., VMs, containers, processes), we focus on PKU-based *in-process* sandboxing due to its presumed performance gains. Similar to other in-process isolation techniques, PKU-based sandboxes must filter the syscall interface, which can be used to escape their sandbox. The remainder of this section presents syscall filtering techniques and and discusses PKU-based sandboxes.

2.1 Syscall Filtering

Syscall filtering can be employed to constrain a process (*i.e.*, sandbox) such that access to certain syscalls and kernel resources is blocked. Thus, it can also reduce the available attack surface by limiting exposure to potential kernel bugs. In Linux, a number of different mechanisms can be used to filter syscalls, which we explain in the following.

ptrace is used to inspect/manipulate a process. It allows intercepting the entry and exit of each syscall of a process (tracee) in a separate tracer process. While designed as a debug feature, it can be used for syscall filtering and sandboxing.

Seccomp-BPF. Seccomp [33] is a kernel mechanism allowing an application to constrain the usable syscalls to a minimum. seccomp-bpf [16] adds support for Berkeley Packet Filter rules. It can restrict (deny/allow) syscalls based on their syscall number as well as their arguments. Installed filters cannot be changed or removed, but it is possible to install additional filters to further constrain syscalls. Today, seccomp-bpf is heavily used for process-based sandboxing (e.g., in Firefox and Chrome) [9, 38]. In case a filter rule is triggered, seccomp-bpf can perform various actions, of which we list the most relevant in terms of syscall interception:

`SECCOMP_RET_TRACE` causes the kernel to notify a ptrace-based tracer process before executing the syscall. Thus, we denote this mechanism as seccomp-ptrace. Compared to ptrace, this is faster because not all syscalls have to notify the tracer process and the tracer-to-kernel communication overhead is smaller due to an improved interface design. The tracer can skip or resume the syscall and perform any ptrace actions.

`SECCOMP_RET_TRAP` sends a signal to the thread that triggered the syscall. Instead of executing the syscall, the registered signal handler is invoked. We dub this "seccomp-trap".

`SECCOMP_RET_USER_NOTIF`, added in Linux 5.0, allows to defer decisions to a (separate) userspace program. Like ptrace, the tracer program can inspect memory referenced in syscall arguments using the `/proc/PID/mem` interface. In the following, we refer to this method as "seccomp-user".

Syscall User Dispatch is a new syscall emulation mechanism that is added in latest Linux releases [60]. It is meant to improve the emulation performance of Windows applications on Linux systems. Syscalls on a certain made from a certain memory region are executed natively. Syscalls made from outside this region are dispatched back to a userspace signal handler, which can emulate or filter them. This is similar to `seccomp-trap`, but faster and more flexible.

2.2 Protection Keys for Userspace (PKU)

Protection keys for userspace (PKU) is a hardware mechanism to quickly change the effective permissions of memory pages from userspace. Page table entries (PTEs), which hold the read/write/execute permissions of a memory page, are tagged with so-called protection keys. A per-thread policy register holds currently active protection keys and their associated access permissions. On each memory access, the hardware matches the policy register against the protection key stored in the PTE to further constrain the access. Intel implemented this feature on their server processors since Skylake and calls it "Memory Protection Keys" (MPK). They use four bits

[1] https://github.com/IAIK/Jenny

to support 16 different protection keys. Its policy register named PKRU is 32-bit wide and holds write- and read-disable bits for each of the 16 protection keys. The PKRU register can be written directly in userspace via the WRPKRU (and XRSTOR) instructions, which allows quick permission changes to a range of memory pages without slow kernel interaction. For this reason, MPK has been used in the past for building efficient in-process isolation frameworks [27, 31, 56, 61].

However, MPK was not designed as a standalone security mechanism. Since the policy register can be written from userspace, sandboxed code may also manipulate it (e.g., via a code-reuse attack) and escape the sandbox. Hence, MPK must be combined with Write-XOR-Execute (W⊕X) and binary scanning to remove or neutralize unwanted WRPKRU and XRSTOR instructions as well as secure WRPKRU call gates [61].

PKU Sandboxes. Notable examples of PKU sandboxes based on Intel MPK are ERIM [61], Hodor [27], and Donky [56]. In this work, we focus on Donky, since Jenny builds upon its software framework. Donky [56] is a PKU system for RISC-V and x86. It proposes ISA modifications to forward protection key violations to an in-process monitor. By flipping a bit in the policy register when entering and exiting the monitor, Donky ensures that only the monitor can change the register. Thus, no binary scanning or W⊕X is needed. The same mechanism is used to trap syscalls to the userspace monitor. Donky software can also be used with Intel MPK but only in an insecure way since it does not protect the PKRU.

3 The Need for PKU-aware Syscall Filtering

Existing PKU-based sandboxes [27, 56, 61] do not safeguard the syscall properly. They lack on various aspects such as comprehensive filter rules to enforce sandboxing on the syscall interface, suitable syscall interposition techniques, expressive filtering logic for advanced use cases, and support for signals. After pinpointing some selected deficiencies, we formulate four challenges addressed in this work.

Selected Deficiencies. ERIM [61] uses highly specialized filters for enforcing a W⊕X policy but misses other attacks [11] and do not provide filters for protecting domain-owned memory. Connor et al. [11] augmented ERIM to filter dangerous syscalls, having an unacceptable overhead of 60 % throughput reduction for nginx due to the slow ptrace method.

Hodor [27] places various security enforcement and filtering logic inside the kernel. The security risk of doing so severely constrains application-defined syscall filters [43].

Donky [56] delegates syscalls back to a userspace monitor but requires hardware modifications. On native x86_64, Donky is not secure and can only naively deny syscalls but not filter them in a customizable way. Donky's basic syscall deny filters miss advanced attacks explained in the following.

Connor et al. [11] showed various syscall attacks on PKU sandboxes, four of which are highlighted in the following. (i) Some syscalls allow access to arbitrary memory bypassing PKRU permissions (e.g., process_vm_writev, ptrace). Likewise, the procfs pseudo-file at /proc/self/mem allows unrestricted memory access via file operations. (ii) Read-only memory can change if it is mutably backed, e.g., by a writable file. Thus, an attacker could generate insecure WRPKRU instructions by writing to a file. (iii) Untrusted domains could install seccomp filters to emulate PKU syscalls like pkey_mprotect and thus, disable its protection. (iv) Untrusted domains could register signal handlers and use the sigreturn syscall to elevate the PKRU to arbitrary privileges. Further, signal handlers cannot be secured in a multi-threaded application.

To sum up, existing PKU sandboxes do not sufficiently safeguard the syscall interface. Furthermore, none of them supports signal handling. Several other deficiencies hinder their adoption as well such as missing multi-threading support [61] or multiple-domain support [27, 61].

Open Challenges. This leaves several questions unanswered, and we identify the following four open challenges:

$\mathcal{C}1$ *How to obtain PKU-secure syscall filter rules?*

PKU sandboxes need to protect their sandboxing logic by safeguarding syscalls appropriately, which is non-trivial. While existing work presented some syscall-based PKU exploits, it is unclear if there exist other dangerous syscalls.

To better understand the syscall attack surface, we analyze the syscall interface of Linux and identify new PKU attacks. Then, we define PKU-protected resources and derive comprehensive and efficient filter rules for them. These filter rules protect not only the sandboxing logic but also different per-domain resources (Section 3.2–3.3).

$\mathcal{C}2$ *What are limitations of existing syscall interposition techniques w.r.t. PKU sandboxes, and how to overcome them?*

Any syscall filtering mechanism needs a way to interpose on syscalls made by the code and, in case of in-process filtering, delegate them to an in-process monitor. A fundamental limitation of many interposition techniques, such as seccomp-bpf, is the inability to distinguish PKU domains, thus creating a recursion problem: a syscall that is intercepted and handed over to the monitor must not be intercepted again while inside the monitor. Existing PKU-based sandboxes proposed hardware extensions [56] or suggested the use of the slow ptrace method [61] or custom kernel changes [27, 61] for enforcing syscall filter rules *in the kernel*. Obviously, the latter is counterproductive for PKU being designed as a *userspace* protection mechanism.

We analyze different syscall interposition techniques with respect to PKU-based sandboxes. Based on their deficiencies, we design an interposition technique that can securely and effectively delegate filter decisions back to the user program, denoted as “pku-user-delegate” (Section 4).

$\mathcal{C}3$ *Can we realize effective and comprehensive syscall filtering for PKU-based sandboxes?*

Recent work [11] suggests that proper syscall filters severely impact the performance because of inefficient syscall interception mechanisms and filtering every `open` syscall. For example, the most widely used technique seccomp-bpf cannot inspect string arguments (e.g., file system paths).

Apart from performance considerations ($\mathcal{C}$**2**), a decent syscall filtering mechanism needs to offer high flexibility in how developers can define filter rules, e.g., which data they can access and modify. Filters need to be aware of different PKU domains. Moreover, nesting of filter rules becomes an essential requirement when PKU domains can spawn their own child domains. Furthermore, syscall filtering should support multithreading, *i.e.*, have the possibility to do thread-specific filtering and act on behalf of the filtered thread. E.g., For ptrace to efficiently handle concurrent syscalls, one has to spawn a new tracer thread for each tracee thread. However support for nested filters (*i.e.*, filter code can execute syscalls, which in turn are filtered by their respective parents), is non-trivial as each normal thread would additionally require new tracer threads for each (possible) nesting level. Furthermore, filtering thread-specific syscalls is hard, since the filter code does not run in the original thread but in the tracer.

