Richard Sadek, BSc

User-space Syscall Filtering for In-process Isolation based on Memory Protection Keys

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisors

Dipl.-Ing. David Schrammel, BSc

Dipl.-Ing. Dr.techn. Samuel Weiser, BSc

Assessor

Univ.-Prof. Dipl.-Ing. Dr.techn. Stefan Mangard

Institute of Applied Information Processing and Communications

Graz, March 2021

Acknowledgements

Thanks to everyone who made this thesis possible.

Special thanks to my advisors David Schrammel and Samuel Weiser for their excellent mentoring of this thesis and help with their *Donky* framework, Michael Schwarz for writing the kernel module, IAIK for inspiring me to pursue my journey in information security, and my friends and family for always being there for me.

Abstract

Complex software systems often contain exploitable bugs. In a modern operating system, process isolation is used as a last defense against software bugs so that a faulty application cannot affect other applications running on the same system. Modern applications usually consist of multiple mutually distrusting components and libraries that could benefit from isolation. Often, splitting up an application into multiple processes is not feasible because of the substantial performance overhead of every context switch. Recent attempts in isolating components in web browsers using process isolation also have shown that there is a significant engineering effort involved.

Thus, efficient in-process isolation with fast switches between isolation domains is in great demand. Existing in-process isolation schemes primarily concentrate on *memory isolation* but usually do not provide sufficient isolation *regarding the kernel interface.* In the worst case, domains have access to resources that could break the memory isolation. Depending on the use-case, other kernel resources (e.g., file descriptor or the file system) need to be isolated too.

In this thesis, we employ sophisticated user-space syscall filtering for in-process isolation based on Protection Keys for User-space (PKU). Additional to the memory protection from PKU, we restrict access to kernel resources on a per-domain basis. Unlike other user-space syscall filtering systems, we do not need an additional process to filter syscalls. We use PKU to provide us with an isolated memory region for the syscall filtering code. We propose a novel way of *syscall delegation* that filters syscalls by redirecting them into a trusted reference monitor in the same application. Our flexible architecture allows so-called *nested-filtering*, where domains can register custom filtering code to constrain their child domains.

We discuss two main use-cases: syscall filters for self-protection of an existing PKU system and syscall filters to implement a lightweight local storage sandbox for isolated domains. We evaluate the performance and security of both novel and adapted existing mechanisms for syscall interception. By delegating the filtering to user-space, even the fastest syscalls (e.g., `getpid`) only have an overhead of 2x. The overhead for the `open` syscall in our local storage sandbox is just 57%–84%. When sandboxing various applications with our novel mechanisms, we measure only 0–20% overhead, whereas existing mechanisms have 5–71% overhead (depending on the syscall frequency).

Our results look promising, proving that PKU sandboxing can be both *fast* and *secure.*

Keywords:. in-process isolation, memory protection keys, user-space syscall filtering

Kurzfassung

Komplexe Softwaresysteme enthalten oft für Angreifer ausnutzbare Fehler. Betriebssysteme nutzen Prozessisolation, sodass Fehler in einer Anwendung andere nicht negativ beeinflussen können. Moderne Anwendungen bestehen oft aus mehreren Programmbibliotheken, die sich gegenseitig nicht vertrauen und daher von einer weiteren Isolierung profitieren können. Oft ist es nicht praktikabel, solche Anwendungen in mehrere Prozesse aufzuteilen, da jeder Kontextwechsel mit einem großen Overhead verbunden ist. Aktuelle Entwicklungen, einzelne Komponenten in Webbrowsern zu isolieren, zeigten, dass das Aufteilen in mehrere Prozesse auch eine große technische Herausforderung darstellt.

Folglich gibt es eine große Nachfrage nach In-Prozess-Isolation, mit der schnell zwischen Isolationsdomänen gewechselt werden kann. Bestehende In-Prozess-Isolationssysteme konzentrieren sich vor allem auf Isolierung des *Speichers*, aber bieten keine ausreichende Isolation der *Schnittstelle zum Kernel*. Im schlechtesten Fall haben Domänen Zugriff zu Ressourcen, die die Speicher-Isolation brechen können. Abhängig vom Anwendungsfall müssen auch andere Kernelressourcen isoliert werden (z.B. Dateideskriptoren).

In dieser Arbeit werden User-Space-Systemaufrufsfilter für In-Prozess-Isolation eingesetzt. Diese basieren auf Protection Keys für User-Space (PKU). So kann zusätzlich zu der Speicherabsicherung von PKU der Zugriff auf Kernelressourcen pro Domäne beschränkt werden. Im Unterschied zu anderen User-Space-Systemaufrufsfiltersystemen, braucht dieses System keinen zusätzlichen Prozess zum Filtern der Systemaufrufe. Es wird PKU verwendet, um die Systemaufrufsfilter in einer isolierten Speicherregion ausführen zu können. Außerdem werden in dieser Arbeit eine neue Art von *Systemaufruf-Delegation* sowie *verschachtelte Systemaufrufsfilter* vorgeschlagen. So können Domänen benutzerdefinierte Filter registrieren, um ihre Subdomänen einzuschränken.

Es werden zwei Hauptanwendungsfälle besprochen: Systemaufrufsfilter um ein existierendes PKU System vor bösartigen Domänen zu schützen und Systemaufrufsfilter für eine Local-Storage-Sandbox für isolierte Domänen. Die Leistungsfähigkeit und die Sicherheit für alle neuartigen und adaptieren Systemsaufrufsfiltermechanismen wird ermittelt. Indem das Filtern in den User-Space delegiert wird, ist der Overhead sogar für die schnellsten Systemaufrufe (z.B. `getpid`) nur 2x. Der Overhead des `open` Systemaufrufs in der Local-Storage-Sandbox ist nur 57%–84%. Für Anwendungen, die in der Sandbox ausgeführt werden, wird für die neuartigen Mechanismen ein Overhead von 0–20%, für existierende Mechanismen ein Overhead von 5–71% verzeichnet (abhängig von der Frequenz der Systemaufrufe).

Diese Arbeit zeigt, dass Isolation mit PKU gleichzeitig *schnell* und *sicher* sein kann.

Schlagwörter:. In-Prozess-Isolation, Memory Protection Keys, Systemaufrufsfilterung

Contents

List of Figures

Chapter 1

Introduction

In a modern operating system, the main form of memory isolation is process isolation. Each process has its own address space that maps virtual addresses to physical ones. By default, memory is private to a process. Nevertheless, a process can also decide to share parts of its address space with other applications.

For modern applications with complex trust relationships between software components and libraries, process isolation is not enough. For example, a browser runs code that renders websites and executes JavaScript. Neither the render engine nor the JavaScript code needs to interact directly with passwords stored in memory. In the past, multiple vulnerabilities in JavaScript engines were found that allowed JavaScript code to escape the sandbox and read or write arbitrary memory [Cho16]. Web servers can also benefit from additional memory isolation. A web server handles requests, but at the same time, a secret cryptographic key is loaded into memory to encrypt the traffic. In the past, vulnerabilities like Heartbleed [CVE14] allowed extracting this secret key.

A common approach for isolation is to split up an application into multiple processes. However, frequent switches between processes incur a substantial performance overhead. Multiple methods for in-process isolation have been proposed in recent years to overcome this overhead. Some are based on intrusive modifications of the operating system [Bit+08; Lit+16]. Others propose introducing a capability register [Vil+14; Wat+16; ARMb] or alternative changes in hardware [Fra+18; MRD18]. Again others use virtualization features [Bel+12; Liu+15; Kon+17] and privilege rings [LSK18; Wan+20] of the CPU. It is also possible to implement in-process isolation purely in software [Wah+93; Yee+09; Cas+09; Seh+10]. Some methods are based on protection keys [Zho+14; Che+16; Hed+19; Vah+19; Sch+20] to isolate memory. Most of these papers only discuss the *memory isolation aspect* of in-process isolation but treat *interaction with the kernel* as an orthogonal problem. However, for in-process isolation to be secure, it is necessary to consider all ways isolated memory can be read or modified.

The primary way of interaction between an application and the kernel is the syscall interface. One can restrict this interaction by syscall filtering. Multiple mechanisms have been used for syscall filtering in Linux: *Seccomp* [Linb], *Ptrace* [JS00], *Ptrace* in combination with *Seccomp* [KZ13], *Seccomp* filters that can trap to user-space [Cor18], and custom kernel modules that redirect syscalls to user space [Gar03; GPR04]

In this thesis, we analyze how in Linux the syscall interface can violate isolation boundaries of in-process isolation schemes. We extend *Donky* [Sch+20], an in-process isolation framework using memory protection keys, with sophisticated user-space syscall filtering. Donky uses a trusted *monitor* comparable to an operating system's kernel to manage transitions from one security domain to another. As the domains cannot directly access the monitor, it is possible to execute filter code there. Thus, in contrast to other user-space syscall filtering frameworks, we can securely filter syscalls in the *same process* as they are emitted. Additionally, in Donky, it is possible to construct hierarchical security relationships. By allowing each domain to register syscall filters for its child domains, we reflect these relationships in syscall filters.