$\mathcal{C}$**4** *How can we securely combine PKU-aware sandboxing and syscall filtering with signal handling?*

Signals present a number of challenges to PKU sandboxing. Signal handlers cannot run on a PKU-protected stack, which makes them vulnerable to privilege escalation attacks [11]. Moreover, it is unclear how to filter signal-related syscalls and virtualize signals between multiple PKU domains, given that signals are a process- and thread-specific resource. Finally, various race conditions could occur from signal delivery to program resumption (e.g., due to asynchronous signals, multi-threading, and syscall impersonation).

Based on $\mathcal{C}$**1** and $\mathcal{C}$**2**, we design the first PKU-based sandbox supporting powerful syscall filtering, addressing the above challenges. To fulfill $\mathcal{C}$**3**, it allows domains to filter their own child-domains in a stateful, domain-aware, nested, and thread-specific manner (Section 5). Furthermore, to address $\mathcal{C}$**4**, we add comprehensive signal support to our PKU sandbox and show how to secure it.

3.1 Threat model

Our threat model is in line with other PKU sandboxes [27,61]. An unprivileged user process (e.g., a web server) wants to run untrusted code (e.g., a plugin) inside a sandbox and shield against it. We make no assumptions on the sandboxed code – it might contain exploitable vulnerabilities or be outright malicious, executing arbitrary code provided by an attacker. Our trusted computing base consists of trusted components in the sandboxing code (e.g., a PKU monitor, startup code, and the linker), the kernel, a tiny kernel module and the hardware. We assume that these parts are correctly implemented and free of exploitable vulnerabilities. We consider hardware flaws, side-channels, and fault attacks to be out of scope. While outside our threat model, we briefly discuss sandboxing of privileged processes in Appendix B.

The Donky paper [56] uses a different threat model, which requires hardware changes to protect PKU from misuse. While we use the Donky framework as a base, we extend it to be secure even on unmodified x86-64 CPUs. For this purpose, we added ERIMs binary scanning [61] and ensured that no unsafe `WRPKRU` occurrences exist in the trusted library or in the sandboxed code. To clearly differentiate between these threat models, we mark any filters and mechanisms relying on the Donky-proposed hardware changes with an asterisk "*".

3.2 New Syscall-based PKU Attacks

Previous work [11] has shown, not all syscalls honor the `PKRU` register by checking it before accessing userspace memory.

By carefully inspecting the syscall documentation, we identified several additional previously unknown exploitable syscalls across different resource categories (*i.e.*, memory, process, files). While automating such an analysis on the syscall interface would be desirable to target completeness, it is outside the scope of this paper.

Memory-related syscalls. The `madvise` syscall is typically used to improve performance by giving the kernel additional memory information. We found that it can also be misused to clear memory pages, irrespective of their associated protection keys. Thus, malicious domains could erase otherwise inaccessible memory pages from other domains. We developed exploits using the flags `MADV_FREE`, `MADV_DONTNEED`, and `MADV_WIPEONFORK`, which all bypass PKU permissions.

`brk,sbrk` are typically used for managing heap memory, but can be exploited, as follows. PKU systems typically allow domains to allocate and protect private memory. If such memory lies in the heap area, malicious domains can use `brk,sbrk` to de-allocate and re-allocate that memory. This does not only wipe memory content but also removes any associated protection key, bypassing the current `PKRU` value.

With `userfaultfd`, one can handle page-faults in userspace. Surprisingly, we found that `userfaultfd` allows to write arbitrary values to any PKU-protected but not-in-memory page. When using `mmap` to request zero-filled memory, the kernel typically maps these pages on-demand when a page fault occurs. However, an attacker registering a userfaultfd handler can replace content of memory pages with arbitrary data as soon as the victim domain accesses it the first time. When combined with `madvise`, one could even overwrite previously non-empty pages.

Process-related syscalls. `(v)fork,clone` create a process copy that can (optionally) run in the same address space (*i.e.*, threads). Since not all resources are preserved, this can lead to synchronization issues and undefined behavior. Most notably, other threads, locks, timers, signals, and some memory areas (e.g., `madvise(MADV_DONTFORK)`) are not preserved.

exec-like syscalls are incompatible with in-process sandboxing since they replace the currently running program. In-process monitors are not preserved by default. However, some kernel-based restrictions remain active (e.g., seccomp filters).

arch_prctl and set_thread_area can be used to re-map the thread-local storage. Donky [56] relies on thread-local storage for trusted metadata. It further assumes that an isolated domain cannot alter the location of the thread-local storage (*i.e.*, the fs/gs registers). Using these syscalls, an attacker can escape its sandbox. Hodor, on the other hand is secure against this attack since it does not rely on any registers. Instead they describe a gettid-like call to look up thread-specific data.

Other syscalls can affect the entire process and the behavior of future syscalls. E.g., personality(READ_IMPLIES_EXEC) makes all future readable mapped memory also executable. Thus, arbitrary WRPKRU instructions can be injected and executed, which bypass the PKU sandbox.

File-related syscalls operating on file paths and file descriptors can directly compromise PKU sandboxes (cf. Section 3).

We found that core dumps present a serious information leakage vulnerability for PKU sandboxes. They are usually created during a program crash and contain the recorded state of the whole program's memory and register state before the crash. An untrusted domain could trigger the creation of a core dump via the kill syscall, or by accessing invalid memory. When the program is later restarted, the untrusted domain can open the core dump file and read any previously protected memory of other domains. An attacker could combine this vulnerability with fork: By crashing the child process, they can immediately read the core dump in the parent program.

3.3 Securing PKU with Syscall Filters

A secure PKU system must protect monitor and domain resources from unauthorized access. In the following, we outline these resources, and define filter rules to protect them.

We define monitor resources as anything that affects its security. This covers the *monitor code and data* to manage domains and threads, including parts of the thread-local storage. the *protection keys* used to protect the monitor code and data, as well as the *policy register* (*i.e.*, PKRU), and *process-wide resources* (e.g., signal handlers). Similarly, domain resources consist of any domain-owned memory, their protection keys, any created child-domains, and registered signal handlers and syscall filters. A PKU sandbox must protect these resources by fulfilling the following requirements:

Monitor code and data must be protected from unauthorized access on three layers: First, in memory (e.g., by protecting it with protection keys); Second, its memory mapping (*i.e.*, rwx permissions and protection keys); and third, on disk, if the mapping is file-backed (cf. [11]). Notably, this also includes any aliasing (e.g., shared memory or symlinks).

In addition, the PKRU policy register must be protected with secure call gates, either via hardware modifications as proposed in [56], or by ensuring no unsafe WRPKRU (and XRSTOR) instructions exist in executable memory [27, 61]. The latter must be protected on all three layers again: First, in memory by scanning it for unsafe instructions before marking it executable. Second, the memory mapping must enforce W⊕X, also for shared memory. And third, if the memory is backed by a file, W⊕X must be enforced there as well.

Syscall Filter Rules for PKU Sandboxes. We design two sets of syscall filter rules to enforce the above requirements: i) *base-mpk* for current x86-64 CPUs and ii) *base-donky** which is only secure with Donky's proposed hardware changes [56].

base-mpk. Donky already protects *domain code and data* on the first two layers. We additionally define syscall filters for mmap(2), (pkey_)mprotect, mremap, munmap, madvise and forward them to the monitor. To protect the third layer, we intercept mmap, mprotect and disallow shared executable mappings, since the in-process monitor cannot prevent other processes from modifying it (cf. [11]). Additionally, we ensure that executable memory is not backed by a writable file. For this we have three options. 1) disallow the mapping if the user can modify the file (either the content or its filesystem permissions). 2) remove the write-permissions from the file and optionally copy the file beforehand to avoid unnecessary changes to the system. Note, these two options requires tracing syscalls that can change file permissions (e.g., chmod). or 3) read the file into memory using mmap(MAP_ANON). For Jenny, we use the first option. To solve issues with core dumps and the procfs (cf. Section 3.2), we make use of an existing kernel feature, namely prctl(PR_SET_DUMPABLE, SUID_DUMP_DISABLE), which disables core dumps and access to the procfs. Hence, we do not need to filter any file or path-based syscalls suggested by [11], allowing more efficient filtering.

We protect the *policy register* by scanning the memory for unsafe instructions when executable mappings are created or changed (mmap, mremap, mprotect). For this, we use ERIM's binary scanner [61] and check for common pitfalls [11]. We enforce Write-XOR-Execute both on the memory mapping as well as any aliases (see above). Similar to memory-related syscalls, we safeguard *memory protection keys* by also intercepting pkey_alloc and pkey_free.

Finally, we protect *process-wide resources* by filtering fork, clone, signal, sigaltstack, sigreturn, sigaction and their relatives. We also deny syscalls that can modify the behvaiour of the entire process, and thus compromise the sandbox [2].

base-donky* shares the same filters as base-mpk, except that it does not generally enforce W⊕X. Instead we only protect the monitor code (and its file-backing) from modifications.

[2] remap_file_pages, process_vm_readv, process_vm_writev, ptrace, seccomp, shmat, shmdt, execve(at), personality, userfaultfd, add_key, request_key, keyctl, (arch_)prctl, set_thread_area, set_tid_address, umask, setpgid, setsid, modify_ldt, _sysctl, unshare, rseq

Table 1: Comparison of Syscall Filtering Mechanisms

	Filter can	Linux seccomp-bpf	Linux seccomp-user	Linux seccomp-trap	Linux ptrace(-seccomp)	Linux sysc. user dispatch	RISC-V user interrupts [56]	**Our libc-indirect***	**Our pku-user-delegate**
	Run in kernel (S), user (U)	S	S/U	S/U	U	U	U	U	U
Delegate	PKRU-aware delegation	n.a.	—	—	✓	—	✓	✓	✓
	Intercept pre-syscall	✓	✓	✓	✓	✓	✓	✓	✓
	Allow syscall	✓	—	—	✓	✓	✓	✓	✓
	Intercept post-syscall	—	n.a.	n.a.	✓	✓	✓	✓	✓
Emulate	Read syscall arguments	✓	✓	✓	✓	✓	✓	✓	✓
	Write syscall arguments	—	n.a.	n.a.	✓	✓	✓	✓	✓
	Read/Write any memory	—	✓	✓	✓	✓	✓	✓	✓
	Read/Write PKRU register	—	—	✓	✓	✓	✓	✓	✓
	Read/Write other registers	—	—	✓	✓	✓	✓	✓	✓
	Manipulate syscall return value	✓	✓	✓	✓	✓	✓	✓	✓
	Perform arbitrary syscalls	—	—	—	✓	✓	✓	✓	✓
Impersonate (threading) syscalls		—	—	✓	✓	✓	✓	✓	✓
Kernel context switches on deny		1	2	2	4+	2	0	0	1

Apart from this, domains can still map writeable and executable memory. Note, however, that this filterset requires the hardware modifications proposed in Donky [56] to protect the PKRU register, since we do not employ binary scanning here.

Extended Domain-aware Filter Rules. Domains may also use other resources such as sensitive files (e.g., databases or private keys), which should be shielded from other domains. To protect them, we safeguard file descriptors and file-system accesses for each domain, as explained in the following.

Based on *base-mpk*, we define a so-called ***localstorage*** filter set, where we additionally filter any path-based and file-descriptor-based syscall. We create a so-called local storage directory for each domain and constrain path-based syscalls to only access a domain's own local storage. We transparently modify all path-arguments in syscalls to point to this directory. E.g., `/tmp/file` is translated to `/tmp/localstorage[PID]/[domain id]/tmp/file`. When a domain invokes a syscall that creates a file-descriptor, we assign that domain as the owner. We filter syscalls such that domains can only use file-descriptors that they have access to. We used strace's [58] annotation of syscalls to automatically find all file-related syscalls. We discuss possible compatibility issues in Section 7.

4 Syscall Filtering Mechanisms

Traditionally, syscall filtering is used for debugging, emulation, malware inspection, or process isolation. Hence, existing syscall filtering mechanisms are either not usable as a security mechanism or not designed for *in-process* isolation via PKU.

In this section, we first outline requirements for comprehensive PKU-aware syscall filtering. We analyze various syscall filtering mechanisms w.r.t. these requirements and detail their limitations. Finally, we design *pku-user-delegate*, a better mechanism suitable for PKU sandboxing on x86-64 systems. Our minimal kernel module is both, fast and secure, by shifting all filter decisions to the userspace monitor, which avoids unnecessary code complexity (*i.e.*, potential for vulnerabilities) in the kernel.

Requirements Analysis. A comprehensive syscall filtering mechanism for PKU sandboxes shall meet four requirements, namely *delegation*, *emulation*, *impersonation*, and *nesting*.

Delegation allows syscalls to be filtered in userspace, e.g., by a in-process monitor, instead of the kernel. Thus, the user-provided filter code is never executed in the kernel, reducing kernel complexity and attack surface [43]. A filter shall be able to resume (*i.e.*, allow) the delegated syscall. Furthermore, delegation needs to be PKRU-aware such that trusted domains can issue syscalls without being (re)intercepted.

Emulation denotes the expressiveness of the filtering logic. E.g., seccomp-bpf filters are highly constrained and cannot perform deep argument inspection of strings provided to path-based syscalls. Ideally, filters can emulate arbitrary behavior, for which they need read and write access to the CPU registers and to application memory. This also includes the possibility to perform arbitrary syscalls in the filter context.

Impersonation enables filter code to perform syscalls on behalf of the filtered domain, which is needed for two reasons: First, we anticipate to fully delegate syscalls into userspace. If filters want to allow a particular syscall, they need to reissue it themselves. Since the kernel adheres to PKU permissions on the syscall arguments, the PKRU register needs to be impersonated to hold the filtered domain's protection keys. Second, a filter might want to execute additional syscalls other than the intercepted one in place of the filtered domain.

Moreover, impersonation should happen on the *same* thread as the filtered syscall to avoid issues with thread-specific syscalls (e.g., scheduling, locking, thread-local storage).

Nesting allows hierarchical syscall filtering, where each domain can filter their child domains in a nested way. E.g., a third-party module that is syscall-filtered by the application can further constrain the syscall interface for its own libraries. This helps to enforce the *principle of least privilege*. Nesting combines all previous requirements, from delegation over emulation to impersonation.

Performance of syscall filtering strongly depends on the number of context switches to the kernel. Since not all investigated filtering mechanisms can allow a syscall, we investigate the context switches for denying a syscall.

4.1 Comparison

Table 1 shows our comparative study of the syscall filtering mechanisms we analyzed.

Performance. seccomp-bpf filters run entirely in the kernel. seccomp-user, seccomp-trap and Syscall User Dispatch forward the decision making to userspace, for which a second kernel context switch is required when exiting the filter. ptrace requires additional syscalls for reading and writing registers. RISC-V User-Interrupts [56] and libc-indirect* require no kernel invocation but rely on custom hardware extensions. Our pku-user-delegate comes with a single kernel invocation.

seccomp-bpf filters run entirely in the kernel. Thus, syscall emulation is severely constrained. Filters only inspect register arguments and the syscall number but cannot access arbitrary memory or dereference pointers needed for deep argument inspection [17]. Newer syscalls (e.g., `clone3`, `openat2`) use structs for extensibility. This makes it impossible to filter these syscalls comprehensively. So-called extended BPF (eBPF) could solve this issue, providing filters with a state that is shared with the userspace and allowing dereferencing pointers. However, it is unlikely that eBPF will be soon used for seccomp [12]. Moreover, they cannot access the `PKRU` register needed for PKU awareness. Finally, filters cannot be relaxed but only constrained further by chaining additional filters.

seccomp-user delegates syscalls to a separate userspace thread/process with no option to securely allow the syscall [6]. Thus, the filter must emulate all intercepted syscalls manually on a different thread, which is problematic. For example, thread-specific syscalls cannot be easily emulated from another thread. Moreover, filters cannot read or change the CPU register state of the intercepted syscall. The inability to access the `PKRU` register hinders distinguishing PKU domains.

seccomp-trap delegates syscalls to userspace via signals. However, signal handlers need to be secured against privilege elevation attacks [11] by using our protection of signal stacks (cf. Section 5.6). seccomp-trap cannot allow or issue intercepted syscalls but needs to emulate them since the delegation logic (implemented in seccomp-bpf) is PKU-unaware.

ptrace runs before and after a syscall and is by far the slowest technique. However, unlike seccomp variants, it supports all features for PKU-based syscall filtering. In particular, it can divert the syscall directly to our monitor. By combining ptrace with seccomp [11, 27, 61], one can increase performance and only interpose those syscalls for which seccomp filter rules do not suffice (e.g., deep argument inspection or emulation).

Syscall User Dispatch lets syscalls within a certain dispatcher region unfiltered, which is deemed insecure: An untrusted domain could simply jump to a `syscall` instruction within this region to bypass syscall filtering [60]. We cannot use PKU to protect the dispatcher region from misuse, since PKU does not limit code fetches. However, Syscall User Dispatch alternatively allows to define a userspace variable (e.g., a monitor-mode flag) that decides whether syscalls are filtered or not. The monitor flips this flag when entering or exiting and protects it by means of a read-only protection key.

Unfortunately, Syscall User Dispatch is incompatible with handling signals, since the proposed monitor flag cannot be atomically flipped upon signal return.[3] Independently, Syscall User Dispatch itself makes use of (unsafe) signals to dispatch syscalls, for which it requires additional safeguards, as discussed for seccomp-trap.

RISC-V User-Interrupts and libc-indirect*. The modified RISC-V hardware of Donky [56] allow to interpose syscalls without kernel interaction. To approximate its performance on x86-64, we implement a method named "*libc-indirect**". We modify the libc such that all `syscall` invocations are delegated to the monitor unless we are already in the monitor. For existing x86-64 CPUs, this method is not secure on its own, since a domain can also execute uninstrumented syscall instructions, thus bypassing our libc delegation.

4.2 PKU-suitable Syscall Interposition

Given our analysis, only a few mechanisms support comprehensive PKU sandboxing, namely seccomp-ptrace and Donky RISC-V user-interrupts [56]. Unfortunately, seccomp-ptrace is too slow in practice, Syscall User Dispatch uses insecure signals, and user-interrupts are not supported by x86-64.

pku-user-delegate is a simple yet powerful syscall interposition technique we devise to overcome these limitations and support native x86-64. In its core, we delegate syscalls back to userspace similar to "Syscall User Dispatch". However, by avoiding signals we only require a single kernel invocation.