We compare all syscall filtering mechanisms listed above plus three new mechanisms inspired by *user-mode syscall handling* for RISC-V [RIS] described in the original Donky paper [Sch+20]. These three mechanisms support our novel form of *syscall delegation*, which allows running syscall filters not only in the *same process* but also in the *same thread*. We call the new mechanisms Indirect-Jump-Delegation, Kernel-Module-Delegation, and Ptrace-Delegation.

Next to the self-protection filters needed for the Donky monitor, we also use syscall filtering for sandboxing domains. For example, each domain can be confined to its own local storage-like folder.

We evaluate the performance of the different syscall interception mechanisms with micro- and macro-benchmarks. The macro-benchmarks consist of a wide range of *unmodified* compute- and IO-heavy applications.

For our novel interception mechanisms, the overhead for a fast syscall like `getpid` is 200%, whereas Ptrace measures 18642% overhead. For the local-storage filters, we measure 57%–84% overhead for the `open` syscall. For self-protection of our evaluated applications, our novel interception mechanisms have 0–20% overhead, whereas existing mechanisms have 5–71% overhead (depending on the syscall frequency). We measure 0–156% overhead for our fast interception mechanisms when using our local storage sandbox with the evaluated applications.

This thesis is part of a paper currently in submission. The nested filtering was designed and implemented in collaboration with the co-authors and also advisors David Schrammel and Samuel Weiser. The design and implementation of the Kernel-Module-Delegation was primarily done by Michael Schwarz and co-author Samuel Weiser.

Outline. In Chapter 2, we provide background on memory isolation and syscall filtering. We discuss in-process isolation and especially the Donky framework in more detail. In Chapter 3, we look into related work in the field of in-process isolation and user-space syscall filtering. In Chapter 4, we present our same-process syscall filtering based on the Donky in-process isolation framework. We discuss possible filter rules in Section 4.6 and propose filter modes for self-protection and a local storage sandbox. In Chapter 5, we evaluate the performance of the implemented syscall interception mechanisms and filter modes. We discuss various implementation details and elaborate on future work in Chapter 6.

Chapter 2

Background

With today's applications getting bigger and more complex, it becomes harder to keep them secure. Two approaches are commonly used to prevent vulnerabilities in a generic way: *mitigation of certain types of exploits* (e.g., preventing certain types of memory corruption bugs [Sze+13]) or *isolation of components* (i.e., *sandboxing* [AR00; Bit+08; Cas+09; Che+16; Sch+20; Gar03; GPR04; JS00; Mad+13; Wah+93]). For the former, detailed knowledge of a wide variety of exploits must be known. As one cannot implement countermeasures for all types of vulnerabilities, a finished application might potentially still contain some. For the latter, it is sufficient to know about the components and libraries of the developed application, how they should be able to interact, and in which aspects they should be isolated.

The term *sandboxing* was introduced by Wahbe et. al for describing the isolation between their software-fault domains [Wah+93]. Today, any mechanism that can isolate one part of code for having effect on the rest can be considered a sandbox. E.g., hardware-backed *system virtualization* [Uhl+05] provides a sandbox for a whole operating system on top of another one, Operating System-backed (OS-backed) *containerization* [Mer14; RFB16; Ker20f] can provide a sandboxed view of the system state for certain applications. Software-based methods, like the original paper by Wahbe et al. [Wah+93], can sandbox parts of applications.

This chapter gives some background on isolation techniques that are already present in modern operating systems and others that have been proposed. It discusses the isolation between kernel and user-space, isolation between processes, and in-process isolation. Furthermore, it discusses syscall filtering as a form of sandboxing in more detail.

2.1 Kernel and User-space Isolation

The privileged part of a modern operating system is called the kernel. Code and data of the kernel are contained in *kernel-space*. User programs on the other hand run in *user-space*. There is an isolation between kernel- and user-space that is enforced by the processor [AD14]. This isolation is visualized in Figure 2.1 by the blue and green box.

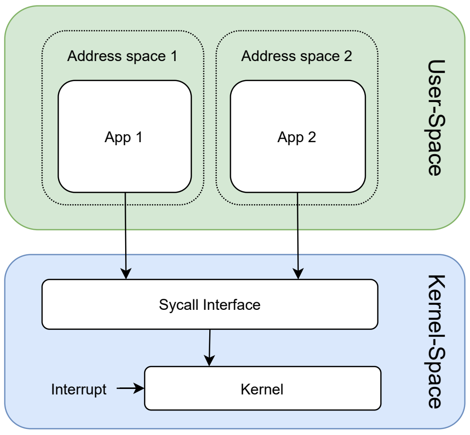


Figure 2.1: Isolation boundaries and interfaces between kernel- and user-space. Dotted lines indicate the boundaries of address spaces. The syscall interface is the main interface to the kernel.

Modern processors can typically operate at multiple levels of privilege. An operating system would use at least two: Supervisor privileges for running the operating system's own code (i.e., *kernel-mode*) and restricted privileges for user programs (i.e., *user-mode*). On Intel processors, these privilege levels are called rings. The operating system runs in ring 0 and has access to all registers and instructions the hardware provides. In contrast, a user program runs in ring 3 and can only use a restricted set of instructions and registers. Rings 1 and 2 are usually unused.

The hardware provides secure ways to transition from unprivileged to privileged mode. The reason for transitions to privileged mode is called an *exception*. Exceptions caused by external events (e.g., input/output) are called *interrupts*. An internal exception (e.g., accessing invalid address, division by zero) is called *fault*. *Traps* are exceptions caused by software interrupts. *Syscalls* are exceptions that are triggered by user programs to invoke privileged functions in the kernel (see Figure 2.1).

The interface between the kernel- and user-space is called the *syscall interface*. Each operating system provides a table of syscalls. Syscalls are referred to by their number and can usually be called by a dedicated `syscall` instruction [AD14].

2.2 Process Isolation

The hardware of a computer is a resource shared between the user programs running on it. An operating system provides the illusion that each user program has full access to the computer. At the same time, it provides isolation between the user programs. I.e., a process is the smallest unit of abstraction, isolation, and executing state [AD14].

To provide the illusion of full access to system memory, each process has its own view on memory. This view is called the process's virtual *address space*. By default, address spaces of individual processes do not overlap and therefore provide isolation between them. In Figure 2.1, they are visualized by dotted lines. Physical and virtual memory is chunked into pieces called *pages*. Their size is usually 4KiB. The *page table* of a process maps virtual addresses to physical ones. In modern processors, page tables have multiple levels. Addresses are resolved in hardware and cached in the Translation Lookaside Buffer (TLB). The memory mappings can only be changed in the kernel. Consequently, a costly context switch is needed to change them. Additionally, unless the TLB is tagged with a process identifier, it must be invalidated for every context switch [AD14].

As a process does not provide any further isolation for memory inside it, a vulnerability in one software component or library can potentially compromise the whole process. Therefore, many application developers split up their applications into small processes to provide appropriate isolation [RG09; Moz]. Interaction between components is usually done via inter-process communication. However, the overhead for a context switch is quite substantial. Hence, it only makes sense to isolate big components that do not interact with each other a lot. A prominent example is the isolation of websites in the *Google Chrome* web browser. Each website runs in an individual process [RG09].

2.3 In-process Isolation

In-process isolation reduces the overhead of switching between isolated components called *domains* while still providing strong isolation between them. I.e., for an in-process isolation scheme to be viable, a domain switch must be faster than a process switch. If this is the case, one can create much more fine-grained isolation boundaries between domains, e.g., isolating a whole library from the rest of an application.

State-of-the-art in-process isolation schemes are primarily concerned with memory isolation. Multiple mechanisms with more or less intrusive changes to the software stack, operating system, and hardware have been proposed in recent years [Bit+08; Yee+09; Bel+12; Zho+14; Vil+14; Wat+16; Lit+16; Hed+19; Vah+19; Sch+20].

2.3.1 Software-based

Pure software-based methods do not rely on hardware features for in-process isolation. Instead, program code is instrumented, so it cannot violate any isolation boundary.

Memory-safe languages, like *Rust* [The20b] or *Java* [AGH00], employ a very intuitive form of software-based memory isolation. As pointers by design can only be used as references to valid objects or a `NULL` object, memory-safety cannot be violated. However, errors in the language's compiler, in *unsafe code* [KN], or in the runtime can still lead to memory safety violations [CVE18; CVE19].