We implement our technique in a loadable kernel module (LKM) that only intercepts syscalls registered by the PKU monitor. We set a bit in the thread_info struct to selectively intercept protected threads, while ignoring other threads and processes. It furthermore checks if the thread is currently in the monitor by reading the `PKRU` register. Monitor syscalls are directly allowed, while others are delegated to our userspace by rewriting the program counter to point to the PKU monitor's syscall handler and storing the return address in the `RCX` register. Thus, pku-user-delegate closely mimicks the `syscall` instruction. Afterward, the monitor continues in the previous domain without invoking another switch to the kernel. In total, our module has 319 LoC.

Prior work also relied on kernel code, however, for a different purpose. ERIM [61] used a kernel module for enforcing W⊕X. Hodor [27] modified the kernel to enforce essential parts of their protection logic. Donky [56] used a kernel module for denying memory-related syscalls but not delegating them to userspace.

In contrast, pku-user-delegate delegates syscalls *back to userspace* for making all decisions in the monitor. Shifting complex filtering logic to the userspace has two main advantages: First, it reduces the kernel's attack surface by minimizing (interpreted) code in the kernel. E.g., BPF has had dozens of vulnerabilities that lead to kernel privilege escalation [43–46, 48, 49]. Second, the filter code is not constrained by BPF's limited instruction set and they can access (read-only) domain-metadata, which allows domain-aware filtering.

[3] The relevant `rt_sigreturn` syscall is issued inside the monitor but resumes outside of it. We can neither flip the flag before or after `rt_sigreturn`.

Third, the filter rules can be extended without requiring kernel changes. E.g., new struct-based syscalls can be easily filtered.

5 Jenny: Secure and Efficient Syscall Filtering for PKU Systems

In this section, we design Jenny– the first comprehensive PKU-based isolation system that supports secure and efficient syscall filtering, signal handling, and multi-domain PKU call gates. Jenny extends the open-source PKU framework Donky [56] with the necessary logic for registering and invoking syscall filters and signal handlers. Furthermore, our filters are not only PKU-domain-aware but also allow hierarchical syscall filtering across domains. That is, parent domains can hook arbitrary syscalls of their child domains with arbitrary nesting levels. Moreover, syscall filters can "impersonate" their filtered domains, that is, issue syscalls on their behalf. Jenny offers a generic programming interface for user-defined syscall filters, supporting different filtering mechanisms (cf. Section 2.1), of which we implement suitable ones.

To work with native x86-64 CPUs, we incorporate W⊕X, binary scanning, and secure call gates [27, 61] that can deal with multiple PKU domains. Finally, we give a security argumentation and design PKU-secure signal handling.

5.1 Design

To protect the PKU system, the base filters (*base-mpk* or *base-donky**) always run in the monitor. Additionally, domains can register their own filters for their child-domains. Each filter can define code to be run before ("enter-filter") and after ("exit-filter") a syscall. As shown in Figure 2 filters can be nested, allowing parent domains to filter syscalls of their children. The filters can contain arbitrary code and make syscalls themselves, which might be filtered by their respective parent domains. A simple example is given in Appendix C. We also provide facilities for so-called "impersonation" such that the filters can issue syscalls on behalf of the filtered child domain. Before doing the actual impersonated syscall inside the monitor, we switch to the PKRU value of the child.

Figure 1 shows syscall filtering for three domains: the monitor (green), the "parent" domain (blue), and the "child" domain. The child domain issues a syscall instruction (1), which is then trapped to the monitor. The monitor saves the user-stack and checks for installed syscall filters, beginning with the currently active domain over the parent tomain until we reach the monitor domain. A filter might be set to allow or deny or point to an filter function for emulating the syscall. If a filter function exists, we switch to the filter domain (2). The enter filter can inspect and manipulate the syscall arguments before re-issuing (impersonating) the syscall (3). From there, the same procedure repeats, but now for the parent-domain instead of the child. When a parent-domain has no parent itself, the monitor runs its own filters to protect all domains. Monitor-issued syscalls are not intercepted again. After a syscall (4), the exit-filter can inspect and manipulate the return value before returning (5). Finally, the monitor resumes the original child domain after the syscall instruction (6).

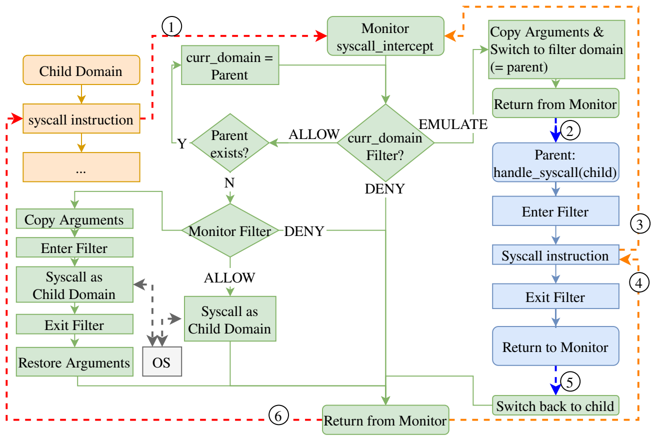

Figure 1: Overview of our syscall interception design with nested filtering. The child-domain is marked in orange, the filter domain in blue, and the monitor in green.

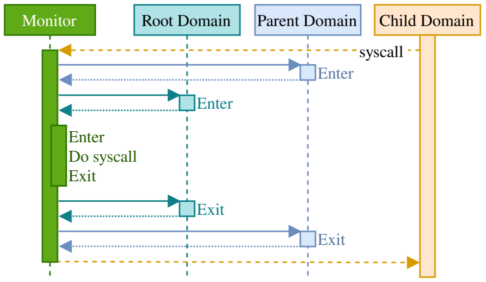

Figure 2: With nested filtering, domains can recursively filter their children with syscall enter and exit filters.

Impersonation. The parent-registered filter runs with (PKRU) permissions of this parent domain. The filter can, using its higher privileges, issue syscalls on its own, e.g., to communicate with a device the child domain does not have access to. However, it can also impersonate the child, *i.e.*, issue syscalls on behalf of its child-domain to use the lower-privilege child permissions in the kernel during the syscall. Before dispatching the syscall to the operating system, the monitor sets the PKRU to the desired value. Of course, any (impersonated) syscall within the filter is also subject to interposition.

5.2 Same-Thread Nested Filtering

Achieving nested syscall filtering is not trivial since a syscall filter needs to be interruptible by another parent syscall filter, demanding reentrant-safety for the monitor. The key ingredient for successful nesting is same-thread filtering, meaning that the filter code runs on the same thread as the filtered syscall. This limits the applicability of certain interposition techniques, as we show in the following.

seccomp-user delivers intercepted syscalls to a separate tracer thread. Since it does not provide secure means to continue or alter the original syscall, one has to run any filter code in the tracer. However, the tracer itself cannot be easily traced again, thus, prohibiting our nested-filtering approach.

seccomp-ptrace. We use seccomp-bpf in order to only trace the syscalls of interest. Furthermore, we do not execute any filter code inside the tracer but delegate the syscall back to the tracee thread similar to pku-user-delegate: The tracer emulates the `syscall` instruction by replacing the instruction pointer to trap into the monitor and storing the return address in the `RCX` register. If the tracee is already in the monitor domain (as indicated by the `PKRU` register), the syscall is allowed.

pku-user-delegate follows the same delegation approach but is simpler and faster by avoiding a separate tracer thread.

5.3 Secure Filter Design

Designing secure syscall filters is challenging [20] due to untrusted syscall arguments, which requires proper locking.

Untrusted Arguments. A well-known problem of syscall filtering is that syscall arguments might change during the filtering, creating TOCTOU issues. For example, assume that the `open("/tmp/benign")` syscall is filtered to check its path argument against an allow list. Two colluding threads could circumvent this check, as follows. While the first thread is issuing a benign `open` call, the second thread is manipulating the path argument in memory to point to a different file – right after the check has happened.

To solve this problem, our monitor copies sensitive pointer arguments used for filter decisions such as buffers or strings to a per-thread "sysargs" memory before invoking an enter-filter. The same applies to return buffers filled by the kernel, which are copied back from the sysargs memory inside the exit-filter. We protect this memory using a separate per-thread sysargs protection key to prevent attacks from colluding threads. We defer a discussion on potential optimizations to Section 7.

The monitor has full `PKRU` permissions. Hence, before copying untrusted syscall arguments, the monitor needs to check whether the filtered domain itself has permission to access this memory. This avoids so-called Boomerang attacks [35].

Locking. Syscall filters might use internal data structures, e.g., for holding a domain's open file descriptors. For multi-threaded programs, these data structures need to be locked while avoiding deadlocks: First, each data structure must only be locked and accessed by a single domain (e.g., the root domain). Second, to avoid deadlocks through recursion, the filter code itself must not perform any explicit domain switch, particularly not towards the child domain it filters.

5.4 Secure Multi-domain Call Gates

Call gates are secure entry points into a PKU domain and perform the `PKRU` switch, e.g., via the `WRPKRU` instruction. To prevent malicious code from reusing this `WRPKRU` instruction with an illegitimate value, ERIM and Hodor double-check the value after the `WRPKRU` instruction, as shown in Figure 3 left.

Unfortunately, their call gate only supports a static `PKRU` configuration (cf. `$PKRUVALUE` in Figure 3). Thus, call gates for all potential `PKRUVALUE`s need to be compiled in advance. Using dynamic `PKRU` values from memory seems infeasible at first since this memory needs to be accessible to the untrusted domains but also be protected from corruption.