Software-Fault Isolation (SFI) [Wah+93; Yee+09; Cas+09; Seh+10] introduces isolated security domains for unsafe programming languages, like C or C++. Before execution, the application's machine code is checked statically. All direct `loads`, `stores` and `jumps` must not cross an isolation boundary. However, indirect instructions cannot be checked statically. They are replaced by safe instruction sequences. Checks are placed before indirect `loads` and `stores`. These checks validate if source and target address are in the correct domain. Additionally, the target address for indirect jumps and returns needs to validated before they are performed [Wah+93].

For an application with SFI to be secure, it is also crucial that an application cannot modify its own code. Usually, this is enforced by an exclusive write and execute policy (W $\oplus$ X) on the code pages. Thus, writable memory pages are not executable at the same time [Wah+93].

Native Client. For an SFI scheme be both secure and fast, the checks for indirect instructions need to be highly optimized. For example, Native client [Yee+09], a prominent SFI sandbox formerly used in *Google Chrome*, uses segmented memory of x86-32 to limit `loads` and `stores`. Segments also limit indirect jumps. However, x86 is a CISC architecture and therefore uses variable instruction sizes. Native Client must guarantee that instructions are aligned properly. It does so by grouping instruction in 32-byte bundles and allowing only 32-byte-aligned indirect jumps.

Native Client also removes all syscalls instructions inside the target application. Communication with the outside world is done over a trusted runtime library that can be accessed via a trampoline from any segment. The authors measure a maximum performance overhead for CPU-heavy workloads of 12%. The overhead for realistic load is quite small ($< 2\%$).

There is an adapted version of Native Client for the x86-64 and ARM architectures. Its realistic performance overhead is about 7% on x86-64 and 5% on ARM [Seh+10].

2.3.2 Operating System-based

As stated previously, a process is the smallest unit of isolation in modern operating systems. Each process has an associated memory mapping that is loaded when the process is scheduled. However, one could also use memory mappings for in-process isolation. By creating new memory mappings for each individual domain, memory isolation can be decoupled from processes. For a domain switch, only memory mappings are changed but not the process context. Thus, a domain switch is faster than a full context switch.

Note that intrusive changes to the operating system need to be made to support this kind of memory isolation.

Wedge. *Wedge* [Bit+08] introduces so-called *sthreads*. Similarly to *pthreads*, they have a `create` and `join` function. However, the semantics for creating a *sthread* are similar to the `fork` syscall. Every sthread has its own signal handlers, file descriptors, and memory mappings. By default, no memory is shared with the parent sthread. An application can define so-called call gates to support secure function invocation for sthreads. Long-running sthreads wait on calls coming in. They execute them, and provide the return value to the caller similarly to `pthread_join`. Memory for arguments must be marked for it to be accessible to the callee.

Light-weight Contexts. *Light-weight contexts* (*LwC*) [Lit+16] implement an isolation scheme with memory mappings in the FreeBSD kernel [The]. Memory isolation is completely decoupled from any scheduling context, i.e., the memory mapping is neither owned by any process or thread. This has positive performance implications. A switch between LwCs (i.e., domains) takes approximately half the time as a switch between processes.

Each light-weight context can have its own set of file descriptors and BSD capabilities [Wat+10] inherited from the parent context. If a light-weight context does not have the capability to execute a specific syscall, the parent can emulate it.

2.3.3 Virtualization-based

Modern x86-64 processors come with dedicated hardware for virtualization. This feature is called *Intel VT-x* on Intel processors and *SVM* on AMD processors. For this thesis we will just focus on *Intel VT-x* [Uhl+05].

Hardware virtualization is primarily used for full machine virtualization, i.e., virtual machines. Virtual machines are provided with what looks like full access to all privileged hardware features. They have access to exception handling, can set up page tables, change privilege mode, and have access to the segment registers. However, the effects of privileged instructions stay in the realm of the virtual machine and do not have any effect on the host system running the virtual machine [Uhl+05].

For memory isolation to the host operating system, another set of page tables is used. The Extended Page Table (EPT) feature of Intel CPUs provides another layer of address translation. The EPT translates virtual addresses in the guest to physical addresses of the guest operating system, which are, in turn, virtual addresses in the host operating system. The host finally translates the addresses to actual physical addresses.

Hardware virtualization cannot only be used to virtualize whole machines but also to provide a virtualization abstraction for processes. This makes it possible to use virtualization as a method for in-process isolation. The most trivial form of in-processes isolation uses the privilege rings of the processor. One could set up the page tables so that some memory regions can only be accessed with supervisor privileges (ring 0), others with normal privileges (ring 3). A mechanism like this is used in *Dune* [Bel+12]. For more advanced isolation, other works suggest the use of `VMFUNC` to change the EPT of the virtualized process. This makes it possible for every domain to have its own page table [Liu+15; Kon+17].

Virtualization also allows efficient syscall filtering. The guest can install a native syscall handler in ring 0 that handles syscalls coming from ring 3 of the guest. The `VMCALL` instruction can be used for actual syscalls into the host operating system [Bel+12].

2.3.4 Capability-based

Capabilities are unforgeable tokens of authority that grant access to objects in a system [Fab74]. These objects can be any resource in an operating system. E.g., a file descriptor is a capability. Capabilities have been used in the past as an alternative approach to virtual memory for memory isolation [Fab74]. In a system with capability-based addressing, every pointer serves as a capability granting access to a memory range. The hardware must be able to handle capabilities for indirect loads, stores, and jumps.

Modern capability systems combine virtual memory and capability paradigm to provide in-process isolation. In a 2016 paper [Wat+16], Watson et al. propose fast domain crossing for the “traditional” capability architecture CHERI [Wat+12]. In CHERI, capabilities are stored in memory as “fat pointers”. There is a tag indicating that a capability is present in memory. The hardware guarantees that a capability's rights cannot be increased. There are also capability registers for instruction pointers, stack pointers, and heap data. The memory accessible is described by the transitive closure over all capability registers and capabilities accessible by these registers. For domain transitions, the authors propose *Object capabilities*, which are pairs of capabilities. Calling one would allow the caller domain to share private memory (arguments) to the callee domain and the callee domain to return private memory. For pointer-heavy applications, the authors measured a maximum overhead of 46%. On average, less than 10% overhead is to be expected. Under the name ARM Morello, the CHERI architecture is being adapted in ARM hardware [ARMb].

CODOMS (COde-centric memory DOMains) [Vil+14] can be considered as more coarse-grained capability system. Permissions are granted on page granularity by tags in the page table. These tags represent domain IDs. The instruction pointer acts as capability, granting memory access to pages in the same domain as the current code page. There are *access protection lists* (APLs) that govern cross-domain accesses (e.g., a rule granting domain A to read memory from domain B). To pass arguments by reference to domains that are not necessarily immediate children, the authors use a *Capability register*. This register can temporarily grant access to a memory range for the course of a function call.

2.3.5 Memory Protection Keys-based

Memory protection keys were first introduced in *System/360* processors [IBM64]. These processors used memory protection as the sole form of memory isolation instead of virtual memory, which is in use today. Each page of physical memory had a protection key associated with it. Each process could own one or more protection keys. If the process accessed memory associated with a protection key that it did not own, an exception was triggered.

Modern forms of memory protection can be used in addition to virtual memory. Page Table Entries (PTEs) are tagged with protection keys. The permissions for each key are stored decoupled in a special protection key register [Int19]. Inside the register, a protection key might be loaded in read-only or write-only mode. The protection key register can be privileged (ARM [ARMa]) or unprivileged (Intel [Int19] and AMD [Adv20]). If the register is privileged, applications cannot change permissions in the register on their own but need to interact with the kernel. This means the overhead for changing the register is also quite high. If it is an unprivileged register, there is no guarantee that the application does not change it on its own. However, changing permissions in the register has minimal overhead. Protection keys for which the protection key register is unprivileged are called Protection Keys for User-space (PKU). On Intel CPUs, the unprivileged protection key register is referred to by the acronym PKRU, which stands for Protection Key Register for User-space.

Intel MPK. Intel MPK (Intel Memory Protection Keys) [Int19] is the most prominent implementation of PKU. For Intel MPK, four of the reserved bits in each PTE are used to store the protection key of each page. Hence, in an Intel MPK system, it is possible to have $2^4 = 16$ different protection keys. The current permissions for pages associated with a specific protection key are stored in a 32 bits wide PKRU. Thus, there are two bits per protection key. One bit to disable just writing (`WD`), the other to disable all access (`AD`) to pages tagged with the specific protection key (see Figure 2.2 for more details).

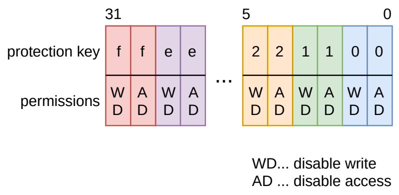


Figure 2.2: PKRU register for Intel MPK. Each key has two control bits: write-disable (`WD`) and access-disable (`AD`).

The final permissions of a page are both traditional page permission bits[1], and the write-disable (`WD`) and access-disable (`AD`) bits of the PKRU register combined. As shown in Figure 2.3, first the PTE for a specific address is resolved. Then the permissions for the protection key are loaded from the PKRU register. PTE permissions and PKRU permissions are combined to determine the effective permissions of the page.

With PKRU set to 0, permissions to all protection keys is granted. 0 is also the default key in each PTE that is not tagged with a protection key. As the protection key register is unprivileged, Intel MPK allows fast permission switches. However, it cannot be used as a security feature on its own.

[1]Traditional page permission bits are: execution-disable (`XD`), writable (`W`), and present (`P`).

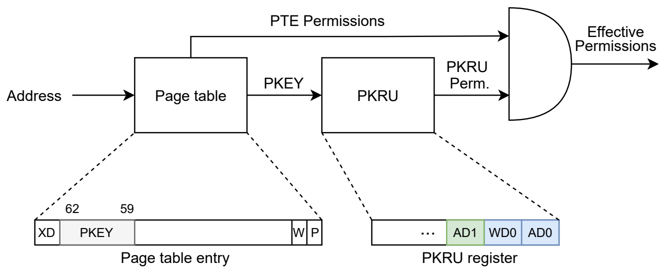


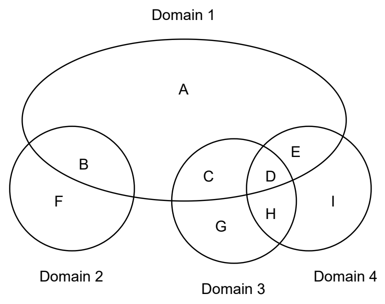
Figure 2.3: Resolving page permissions with Intel MPK. First, permissions in the page table are checked, then the permissions for the protection key (PKEY) in the PTE are compared with the loaded keys in the PKRU register.

Other Architectures with Memory Protection Keys. Other architectures than Intel x86-64 and System/360 also provide memory protection keys. E.g., ARM [ARMa], IBM Power [IBM17], Itanium (IA-64) [Int00], and HP PA-RISC [Hew94]. We discuss the protection key support for ARM and HP PA-RISC in more detail.

ARM has *Memory domains* [ARMa] in the 32 bit version of the ARMv8 architecture. However, domain IDs (protection keys) are only in the first level of the page table. Hence, memory can only be protected at a granularity of 1MB pages. The protection key register is privleged and can deny access, allow access, or allow access and even bypass the regular permissions in the page table.

The HP PA-RISC [Hew94] architecture has 15 to 18 bit protection keys. It uses four key slots (registers) to load a specific key and enable or disable write access to the pages associated with it. The key registers are privileged.

Memory Protection Key Schemes. For in-process isolation schemes based on memory protection keys, a *domain* usually corresponds to one specific configuration of the protection key register [Zho+14; Hed+19; Sch+20]. Shared memory between domains can be implemented by having one protection key for each domain's private memory and one for each region of shared memory between a pair of domains. In Figure 2.4, each letter corresponds to a protection key. The configuration of the protection key register for a specific domain contains all the keys for domain-private and domain-shared memory.



Domain	PKRU Configuration
1	{A, B, C, D, E}
2	{B, F}
3	{C, D, G, H}
4	{D, E, H, I}

Figure 2.4: Sharing memory between domains in a memory protection key system. Each letter corresponds to a protection key.

A memory protection key scheme needs to have a secure way for switching domains, i.e., a secure call gate. A domain must not change the protection key register on its own, giving it more privileges than it was intended to have.

For architectures with a privileged protection key register, usually a syscall is used for modifying permissions. To make this syscall a secure call gate, it could, e.g., control the instruction pointer and always continue execution at a secure entry point of the target domain [Zho+14]. PKU schemes can provide a secure call gate in software by statically analyzing the binary and only allowing domain switches in trusted sections of code [Che+16; Vah+19]. Donky [Sch+20] proposes a secure hardware call gate mechanism.

2.4 Donky

Donky [Sch+20] is an in-process isolation framework based on PKU. It is proposed as a hard- and software co-design. Donky provides isolation for security domains with complex trust relationships. A domain is defined by a set of protection keys and the permissions associated with them. Additionally, domains can define functions that are secure entry points into them (so-called *dcalls*). An overview of the system and a exemplary domain structure is shown in Figure 2.5. Donky domains are a pure user-mode concept. Consequently, the kernel does not need to be modified to use the Donky framework.

Donky is designed for an unprivileged protection key register (PKRU). Thus, transitions are very fast, but the register needs to be protected against unsolicited changes. Donky provides a trusted reference *monitor* as the only place where the PKRU can be modified. This monitor is visualized by an orange box in Figure 2.5. The monitor can only be entered via a secure hardware call gate that enables access to the PKRU. The monitor is in user-space and protects its memory with its own memory protection key. When in the monitor, protection keys are not enforced, i.e., all memory accesses are allowed.

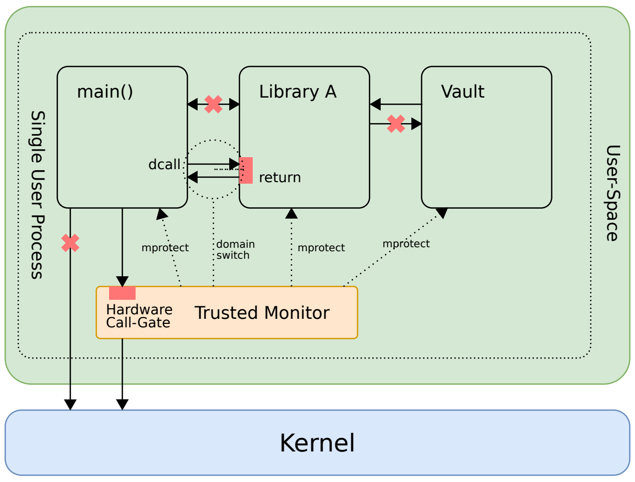


Figure 2.5: Donky system overview with exemplary domain structure. The trusted monitor manages domains and protects their memory. It handles exceptions in user-space. Any memory violation or syscall is delegated to the monitor. The main-function lies in the root domain. It cannot access Library A's memory but only use predefined functions (so-called *dcalls*) to interact with it. Additionally, Library A cannot access any memory of the root domain. The vault can only access memory of Library A but not the other way around.

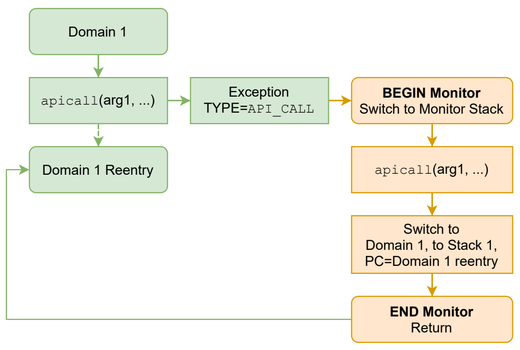
Multi-threading. Donky also works with multi-threaded applications. There is a protected *user stack* for each thread and domain combination allocated automatically when a thread enters a domain the first time. An additional *exception stack* (only accessible to the monitor) is allocated for each thread. It is used, when the thread is inside of the monitor. This prevents a domain from passing a fabricated stack pointer that would make the monitor write to a memory region the domain does not have access to. It also protects the monitor's stack from being corrupted by any other thread.

Thread-Local Storage (TLS) is generally unprotected, as it needs to be accessible by all domains in a thread. However, there is a part of the TLS used for bookkeeping and therefore is only accessible by the monitor. It is called the Trusted Thread-local Storage (TTLS).

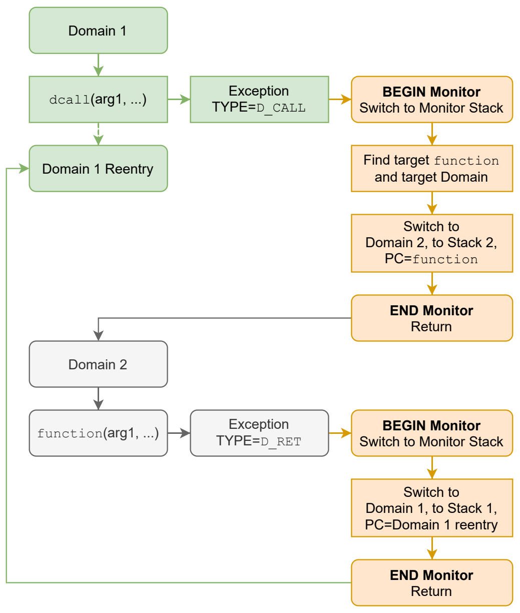
Hardware Call Gate. The hardware call gate mechanism extends the PKRU by one bit, the *monitor bit*. After entering the monitor via the call gate, the bit is set in hardware. Only in this state, the current thread can write the PKRU. Additionally, the thread also has full access to any page tagged with a protection key. When exiting the monitor, the bit is cleared by the hardware. For untrusted domains, protection keys are enforced and trigger an exception. Protection key exceptions are configured to trigger the call gate and to be delegated directly to the monitor instead of the kernel.