We employ a generic multi-domain call gate by using PKU for its protection. As shown in Figure 3 (right,lines 3–4), the `PKRU` value is fetched from a fixed offset in the thread-local storage (TLS), relative to the `fs` register. For this, we reuse an additional page that Donky places on the TLS, protected with a key for which each domain has read-only access. Thus, our call gate can securely fetch the intended `PKRU` value again (line 6–7) and compare it with the written value (line 8). When maliciously invoking the call gate, each thread can only use its own current `PKRU` value stored inside its TLS.

Our call gate is generic enough such that we do not need to place it in every domain's entry function but only at the monitor's exit points. Our monitor switches domains by writing the desired `PKRU` value to the TLS before exiting. The monitor exit points are the only place where transitions into non-monitor domains occur. Similar to ERIM, the monitor's entry point needs no special call gate protection but simply elevates the `PKRU` value to full permissions before doing a secure context switch. Through binary scanning, one can ensure that no other unsafe `WRPKRU` or `XRSTOR` occurances, which may be used to switch to arbitrary domains, exist.

5.5 Security Argumentation

Security of Jenny involves security of the syscall interposition technique, the base filters, the Jenny design, as well as the security of concrete application filter rules.

Syscall Interposition. Our pku-user-delegate intercepts all desired syscalls of a process and its threads and child processes in the kernel and securely delivers them to the PKU-protected userspace monitor. However, pku-user-delegate is susceptible to the following attacks: First, as with other in-process sandboxes, it does not survive `execve`. Thus, all of our filter sets block the `execve` syscalls. Second, it is reconfigurable via the `ioctl` syscall. However, once initialized the pku-user-delegate can simply deny other `ioctl` commands not coming from the monitor by checking the `PKRU` register.

Our libc-indirect* method can be applied in addition to pku-user-delegate to improve performance of filtered syscalls. If a domain bypasses libc-indirect*, e.g., by using the `syscall` instruction directly, our pku-user-delegate will intervene. When

```
2:
xorl %ecx,%ecx
xorl %edx,%edx
movl $PKRUVALUE, %eax
wrpkru
cmpl $PKRUVALUE, %eax
jne 2b ; error
```

```
xor %rcx, %rcx
xor %rdx, %rdx
mov tls_offset, %rax
mov %fs:(%rax),%rax
wrpkru
mov tls_offset, %rcx
mov %fs:(%rcx),%rcx
cmp %rax, %rcx
je 1f
ud2   ; die here
1:
```

Figure 3: PKRU switches during secure call gates.

using such a kernel-based mechanism, the sandboxed code may contain arbitrary syscall instructions.

Base Filters. Our base filter rules (base-donky*, base-mpk) address all security-critical monitor resources, including memory and file system resources, process-related configurations. Of course, we cannot guarantee preventing all syscall attacks. To reduce the likelihood of attacks, we block new syscalls by default. We further used strace [58] source code to cluster syscalls into classes (e.g., memory-modifying syscalls), which we then analyzed manually.

For protecting the PKRU register on x86-64 CPUs, our base-mpk additionally employ binary scanning and W⊕X [11] as well as secure multi-domain call gates, as detailed below.

Call gates Any WRPKRU occurrence within our trusted library are either monitor-enter call gates or use our secure multi-domain call gate for monitor exits. An interesting corner occurs due to our syscall impersonation method, for which the monitor temporarily drops PKRU privileges for impersonating the syscall and elevating privileges again afterwards. Since this PKRU elevation presents a monitor-enter call gate, we also need to restore the monitor's register state, including the monitor stack pointer. We do so by storing the monitor stack pointer on a protected TLS variable during impersonation.

Our multi-domain call gate cannot be manipulated since the PKRU value is stored on protected TLS memory, and the relevant fs register is read-only on our Linux 5.4 setup. Newer Linux kernels (since version 5.9) can optionally enable the userspace to write the fs register. If enabled, one could use an alternative to the fs discussed in [27].

Jenny ensures that domains cannot evade syscall filtering, as follows: First, syscall filters installed by domain A are enforced on *all* of its child domains, also those created later on. Second, the monitor ensures that all registered syscall filters are executed in a hierarchical order until one filter denies the syscall. Third, domains cannot be deleted when they have child domains. Fourth, when a child domain B orphans its child domain C using the Donky monitor API call dk_domain_release_child (cf. [56]), domain C is assigned A as new parent. This ensures that filters of A cannot be evaded. Fifth, monitor filters are globally applied to all domains.

Application Filters need to sanitize syscall arguments before accessing them [20]. Jenny copies syscall arguments to a PKU-protected thread-local memory that is only accessible to the filtering code. When writing syscall filters, developers need to further follow a few security principles: (i) protect the syscall filter rules and data structures by means of PKU, (ii) lock internal data appropriately and avoid domain calls (cf. Section 5.3) (iii) identify threads and domains appropriately.

Our *localstorage* filter rules prevent untrusted domains from escaping a path sandbox, as follows: Any path-related syscalls are intercepted, and paths are resolved, e.g., to neutralize symbolic links. Before the respective syscall is allowed, the path is prepended with a per-domain sandbox path. Finally, any file descriptor-based syscalls are intercepted such that a domain can only access file descriptors it has opened itself. However, these filters, currently, do not protect against non-sandboxed colluding applications running with the same user privileges since they can also manipulate this directory.

5.6 Secure Signals

Signals expose an inherent design weakness of the Linux kernel w.r.t. PKU sandboxes [11]. To date, none of the existing PKU sandboxes provide secure signal handling.

While Donky [56] describes how secure signal handling could be supported, it misses the above design weakness. Furthermore, it neither provides a semantic to PKU-aware signal handling nor a prototype implementation. Hodor simply disables signal delivery in the kernel while inside an isolated domain [27]. ERIM does not discuss secure signal handling [61].

In the following, we introduce a PKU-secure signal API, discuss the protection of signal stacks, and show how signal handling is made secure against race conditions.

Secure Signal API. Designing a meaningful signal handling API for use with PKU sandboxing is non-obvious. If multiple domains attempt to register a signal handler for the same signal, who should get the signal delivered? In case of synchronous signals one might deliver to the domain triggering the signal. However, this could be abused by certain attacks relying on the suppression of segmentation faults [34]. Even worse, for asynchronous signals (e.g., timers), it is ambiguous which domain triggered the signal or is eligible to take it.

We follow a secure-by-default philosophy: Our monitor allows one signal to be handled by exactly one domain. Moreover, a parent domain can override a signal handler registered by one of its children and thus, virtualize a child's signal handlers. The parent can then decide whether to manually redirect the signal to the child domain or not.

To realize our API we register our own monitor-protected signal handler as a single entry point for all signals. For this, we filter the syscalls (rt_)sigaction and signal and store domain-provided signal handler information in the monitor. On signal delivery, our (monitor) handler looks up the corresponding domain signal handler and executes it. Domains can use standard signal registration functions without any code modifications. Jenny intercepts these functions and aborts with an error in case the signal was already registered.

Secure Signal Stack. Protection of the so-called signal frame is of key importance, as an attacker manipulating it (e.g., the stored `PKRU` register) could elevate privileges [11]. The Linux kernel stores the signal frame holding all CPU registers on the stack, from which it will be restored upon signal return via the `(rt_) sigreturn` syscall. Unfortunately, one cannot protect the signal frame by means of PKU, since the Linux kernel deprivileges the signal handler to protection key zero.

To solve this, we need to ensure that signal frames are always pushed onto a protected signal stack. Thus, we allocate a separate alternative signal stack per thread, protect it with the monitor's protection key, and register it in the kernel via the `sigaltstack` syscall. We propose to extend the `sigaltstack` syscall by adding an associated protection key to the existing `ss_flags` field. The kernel then registers this protection key alongside the signal stack. Whenever a signal arrives, the kernel first loads the registered protection key before storing the signal frame on the protected stack and invoking our handler. To guarantee that our protected alternative signal stacks are always used, we augment the above hooked signal registration functions to always inject the `SA_ONSTACK` flag. Additionally we filter the syscalls `sigaltstack` and `(rt_) sigreturn`. Thus, signal delivery and return are protected. In total, our patch adds 33 LoC to the kernel.

Secure Signal Handler. Race conditions present another challenge. E.g., signals may occur while other signals are still handled, which corrupts our signal stack. Also, asynchronous signals (e.g., `SIGALRM`) can occur during critical monitor execution, e.g., while a lock is held, leading to deadlock situations. Also, the signal frame might contain sensitive register values that must not be exposed to a domain's signal handler.

We defeat race conditions, as follows. First, to prevent signals from interrupting each other, we temporarily block all signals via the `sa_mask` field in the `(rt_) sigaction` filter. This is not a limitation in practice, since the blocked signals will arrive when the current one is finished.

Second, to prevent signals from interfering with current monitor execution, we defer them to the monitor exit point. That is, our universal signal handler puts the signal on a per-thread "monitor pending" variable together with the original signal mask and blocks all signals for this thread. We augment all monitor exit paths to check for deferred signals, re-raising them using `tgkill` and unblocking them using `sigprocmask`. Thus, the monitor now gets interrupted at a precise location in the exit path that does not make use of monitor data anymore.

To avoid information leakage, we keep the signal frame only on our protected signal stack and do not serve it to the domain signal handler. This prevents the domain from extracting or manipulating register values in case the signal was raised while in monitor mode. This is not a severe limitation, as commonly, "the handler function doesn't make any use of the third argument", *i.e.*, the signal frame [3]. Alternatively, one could provide a sanitized frame to preserve compatibility.