Depending on the address the exception is triggered on, it is an actual protection key exception by a domain, a deliberate *API call*, or a deliberate *dcall*. For a deliberate call, the exception needs to occur on a specific address (e.g., the address of the monitor's entry point). Any other address would be considered as an ordinary protection key exception. The call gate is exited by the exception handler returning, i.e., after the return-from-interrupt instruction.

A secure version of this scheme is implemented in the original paper for the RISC-V architecture [RIS]. The "Standard Extension for User-Level Interrupts" (*N-extension*) [RIS20] is used for delegation of exceptions to user-space. A protection key mechanism with the hardware call gate is provided as a custom extension to the instruction set. There is also a version of Donky for x86-64 with Intel MPK. As there is no hardware call gate, its behavior is only emulated in software. Thus, it is not secure.



(a) API call.


Domain 1
dcall(arg1, ...)
Exception TYPE=D_CALL
BEGIN Monitor Switch to Monitor Stack
Find target function and target Domain
Switch to Domain 2, to Stack 2, PC=function
END Monitor Return
Domain 1 Reentry
Domain 2
function(arg1, ...)
Exception TYPE=D_RET
BEGIN Monitor Switch to Monitor Stack
Switch to Domain 1, to Stack 1, PC=Domain 1 reentry
END Monitor Return

(b) Dcall.

Figure 2.6: Donky API calls (for executing an API function inside the monitor) and Dcalls (for executing a function inside of a different domain) using the hardware call gate for entering and exiting the Donky monitor.

API Calls. The monitor can be entered in the form of API calls. The monitor provides API calls for creating and deleting domains, as well as for creating, assigning, and freeing protection keys. The monitor also has API functions for allocating, protecting, and freeing the memory associated with protection keys. Furthermore, there is an API for defining entry points into domains.

As shown in Figure 2.6a, a domain can execute an API call by triggering an exception of type `API_CALL`. By entering the exception handler, the domain enters monitor mode. The monitor switches to its exception stack and calls the specified API call with the given arguments in monitor mode. Next, the monitor switches back to the caller's domain and stack. It gives back control to the calling domain by returning from the exception handler. The monitor provides the return value according to the C calling convention.

Dcalls. *dcalls* are secure entry points for one domain to enter another one. A domain registers functions in the monitor and defines from which domain they may be called. Another domain can call such function by performing a *dcall*.

As shown in Figure 2.6b, a domain does so by generating an exception of type `D_CALL`. The monitor finds the associated target domain (in Figure 2.6b this is "Domain 2") and target function of the *dcall*. It switches to the target domain and sets the return address of the exception handler to point to the target function. Thus, after returning, the target domain executes the target function. The target domain returns from the *dcall* by triggering an exception of type `D_RET`. The monitor then switches back to the original domain and stack. After the *dcall*, execution continues in the original domain. The return value is accessible according to the C calling convention.

Kernel Interaction. As the kernel has the role of a supervisor, it can circumvent memory protection. The kernel manages PTEs, which contain the protection keys. A user-space program has access to syscalls that can change PTEs. Donky and most other PKU-based in-process isolation systems must guard syscalls that can do so. In the x86-64 version of Donky, a kernel module was needed. In this thesis, we show how syscalls can also be filtered in user-space.

2.5 Syscalls

In a modern operating system, an application on its own does not have the privileges to make permanent changes to the system. For this purpose, the kernel provides low-level API functions called *syscalls* [AD14].

Most architectures offer a specific instruction to trigger a syscall. If there is none, software interrupts can be used. The kernel has a dedicated entry point for syscalls, called the *syscall handler*. As shown in Figure 2.7, the syscall handler internally calls the requested syscall function in the kernel.

The user application can reference a syscall by its number. Syscall number and arguments are commonly passed from the application to the kernel via registers [AD14]. On Intel x86-64, a syscall can be triggered by the `syscall` instruction. The syscall number is passed in the `%rax` register, the arguments in `%rdi, %rsi, %rdx, %r10, %r8` and `%r9`. Only six arguments are supported. The return value is placed in the register `%rax`. The entry point for the syscall handler can be set up by writing to the `LSTAR` Model-Specific Register (MSR) [Int19].

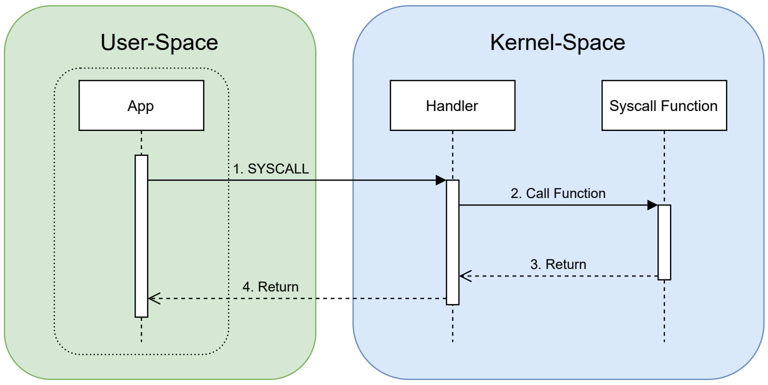


Figure 2.7: Sequence diagram of a Syscall.

The syscall API is *not platform independent.* In theory, an operating system can define different syscalls for every platform [AD14]. An architecture might also have platform-specific syscalls.

Furthermore, the syscall API is *not stable over time.* Although in Linux, existing syscalls are usually kept for backward compatibility reasons, some were actually removed. Bagherzadeh et al. studied the changes in the syscall API in Linux from 2005 to 2015 [Bag+18]. Examples of removed syscalls found were `set_zone_reclaim`, `perfctr` and `nfsservctl`. The authors document multiple added syscalls. The main addition over the last years were "sibling syscalls". These syscalls can be broadly categorized into syscalls with more arguments (e.g., `dup` and `dup2`), syscalls with bigger arguments (e.g., `truncate` and `truncate64`), file descriptor relative syscalls (e.g., `open` and `openat`), backward compatibility syscalls (e.g., `vm86` and `vm86old`), real-time syscalls (e.g., `rt_sigreturn` and `sigreturn`), and batch execution syscalls (e.g., `sendmsg` and `sendmmsg`).

Also some syscalls with completely new functionality were added. Most of them are in the categories "controlling" and "synchronization" (e.g., `inotify`, `getcpu`, `eventfd`, `signalfd`) [Bag+18].

2.6 Syscall Interposition

Syscall interposition is a method to restrict the effects an user application can have on a system [Jon93; Gol+96; AKS98; AR00; JS00; Gar03; GPR04; KZ13; SN19]. E.g., by denying all syscalls but `read`, `write`, and `exit`, an application can only interact with the standard input and output but not with any other system resources.

Generally speaking, a syscall filter is a hook either executed before (*enter-filter*), after (*exit-filter*), or instead of a syscall (*syscall emulation*). Depending on the syscall interception mechanism, only some of these hooks might be possible.

As syscall hook and target code usually do not run in the same thread or even in the same application, a hook cannot (directly) reference resources passed to the hooked syscall. E.g., if the hook runs in a different application, it does not have access to the same address space as the target application. Thus, it cannot directly dereference pointers inside of syscall arguments. Depending on the interception mechanism, it might be possible to do it indirectly via syscalls like `process_vm_readv` and `process_vm_writev` [Ker20m]. Additionally, syscalls inside hooks cannot reference the resources of the target application. E.g., a file descriptor used inside a syscall by the target application cannot be used for syscalls in hooking code.

Syscall Filtering. For syscall filtering, the original syscall is executed in the same context (i.e., thread) as it was issued. However, there is a hook before and after the syscall. The *enter-filter* before the syscall can access and modify the syscall number and arguments. The *exit-filter* after the syscall can inspect and modify the return value. A syscall is denied, by setting the syscall number to an invalid value, e.g., -1. It is allowed, by leaving both *enter*- and *exit-filter* empty.

Notable syscall filtering systems are Janus [Gar03], Seccomp [Linb], and Ptrace [Ker20o]. Whereas with Ptrace one can register both *enter*- and *exit-filter*, Janus only supports *enter-filters*. Seccomp additionally restricts its filter code by not allowing it to modify syscall number or arguments. Moreover, Seccomp does not support deep argument inspection.