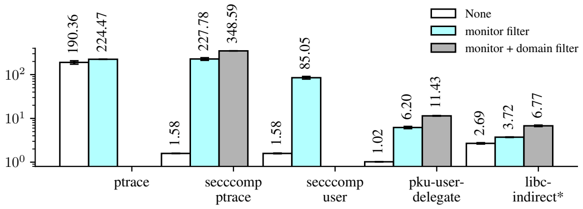

Figure 4: Relative execution time of a `getpid` syscall.

6 Evaluation

The performance of Jenny is influenced by the syscall interception mechanism, which and how many syscalls are made, and the filtering logic. Our micro-benchmarks measure the overhead of individual syscalls. Macro-benchmarks measure the impact on whole applications and libraries.

Filtering Mechanisms. We evaluate our seccomp-ptrace, pku-user-delegate and libc-indirect* methods on all benchmarks and ptrace only for micro-benchmarks to compare against the fasterseccomp-ptrace. As discussed in Section 5.2, some filtering mechanisms have functional limitations. E.g., Seccomp-BPF and seccomp-trap are incompatible with PKU sandboxing and, thus, excluded. We further exclude seccomp-user from application benchmarks due to its limitations in multi-threading and nested filtering. Finally, we omit Syscall User Dispatch, as it does not support safe signal handling, however, pku-user-delegate serves as a lower bound.

Setup. We use Linux 5.4.0 on an Intel Xeon 4208 CPU with a fixed frequency of 1.8 GHz. We measure the performance using the instruction sequence `lfence,rdtsc,lfence`. All file-based operations run within a `tmpfs`. The standard deviation is included in all figures. We removed outliers, *i.e.*, measurements deviating by twice the median value.

For benchmarking unmodified applications, we compile Jenny as a shared library and use `LD_PRELOAD` to load it. We use constructors to initialize Jenny, create a new domain for sandboxing the application, install the necessary filters, and start timing measurements, which are then evaluated by a destructor. We evaluate the startup overhead separately in Appendix A. For nginx/apachebench and lmbench, we use the performance numbers given by these tools instead.

6.1 Micro-benchmarking

For each mechanism and rule set, including no filters as denoted by "none", we time the execution of 100 syscalls and repeat the measurement 100x. The plots show the relative runtime compared to when no mechanisms are in use.

Figure 4 shows the results for a `getpid` syscall, one of fastest syscalls showing the upper-bound overhead for each

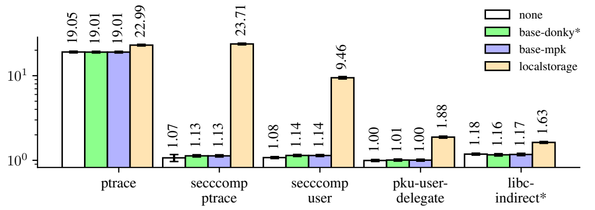

Figure 5: Relative execution time of an open syscall.

syscall interposition technique. We install an empty filter that simply allows the syscall– once as a "monitor filter" and once both in the monitor and in the domain. The difference between "monitor" and "monitor + domain" is the constant domain-switch overhead of each additional nesting level, e.g., when a (parent) domain installs a custom filter for its children.

Our libc-indirect* mechanism takes 2.69 x the time of `getpid` for *each unfiltered* syscall, since the monitor is always invoked. For *filtered* syscalls, it is the fastest mechanism since it remains in userspace. On the other hand, pku-user-delegate filters a bit slower due to the kernel context switches but has a negligible (2%) overhead for unfiltered syscalls. Seccomp decisions run with 1.58 x the time in the kernel.

For Figure 5 we enable *base-mpk* and *base-donky** protection as well as our *localstorage* filter to isolate domains on the file system (Section 3.3). We benchmark the most affected `open` syscall. Note that we populate the localstorage directory with the required files before our benchmarks. As expected, apart from ptrace, only the *localstorage* filter set sees a significant slowdown. Contrary to prior claims [11], our filter rules *base-donky** and *base-mpk* do not need to trace the `open` syscall. Thus, *our overhead for protecting a PKU sandbox is negligible.*

As expected, the filtering overhead decreases for the slower `open` syscall: For `getpid`, we observe a 58% overhead for seccomp-bpf, compared to 8% for `open`. In contrast, pku-user-delegate has no measurable overhead on unfiltered syscalls.

Syscall Compatibility and Signals. To show compatibility with the syscall interface and the performance overhead of our secure signal handling (Section 5.6) we used the standard Linux micro-benchmarking tool "lmbench" [37]. Here, we enable the *localstorage* filter set in addition to *base-mpk*. lmbench internally runs its benchmark multiple times and reports a single number. In addition, we run each lmbench binary 10x. We directly used the reported latency numbers unless they are given as a bandwidth unit (e.g., MB/s), in which case we invert them. Figure 7 shows the relative performance slowdown. Most notably, `sig catch` reports a relative execution time for sending and catching a signal of 15.42x, 1.22x, and 2.76x for seccomp-ptrace, pku-user-delegate, and libc-indirect*, respectively. This benchmark also includes sending the signal (via the `kill` syscall), which reports much higher numbers for libc-indirect* intercepting `kill`. All lmbench programs are compatible without modifications, except `lat_rpc`, which did not start on our system even without Jenny.

6.2 Application Benchmarks

We benchmark Jenny with a range of unmodified applications to show its compatibility with syscalls, multi-threading, and signals as well as the slowdown of the different filter rules.

Figure 6 shows the relative runtime overhead when applications are sandboxed using Jenny. To evaluate our different rule sets, we use a mix of file-intensive and compute-intensive applications and run each one 10 times. We use `dd` to write a zero-filled 1 MB file using 1024 separate `read` and `write` syscalls. For `git status`, `ls`, and `zip`, we inspect or compress a clone of `https://github.com/git/git.git` at the branch `v2.30.0-rc0` with a clone depth of 10 000 For openssl, we create an ECDSA signature for a 1KiB file using a `secp521r1` key. For ffmpeg, we re-encode a 2 s 720p H.264 file using libx264 with the `-threads 3` option, which spawns nine threads, three of which are actual encoding threads.

We observe that compute-intensive applications (*i.e.*, zip, openssl, and ffmpeg) have a negligible overhead for the faster filtering mechanisms (*i.e.*, pku-user-delegate and libc-indirect*), irrespective of the complexity of the filter-rules. This is expected since the runtime of these applications is dominated by computation and not by syscalls. However, due to synchronization (`futex`), ffmpeg still shows a slowdown for seccomp-ptrace. Applications with excessive file-based syscalls (*i.e.*, dd, git, ls) perform considerably worse when isolated in a *localstorage* directory, since any file-descriptor- and path-based syscalls are intercepted. E.g., the syscall-intense git reaches 295% overhead for pku-user-delegate, while base-mpk only shows a 0–24% overhead.

6.3 Case Study: Webserver

We evaluate a real-world use-case, namely sandboxing libraries and modules in a webserver. This shows library isolation and nested domains with our syscall filtering framework. We use the nginx web server and configure it to use eight worker threads and optionally the gzip module as well as the HTTP Basic Authentication module, denoted as "auth". As seen in Figure 8, the nginx core runs in the root domain, for which we enable the *base-mpk* filter. Two child domains simulate a potential attacker misusing the syscall interface. For the gzip module, we isolate the underlying zlib library in a separate child domain and install filters to prevent any syscalls. The "auth" module runs in another child do, for which we enable the *localstorage* filters to constrain it into its local directory, having only access to its authentication file.

We benchmark the unmodified nginx against Jenny, using apachebench with 10 000 requests in total and 10 concurrent requests. We request 0 KiB files to show the worst-case overhead. Figure 9 shows the evaluation results. If no modules are

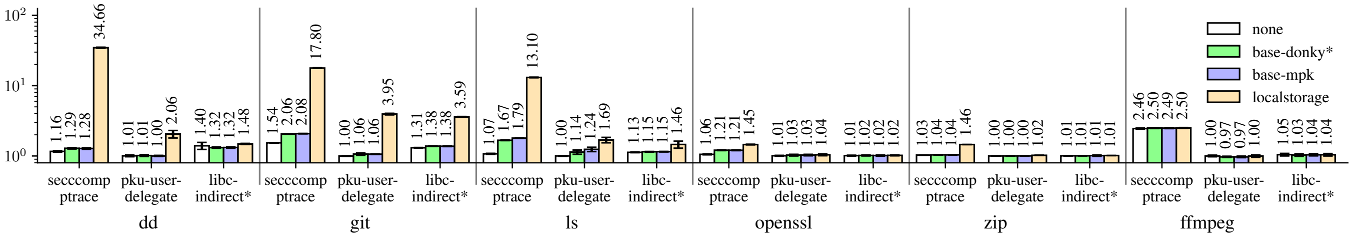

Figure 6: Runtime of different applications when Jenny is preloaded, relative to their native execution.

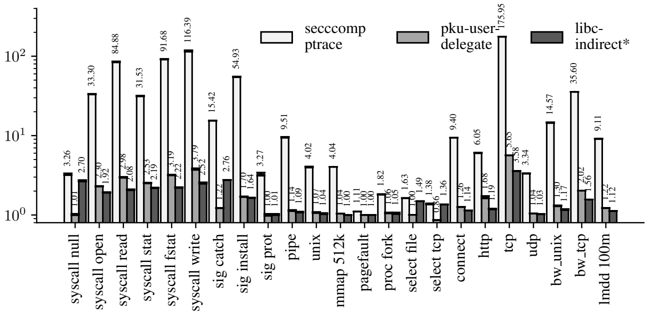

Figure 7: Relative runtime of individual lmbench programs when isolated with *localstorage* filters.

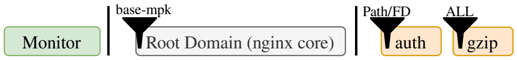

Figure 8: Overview of the separate modules constrained within their domains and their respective filters.

active, all worker threads run in the same domain, incurring no domain-switch overhead. Also, our *base-mpk* base filters do not negatively affect the performance. For "gzip", there are two domain switches per request for entering and leaving the child domain performing zlib. Since zlib does not require syscalls, the overhead is negligible as well. For "gzip + auth", there are two additional domain switches for the "auth" domain plus file-operation syscalls for reading the file, which is isolated in the *localstorage*. Here, we still only measure a 5% overhead for our pku-user-delegate. In total, 14 protection keys are used: two for the monitor, three for the domains, and nine for the threads, including the main thread. For this case-study, we added 167 and removed 36 LoC from nginx 1.20.0.

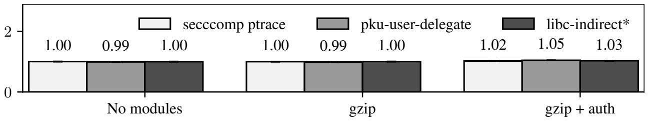

Figure 9: Relative runtime of a HTTP request in nginx with different (isolated) modules active, normalized against the unmodified nginx.

7 Discussion and Future Work

Compatibility. Since our base filters deny a set of dangerous syscalls, applications relying on them are incompatible with our sandbox. Furthermore, our optional localstorage filters constrain a domain into its isolated directory, which can cause compatibility issues if the sandboxed application needs to access files outside. For our evaluation we add any files needed to this directory. Protected file descriptors can also cause problems if they are not assigned to the sandbox domain.

As demonstrated in Section 6.2, syscalls and signals are transparently intercepted by the monitor and redicted to the respective handlers. Thus, Jenny works without having to modify the sandboxed code. However, Jenny scans the binary for unsafe `WRPKRU` (and `XRSTOR`) instructions. Hence some binaries are not compatible. In case the unsafe instructions only occur in a *misaligned* manner, binary rewriting [61] or breakpointinng [27] could be used to improve compatibility.

Currently, our prototype kernel module only supports filtering a single process, including its threads and forked child processes. Support for arbitrary processes could be easily added.

Kernel Interface. Linux's syscall interface is steadily growing, both in size and complexity. Our analysis was based on the documentation of Linux 5.4.0. Since future syscalls could further exploit PKU systems, we suggest to only allow a small set of syscalls that are needed and deny unknown syscalls.

Number of Protection Keys. Intel MPK only supports 16 keys, of which Linux reserves up to two keys. Jenny currently uses one private key per domain and one extra key for read-only accessible global monitor data. Furthermore, our solution for protecting filtered syscall arguments uses one sysargs protection key for each thread to conservatively prevent TOCTOU and Boomerang attacks [35]. Thus we limited our nginx and ffmpeg use-cases to eight worker threads.

There are multiple ways to relax this limitation. Sysargs protection keys are not required for threads that run in isolation, *i.e.*, only stay in one domain. There, we can copy syscall arguments to the filteree-private memory instead. Moreover, sysargs keys can be omitted when the syscall filters affecting a thread do not access pointer arguments. One could, for example, allocate sysargs keys on-demand when a thread enters a domain that requires such complex filters the first time.

Another way would be to pre-allocate a small pool of sysargs protection keys that are dynamically dispatched by the monitor only when needed. However, this limits the number of concurrently filtered syscalls and they could stall when the pool is exhausted.

Alternatively, for large applications (e.g., web browsers) one can virtualize protection keys at a loss of security (e.g., only giving probabilistic isolation guarantees). The Donky software already provides infrastructure for specifying whether an allocated protection key can be virtualized (e.g., reused multiple times). Thus, software architects can unlock certain keys for virtualization, for which the security risk is manageable.

8 Related Work

Application Sandboxing. Traditionally, syscall interception was used for whole-application sandboxing. In the past, a number of different solutions have been presented to filter or monitor syscalls [4, 8, 21, 24, 25, 29, 30, 53–55]. They use various of mechanisms for enforcement, ranging from binary rewriting [8, 54, 55] to new kernel features [21, 29]. Garfinkel et al. [20] presented best practices and common pitfalls for such systems, and Parampalli et al. [52] presented powerful attacks against them. Our implementation closely follows these best practices to minimize potential vulnerabilities.

Linux provides AppArmor, SELinux, and seccomp-bpf for confining syscalls for applications. However, they only support simple static filter rules that cannot be changed while the application runs. seccomp-bpf only allows installing additional rules to further constrain itself. To restrict other kernel resources, Linux also provides other mechanisms like "control groups" and "namespaces". However, they are unsuitable for PKU-based in-process isolation. Other work [5, 18, 36] used virtualization techniques to emulate the entire syscall interface and sandbox applications.

Generating optimal rules, such that only required syscalls are possible, is orthogonal work. By statically or dynamically analyzing applications, existing work [7, 8, 15, 19, 22, 23, 62] generates such rules, which can also be applied to Jenny.

In-Process Isolation. To sandbox code *within* an application also requires filtering or blocking syscalls. For NativeClient, WebAssembly, or other interpreted languages, this is typically enforced through (re)compilation. The compiler makes sure not to emit any unsafe instructions (e.g., `syscall`). The isolated code uses predefined functions to communicate with the rest of the application, which can then issue syscalls [1, 59].

PKU-based Systems. Recently, a number of PKU-based sandboxes have been proposed [27, 56, 61]. Such systems require syscall filtering to prevent `WRPKRU` exploitation and to enforce their sandbox. Connor et al. [11] showed vulnerabilities for such systems and how to overcome them. However, none of them offer a complete solution, and some claim that the overhead of proper syscall handling would be prohibitive. Existing work fails to explore different syscall filtering mechanisms and ways to optimize the necessary filters.

9 Conclusion

In this work, we addressed various syscall filtering challenges for PKU-based memory isolation systems leading to Jenny, the first PKU system with comprehensive syscall support. We uncovered previously unknown PKU-related syscall attacks, compared various syscall interception mechanisms and designed a faster mechanism that fits the needs of PKU systems. We designed comprehensive and efficient filter rules for protecting a PKU sandbox. We further designed filters for confining sandboxes in local directory, similar to chroot. Many other filter sets are conceivable with Jenny, ranging from file system virtualization, in-process namespaces and browser site isolation towards isolation of cloud workloads [10].

Jenny provides filtering on the same thread, thus enabling and simplifying impersonation of syscalls, as well as nested filtering and signal handling. In conclusion, we showed that syscall filtering for PKU systems is both practical and secure, and we achieved a minor performance impact of 5% for nginx.

Acknowledgments

We thank the anonymous reviewers and especially our Shepherd, Nathan Burow, for their valuable suggestions and comments, that substantially helped in improving the paper. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). This work has also been supported by the Austrian Research Promotion Agency (FFG) via the competence center Know-Center (grant number 844595), which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, and Styria and via the K-project DeSSnet, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia.

References

[1] Anatomy of a Syscall, Chromium. https://www.chromium.org/nativeclient/reference/anatomy-of-a-sys, 2020.

[2] *capabilities(7) — Linux manual page*, December 2020.

[3] *sigaction(2) — Linux manual page*, December 2020.

[4] Anurag Acharya and Mandar Raje. MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications. In *USENIX Security Symposium*, 2000.

[5] Adam Belay, Andrea Bittau, Ali José Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *OSDI*, pages 335–348, 2012.

[6] Christian Brauner. Seccomp Notify. https://brauner.github.io/2020/07/23/seccomp-notify.html, 2020.

[7] Alexander Bulekov, Rasoul Jahanshahi, and Manuel Egele. Saphire: Sandboxing php applications with tailored system call allowlists.

[8] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. Automating Seccomp Filter Generation for Linux Applications. *CoRR*, abs/2012.02554, 2020.

[9] Chromium. Linux Sandboxing. https://chromium.googlesource.com/chromium/src/+/master/docs/linux/sandboxing.md, 2020.

[10] Cloudflare. Introducing cloudflare workers: Run javascript service workers at the edge. https://blog.cloudflare.com/introducing-cloudflare-workers/, 2017.

[11] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1409–1426, 2020.

[12] Jonathan Corbet. Reconsidering unprivileged BPF. https://lwn.net/Articles/796328/, 2019.

[13] Alexander Culafi. Apache Struts vulnerabilities allow remote code execution, DoS. https://searchsecurity.techtarget.com/news/252487813/Apache-Struts-vulnerabilities-allow-remote-code-execution-DoS, 2020.

[14] Leila Delshadtehrani, Sadullah Canakci, Manuel Egele, and Ajay Joshi. Efficient Sealable Protection Keys for RISC-V. *CoRR*, abs/2012.02715, 2020.

[15] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P Kemerlis. Sysfilter: Automated system call filtering for commodity software. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*, pages 459–474, 2020.