Syscall Emulation. Replacing a syscall completely by its hook is called *syscall emulation*. If the emulation code is executed in a different application, it only has indirect access to the resources of the target application. This makes it hard to emulate only a subset of syscalls. Examples for syscall emulation systems are Ostia [GPR04], gVisor [You+19], and *Seccomp User Trap* [Cor18].

Mechanisms. Syscall interposition can be implemented in many different ways. In Table 2.1, we give an overview of the main characteristics of different mechanisms.

The first column divides the mechanisms into a group that *filters* and one that *emulates* syscalls. The next column indicates whether a mechanism executes the filter hook in kernel- or user-space. The third column shows if a syscall hook can be stateful. The fourth column indicates wether a mechanism can inspect integer arguments, the fifth column if they can also change them. The sixth column states whether a mechanism has access to the contents of buffers passed to syscalls. The last column shows whether the syscall hooks are executed in a separate process.

In the following sections the mentioned syscall interposition mechanisms are described in more detail.

	filter/emulate	kernel/user	stateful	inspect args	modify args	read/write buffers	separate process
Seccomp	filt.	kernel	✗	✗	✗	✗	✗
Seccomp BPF	filt.	kernel	✗	✓	✗	✗	✗
Kernel Module	emu./filt.	[1]	✓	✓	✓	✓	[1]
Ptrace	filt.	user	✓	✓	✓	✓	✓[2]
Ptrace+Seccomp	filt.	user	✓	✓	✓	✓	✓[2]
Seccomp User Traps	emu.	user	✓	✓	✓	✓	✓[2]

[1] Can make a decision on its own or delegate it to a user process.
[2] We show that a separate thread is sufficient (see Section 4.5).

Table 2.1: Categorization of syscall interposition mechanisms.

2.6.1 Seccomp

Seccomp (SECure COMPuting) is designed as a utility to reduce the exposed kernel surface inside of a user application in Linux. The filtering is implemented directly in the kernel. With Seccomp, an application can restrict itself to only use a small subset of syscalls. Once a filtering policy is set, it cannot be removed. The policy is also preserved over `clone` and `exec` syscalls. In its strict mode (`SECCOMP_MODE_STRICT`), a predefined policy dictates that only the most essential syscalls are allowed: *read*, *write*, *exit* and *sigreturn*. In filtering mode (`SECCOMP_MODE_FILTER`), more fine-grained filters can be employed. Filters are implemented in the form of Berkeley Packet Filters (BPFs) [Ker20q].

Berkeley Packet Filters. BPFs were originally used as a safe way for pre-filtering network traffic directly in the BSD operating system's kernel. They provided a performance advantage to mechanisms where every packet had to go through a user program. BPFs are written in the machine language for the BPF virtual machine. This virtual machine is quite simple and consists only of an accumulator, an index register, a scratch memory store, and an implicit program counter. Instructions can load fields of an incoming data structure into registers and perform arithmetic and bitwise operations on them. With the store instruction, the contents of a register can be stored in scratch memory. Simple branch instructions can be used for control flow. A return instruction returns the result of the filter. BPFs are stateless, i.e., there is no global memory that is shared between the filters [MJ93].

BPFs were also integrated into the Linux kernel. Today, they are used for packet filtering, but also other filtering tasks, like syscall filtering.

Seccomp BPF. The kernel executes a registered Seccomp BPF program in the syscall handler before any syscall. A Seccomp BPF program gets a `seccomp_data` struct, which contains a field to identify the architecture on which a program is running, the instruction pointer, the syscall number, and an array with the six syscall arguments. As Seccomp does not have access to main memory, it cannot dereference pointer arguments[2]. Thus, syscall filtering with Seccomp is quite limited.

As shown in Figure 2.8, a Seccomp filter can have multiple different outcomes. The most common ones are `SECCOMP_RET_ALLOW` and `SECCOMP_RET_ERRNO`. `SECCOMP_RET_ALLOW` lets the syscall execute normally. For `SECCOMP_RET_ERRNO` the syscall is blocked and an error number is returned. `SECCOMP_RET_KILL_PROCESS` and `SECCOMP_RET_KILL_THREAD` would kill the application or the current thread immediately [Ker20q].

If a process installs multiple Seccomp filters, they are executed after each other until the syscall is denied or the last filter allows the syscall.

eBPF. There is an extended version of BPF in Linux called eBPF [Lina]. It supports ten registers instead of two. Filters can be written in a restricted version of C. The kernel compiles them just-in-time. There is also a way to persist data inside the filter. The data can be shared between user-space and kernel inside of so-called *maps*. However, this version of BPF is not available for Seccomp and would also not be able to inspect buffer arguments[2].

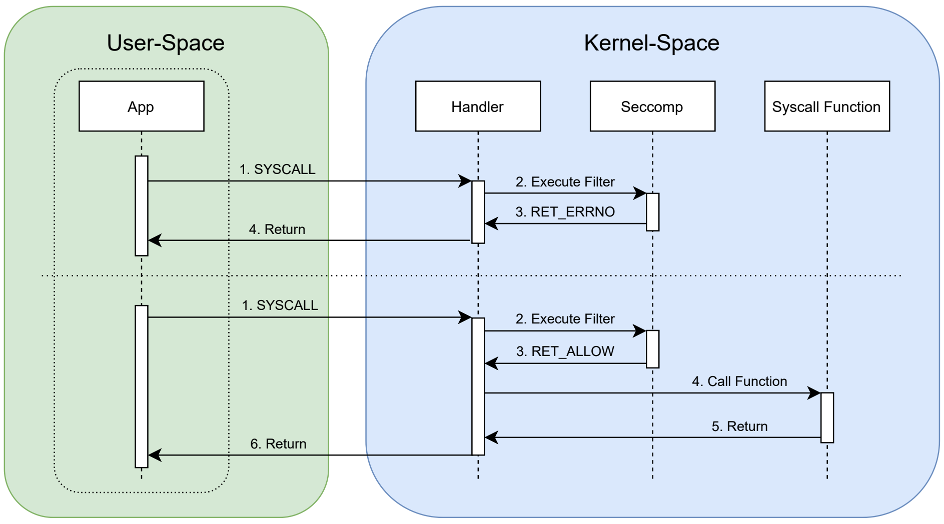


Figure 2.8: Sequence diagram for Seccomp. There are two alternative code paths: `SECCOMP_RET_ERRNO`: an error is returned to the application inside of *errno*, `SECCOMP_RET_ALLOW`: the syscall is executed.

[2]User-space buffers are not yet copied to the kernel. Thus, they are not safe to deference at this stage in syscall handling [Linb]. Note, there has been some discussion about including the ability to inspect buffer arguments into Seccomp BPF for the new `clone3` [Ker20a] and `openat2` [Ker20i] syscalls [Edg20].

2.6.2 Ptrace

`ptrace` is a syscall that allows one process to observe and control (i.e., *trace*) another one. In this thesis, we call the controlling process *tracer* and the controlled process *tracee*. Ptrace is designed for implementing debugging in Linux. Nevertheless, it can be used for syscall filtering. Usually, the tracer would initialize itself and then spawn a child process to which the tracer would then attach [Ker20o].

Syscall Tracing. We discuss the details of syscall filtering with Ptrace on the basis of the simplified sequence diagram in Figure 2.9. Ptrace is signal driven. In Unix, signals are a simple form of inter-process communication. A process can signal itself or another process. The operating system can also send signals to a process. This happens, e.g., when a hardware exception occurs (e.g., SIGSEGV for invalid memory accesses) [Ker20s].

After attaching to the tracee, the tracer informs the OS with a call to `PTRACE_SYSCALL` that it should signal the tracee before the next syscall. Once a tracee gets a signal, it is put in a stopped state. The tracer uses the `wait` syscall [Ker20v] to wait for an event like this to happen. The tracer executes an *enter-filter* before the syscall (2.). It can call any Ptrace command on the tracee and has full access to registers and memory of the tracee. It can change syscall number and arguments by modifying the registers they are stored in. Additionally, the tracer can inspect and modify buffers behind pointer arguments. A syscall is blocked by setting the syscall number to -1. In addition to the Ptrace commands, the tracer can also use the more efficient `process_vm_readv` and `process_vm_writev` syscalls to read and write memory of the tracee.

After the *enter-filter*, the tracer continues the execution of the tracee by calling `PTRACE_SYSCALL` (3.). This communicates the kernel to stop the tracee once again after the syscall is finished [Ker20o]. If the tracer has not blocked the syscall, it is executed in the original thread context (4. and 5.). The tracer can call an *exit-filter* after the syscall (6.) and continue execution (7.) again with `PTRACE_SYSCALL` to be informed about the next syscall in the tracee. Meanwhile, the tracee continues execution at the reentry point of the syscall (8.).

Limitations. Although Ptrace is very versatile, its interface for syscall filtering is quite cumbersome.