[16] Will Drewry. [RFC,PATCH 2/2] Documentation: prctl/seccomp_filter. https://lwn.net/Articles/475049/, 2012.

[17] Jake Edge. Seccomp and deep argument inspection. https://lwn.net/Articles/822256/, 2020.

[18] Bryan Ford and Russ Cox. Vx32: Lightweight User-level Sandboxing on the x86. In *USENIX ATC*, pages 293–306, 2008.

[19] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A Sense of Self for Unix Processes. In *S&P*, pages 120–128, 1996.

[20] Tal Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *NDSS*, 2003.

[21] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *NDSS*, 2004.

[22] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. Confine: Automated system call policy generation for container attack surface reduction. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, 2020.

[23] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Temporal system call specialization for attack surface reduction. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[24] Douglas P. Ghormley, Steven H. Rodrigues, David Petrou, and Thomas E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *USENIX ATC*, 1998.

[25] Ian Goldberg, David A. Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications. In *USENIX Security Symposium*, 1996.

[26] Google Developers. Sandboxed API. https://developers.google.com/sandboxed-api/, 2020.

[27] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *USENIX ATC*, pages 489–504, 2019.

[28] Docker Inc. Docker security. https://www.docker.com/, 2020.

[29] K. Jain and R. Sekar. User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement. In *NDSS*, 2000.

[30] Taesoo Kim and Nickolai Zeldovich. Practical and Effective Sandboxing for Non-root Users. In *USENIX ATC*, pages 139–144, 2013.

[31] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *EUROSYS*, pages 437–452, 2017.

[32] Linux kernel. Memory Protection Keys. https://www.kernel.org/doc/Documentation/x86/protection-keys.txt, 2017.

[33] Linux kernel. SECure COMPuting with filters. https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt, 2017.

[34] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*, pages 973–990, 2018.

[35] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments. In *NDSS*, 2017.

[36] Eyal Manor. Bringing the best of serverless to you. https://cloudplatform.googleblog.com/2018/07/bringing-the-best-of-serverless-to-you.html, 2018.

[37] Larry W. McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *USENIX ATC*, pages 279–294, 1996.

[38] Mozilla. Security/Sandbox/Seccomp. https://wiki.mozilla.org/Security/Sandbox/Seccomp, 2020.

[39] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 699–716, 2020.

[40] National Vulnerability Database NIST. CVE-2002-0059. https://nvd.nist.gov/vuln/detail/CVE-2002-0059, 2019.

[41] National Vulnerability Database NIST. CVE-2003-0107. https://nvd.nist.gov/vuln/detail/CVE-2003-0107, 2019.

[42] National Vulnerability Database NIST. CVE-2005-2096. https://nvd.nist.gov/vuln/detail/CVE-2005-2096, 2019.

[43] National Vulnerability Database NIST. CVE-2020-8835. https://nvd.nist.gov/vuln/detail/CVE-2020-8835, 2019.

[44] National Vulnerability Database NIST. CVE-2021-20194. https://nvd.nist.gov/vuln/detail/CVE-2021-20194, 2019.

[45] National Vulnerability Database NIST. CVE-2021-23133. https://nvd.nist.gov/vuln/detail/CVE-2021-23133, 2019.

[46] National Vulnerability Database NIST. CVE-2021-29154. https://nvd.nist.gov/vuln/detail/CVE-2021-29154, 2019.

[47] National Vulnerability Database NIST. CVE-2021-3156. https://nvd.nist.gov/vuln/detail/CVE-2021-3156, 2019.

[48] National Vulnerability Database NIST. CVE-2021-33200. https://nvd.nist.gov/vuln/detail/CVE-2021-33200, 2019.

[49] National Vulnerability Database NIST. CVE-2021-3444. https://nvd.nist.gov/vuln/detail/CVE-2021-3444, 2019.

[50] Charlie Osborne. Remote code execution vulnerability exposed in popular JavaScript serialization package. https://portswigger.net/daily-swig/remote-code-execution-vulnerability-exposed-in-popular-javascript-serialization-package, 2020.

[51] Pierluigi Paganini. Google found zero-click vulnerabilities in Apple's multimedia processing components. https://securityaffairs.co/wordpress/102459/hacking/apple-zero-click-vulnerabilities.html, 2020.

[52] Chetan Parampalli, R. Sekar, and Rob Johnson. A practical mimicry attack against powerful system-call monitors. In *AsiaCCS*, pages 156–167, 2008.

[53] Niels Provos. Improving Host Security with System Call Policies. In *USENIX Security Symposium*, 2003.

[54] Mohan Rajagopalan, Matti A. Hiltunen, Trevor Jim, and Richard D. Schlichting. Authenticated System Calls. In *DSN*, pages 358–367, 2005.

[55] Mohan Rajagopalan, Matti A. Hiltunen, Trevor Jim, and Richard D. Schlichting. System Call Monitoring Using Authenticated System Calls. *IEEE Trans. Dependable Secur. Comput.*, 3:216–229, 2006.

[56] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain Keys - Efficient In-Process Isolation for RISC-V and x86. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1677–1694, 2020.

[57] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *USENIX Security Symposium*, pages 1–12, 2010.

[58] strace Developers. strace – linux syscall tracer. https://strace.io/, 2021.

[59] WebAssembly System Interface Subgroup. The WebAssembly System Interface. https://wasi.dev/, 2020.

[60] The kernel development community. Syscall User Dispatch. https://www.kernel.org/doc/html/latest/admin-guide/syscall-user-dispatch.html.

[61] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *USENIX Security Symposium*, pages 1221–1238, 2019.

[62] David A. Wagner and Drew Dean. Intrusion Detection via Static Analysis. In *S&P*, pages 156–168, 2001.

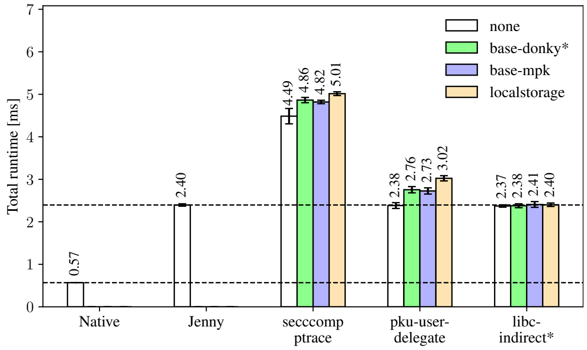

Figure 10: Runtime of `/bin/true` with and without Jenny for various interception mechanisms and filter rules.

A Evaluation of the Initialization Overhead

Our evaluation in Section 6.2 does not include the additional time it takes to load and initialize Jenny. Jenny is built as a shared library, which we preload using `LD_PRELOAD`. To measure the constant overhead of preloading and initialization (*i.e.*, the creation of a default domain, scanning the binary), we run the program `/bin/true` without filter rules. Our measurement setup benchmarks the time between the `execve` syscall of `/bin/true` and the continuation of the `waitpid` syscall of a parent program.

Figure 10 shows the total runtime of `/bin/true` under different conditions. The first two bars show the native runtime and the runtime when the application is isolated in a domain, but with no syscall filtering. One can see that Jenny takes 1.8 ms to initialize. While outside of scope for our work, we believe that further optimizations of the startup procedure are possible here.

Other bars show the runtime when Jenny is used, with different filter mechanisms and filter rules applied. libc-indirect* traces all libc syscalls without distinction – hence the fast initialization. The minor overhead of libc-indirect* stems from registering the filter rules in the monitor and also applies to all other mechanisms. In contrast, other mechanisms take longer to initialize when more syscalls need to be registered for interception. seccomp-based filtering shows a larger overhead of up to 2.6 ms.In contrast, our pku-user-delegate mechanism shows small overheads of 0.6 ms.

B Sandboxing privileged processes

PKU sandboxing might also be used to protect *privileged programs*, e.g., setuid binaries (cf. the sudo exploit [47]). In this case, care must be taken to shield *all* privileged system resources accordingly. For this, one should additionally block all syscalls that require capabilities (e.g., `CAP_SYS_ADMIN`) [2] such as `pivot_root`, `mount`, and `reboot`. Moreover, privileged file system resources (e.g., `/proc/sys`, `/sys` and `/dev`) need to be blocked. Here, using the `prctl` approach from *base-mpk* is ineffective, instead new rules need to be designed similar to *localstorage*.

C Filter API

In Jenny the filtering mechanism and base filter rules (cf. Section 3.3) can be supplied via environment variables (e.g., `FILTER=base-mpk MECHANISM=ptrace_seccomp`) or via an API call. Additionally, as shown in Figure 11, one can install custom filter rules for each (sub-)domain. Note, that these filters run in addition to the base filter rules as well as any registered filter rules of their own parent domains. Line 9 registers a simple rule to deny all `write` syscalls. Line 10 registers a filter function, which logs (line 3) all attempted file accesses via the `open` syscall and then denies them.

```
 1 void custom_open_filter(trace_info_t *ti) {
 2     if (IS_SYSCALL_ENTER(ti)) {
 3         printf("Domain %d is about to open file: %s\n",
 4                ti->did, ti->args[0]);
 5         SYSFILTER_RETURN(ti, -EACCES);
 6     }
 7 }
 8 
 9 //register syscall filters for sub-domain
10 jenny_sysfilter_domain(domain_id, SYS_write, SYSCALL_DENIED);
11 jenny_sysfilter_domain(domain_id, SYS_open, custom_open_filter);
```

Figure 11: Usage example of our filter API