E.g., multiple Ptrace syscalls are needed to trace a single syscall in the tracee. In addition to syscalls for waiting on and resuming the tracee, syscalls for retrieving register state and memory contents are necessary. As this needs to be done for every syscall, the overhead is relatively high.

A multi-threaded tracee makes tracing even more complicated: Thread creation needs to be detected and new threads attached to the tracer.

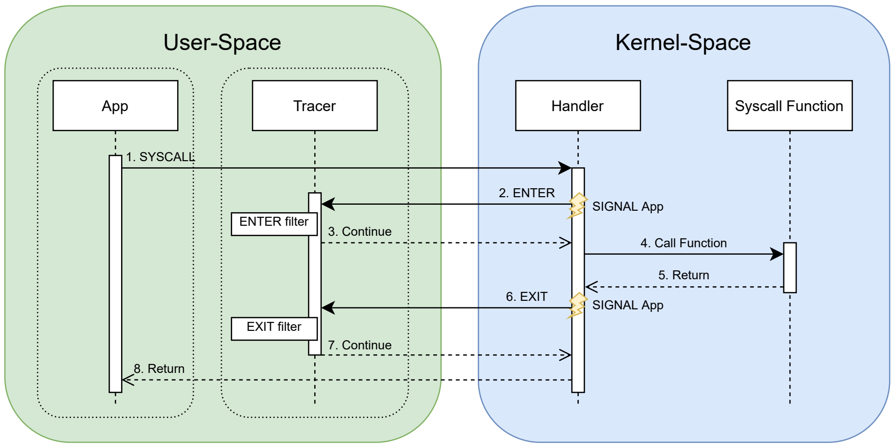


Figure 2.9: Simplified sequence diagram for Ptrace. Communication between kernel and tracer is primarily done via calls to the `ptrace` syscall. The callbacks to the tracer are implemented by the tracer waiting on the tracee to be signaled (symbolized by yellow lightning symbols).

2.6.3 Ptrace+Seccomp

Ptrace (see Section 2.6.2) can be combined with Seccomp (see Section 2.6.1) to use the advantages of both of mechanisms. As shown in Figure 2.10, Seccomp can be used for fast pre-filtering in the kernel (1.-2.). It can deny syscalls that need to be blocked regardless of the arguments and allow those safe to execute (not visualized in the figure). For more complicated filter decisions, Seccomp can delegate the filtering decision to a tracer application. With the Seccomp command `SECCOMP_RET_TRACER` (3.), a special Seccomp signal is sent to the tracee before the syscall is executed in the kernel. A Ptrace tracer process would observe the signal and handle it in the same way as for traditional tracing with Ptrace.

Compared to Ptrace, the overhead can be reduced dramatically by only tracing syscalls where a simple Seccomp filter cannot make the filtering decision.

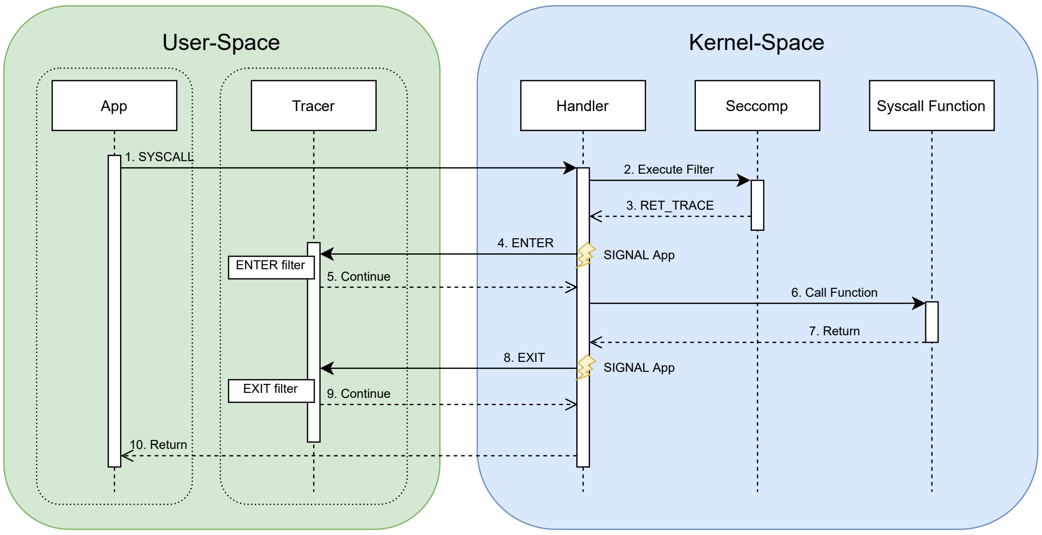


Figure 2.10: Sequence diagram for Ptrace+Seccomp. Seccomp is used to pre-filter syscalls in the kernel. Same simplifications as in Figure 2.9.

2.6.4 Seccomp User Trap

Similar to Ptrace+Seccomp, there is a platform-independent mechanism that pre-filters syscalls in the kernel but can also execute syscall hooks in user-space. In this thesis we call this mechanism *Seccomp User Trap*, which is short for “seccomp trap to user-space” [Cor18]. As Seccomp User Trap is designed for syscall interposition, the interface for intercepting syscalls and tracing child threads is much cleaner than that of Ptrace. E.g., by the design of Seccomp, all new threads inherit the same Seccomp filter and thus are attached to the same tracer automatically.

In contrast to Ptrace and the other previously mentioned syscall interposition mechanisms, Seccomp User Trap does not filter syscalls but it emulates them. Rather than executing filters in the tracer and the syscall in the tracee, with Seccomp User Trap, all code needs to be executed in the tracer.

Syscall interposition with Seccomp User Trap works as follows: A `seccomp` syscall with the flag `SECCOMP_FILTER_FLAG_NEW_LISTENER` returns a file descriptor. As shown in Figure 2.11, for each call to `SECCOMP_RET_USER_NOTIF` inside the Seccomp filter (3.), a tracer-like process receives an event on the file descriptor (4.). The data transferred to the tracer consists of an ID, the tracee’s Process ID (PID), and the `seccomp_data` struct. This struct is the same as for Seccomp BPF filters. Similar to ptrace, the tracer process can read and write arbitrary memory in the tracee via `process_vm_readv` and `process_vm_writev` syscalls. Thus, a filter can also dereference buffer arguments.

Once Seccomp redirects a syscall into the tracer, it can only deny or emulate the syscall in its own process (see Figure 2.11). For many self-directed syscalls like `getpid` or `futex` there is not a simple way to do so. In Linux 5.5, the flag `SECCOMP_USER_NOTIF_FLAG_-`

CONTINUE was introduced [Cor19], allowing the tracer to additionally continue a syscall after it is handed over to the tracer.

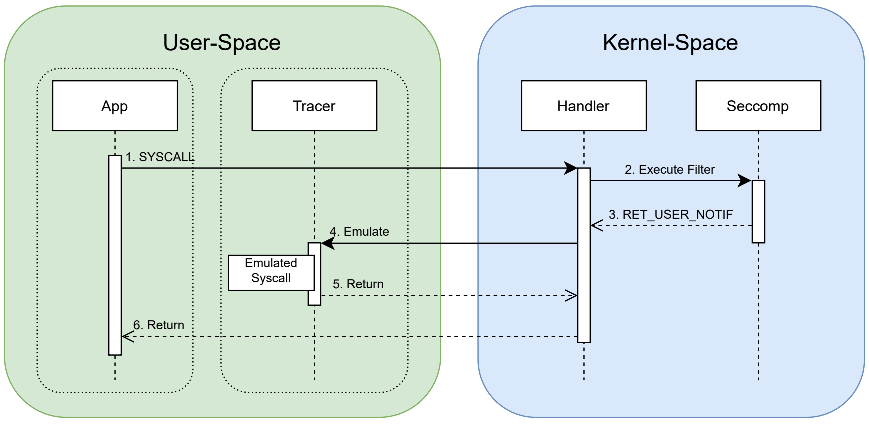


Figure 2.11: Sequence diagram for Seccomp User Trap. Inside syscall filters, the tracer can only deny or emulate syscalls for the tracee (or use SECCOMP_FILTER_-FLAG_NEW_LISTENER).

2.6.5 Kernel Module

By writing a custom kernel module, one can create a customized syscall interposition mechanism. A kernel module runs with the same privileges as the rest of the kernel. Therefore, it can hook the syscall table or syscall handler internally.

Syscalls can be filtered directly in the kernel, in user-space, or both. The classic Janus sandbox [Gar03] pre-filters syscalls in the kernel by hooking only a subset of them. As shown in Figure 2.12, the kernel module redirects each hooked syscall to a user-space tracer process (2.). The tracer can make complex decisions based on the syscall's arguments and either allows or denies the syscall. It returns the result back to the kernel module (3.). If the syscall is allowed, the kernel executes the original syscall function (4.) and the module passes the return value back to the application (5.-6.). For a denied syscall, the kernel module can directly return an error (not shown in Figure 2.12).

Note that the architecture in Figure 2.12 serves only as an example. One can easily implement Ptrace-style syscall filtering (see Section 2.6.2) by additionally jumping into the tracer after each syscall.

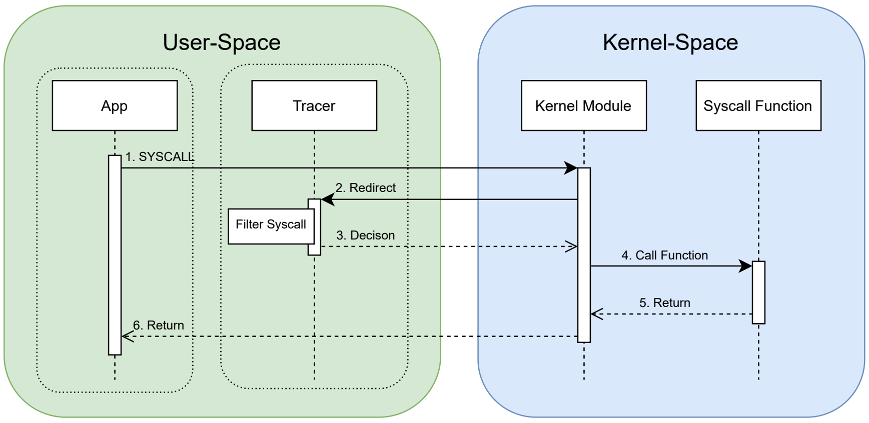


Figure 2.12: Sequence diagram for syscall filtering with a kernel module. This sequence diagram describes the architecture of Janus [Gar03]. Syscalls are redirected to a tracer after the kernel module intercepts them.

Chapter 3
Related Work

The user-space syscall filtering presented in Chapter 4 is based on the memory protection key in-process isolation framework *Donky* [Sch+20]. In this chapter, we present other in-process isolation mechanisms based on memory protection keys. We also show related work in the field of syscall filtering.

3.1 In-process Isolation with Memory Protection Keys

In this section, we take a detailed look at memory protection key-based in-process isolation systems *ARMLock* [Zho+14] (using ARM memory domains), *Hodor* [Hed+19], and *ERIM* [Vah+19] (both using Intel MPK).

3.1.1 ARMLock

ARMLock [Zho+14] is based on *ARM memory domains.* As the protection key register for ARM is privileged, ARMLock uses a kernel module for domain switches. The authors only make small modifications to the rest of the kernel. ARMLock is designed to isolate modules in the form of shared libraries from the host and from each other. The exported functions of the libraries act as call gates between domains. The standard PLT/GOT mechanism of shared libraries is used to provide stubs for each library function. When calling an exported function, instead of the actual function, an `ARMLock_CALL` to the kernel module is made. The buffers for pointer arguments must either be copied by the kernel module or reside in shared memory that is accessible by both domains. The kernel module performs the domain switch and returns control to the *entry gate* of the target domain. The target domain checks if the transition is allowed. If so, it executes the original function. For returning, `ARMLock_RETURN` is called. As a result, the kernel module switches back to the caller domain's *return entry gate*, which then returns control to the caller. The domain switch is visualized in Figure 3.1. The overhead for an inter-domain call is about 2 null-op syscalls (like `getpid`).

ARMLock uses Seccomp BPF for syscall filtering inside isolated modules. The syscalls `fork` and `exec` are only allowed for the host. Memory related syscalls, like `mprotect`, `mmap`, and `brk`, are closely monitored. There can be individual filters for each domain. The kernel module also changes Seccomp filters when transitioning domains.

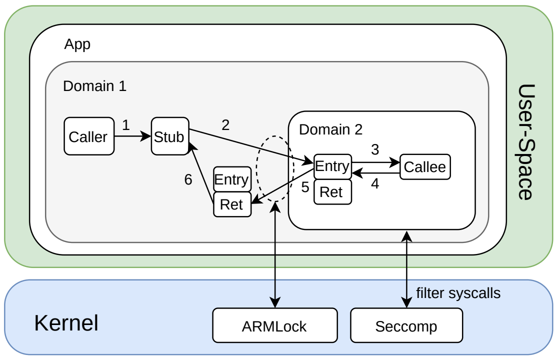


Figure 3.1: Overview of ARMLock in-process isolation. Domain switching is done in the ARMLock kernel module. After a domain switch, the target domain is entered at a predefined entry point.

3.1.2 Hodor

Hodor-PKU [Hed+19] proposes in-process isolation for Intel MPK. Hodor's main use-case is to isolate libraries that provide kernel-bypass access to hardware, e.g., for network or disk IO. The memory for these libraries should be isolated from the host application and from each other. As the protection key register on Intel x86-64 is unprivileged, it can be changed from user-space. Therefore, Hodor needs to protect itself from unsolicited changes to the register. Intrusive changes to the kernel are necessary for Hodor to function properly.

Hodor uses binary scanning to unmap all code pages with `WRPKRU` instructions. However, in x86-64, instructions do not have a fixed length. Thus, it is not decidable if a specific opcode is reached. Figure 3.2 shows two assembly snippets. One, where `WRPKRU` is explicitly called, the other, were the opcode appears as part of another instruction. Hodor unmaps all pages that have an explicit or implicit occurrence of `WRPKRU`. When the instruction pointer encounters such page, the application traps into the kernel. The kernel would place one of the four hardware watchpoints at each occurrence of `WRPKRU`. If an `WRPKRU` opcode is actually reached, it would not be executed, but the hardware watchpoints would be triggered and trap into the kernel. The kernel can then decide if it allows the instruction to proceed.

Hodor does not use syscall filtering to safeguard memory mapping syscalls but modifies the `mmap` and `mprotect` syscalls directly in the kernel. Hodor keeps track of the memory mappings in the kernel and checks if the current domain is allowed to change a specific mapping. Just-in-time compiled code is not supported in the original paper. However, support could be easily added by monitoring `mprotect` syscalls and re-scanning a page once it is changed from writable to executable.

Additionally, there is an implementation of Hodor that uses dedicated page tables for each domain and one using extended page tables to isolate memory. The authors compare the overhead of a domain switch for each mechanism. They measured 105 cycles for a domain switch with Intel MPK, 268 cycles for switching extended pages table with

VMFUNC, and 577 cycles to switch page tables with a syscall. In contrast, for a normal switch of stacks without domain transition, they measured 9 cycles.

```
...
09 f0       or  %esi, %eax
0f 01 ef    wrpkru
31 c0       xor %eax, %eax
...
```

```
...
8d 04 0f   lea (%rdi, %rcx, 1), %eax
01 ef      add %ebp, %edi
...
```

Figure 3.2: Left: explicit occurrence of WRPKRU. Right: implicit occurrence of WRPKRU. This example is from the official presentation slides of the Hodor paper [Hed+19]

3.1.3 ERIM

ERIM [Vah+19] uses Intel MPK and binary rewriting to provide isolation between a trusted and untrusted component (=domain). ERIM provides trampoline code to switch from one domain to another. This code acts as a secure call gate because the WRPKRU instruction is not allowed outside of the trampoline. The binary analysis tool checks that the control flow is transferred to the trusted component after a WRPKRU instruction. All implicit occurrences (see Figure 3.2) are replaced by equivalent instructions. As a result, a malicious party cannot jump to a WRPKRU opcode without redirecting code flow to the trusted component.

The kernel does not need to be modified for ERIM to be secure. Instead, ERIM uses syscall filters with Ptrace+Seccomp (see Section 2.6.3). It delegates mmap, mprotect, and pkey_mprotect syscalls to a tracer in user-space. The tracer ensures that these syscalls are only allowed to create executable memory mappings if executed from the trusted component. In the untrusted component, they are denied. For just-in-time compiled code, binary scanning can be employed at runtime. Every time an executable memory mapping is created, one could scan for WRPKRU instructions and replace them.

ERIM does not switch stacks for domain transitions and has a slightly lower overhead than Hodor (see Section 3.1.2). A domain switch combined with a direct call needs about 9x as many CPU cycles as the direct call alone.

3.2 Syscall Filtering

In this section, we look at different syscall filtering systems that are similar to ours. *Janus* [Gol+96], Ostia [GPR04], and *Mbox* [KZ13] are all user-space syscall filtering systems that can filter syscalls of a whole process. *gVisor* [You+19] is a Unikernel sandbox [Mad+13] and therefore emulates most syscalls it intercepts. *Light-weight contexts* [Lit+16] provide syscall filtering for their in-process isolation. Their in-process isolation and syscall interception is implemented completely in the kernel.

3.2.1 Janus

Janus [Gol+96], specifically Janus version 2 [Gar03], provides a syscall sandbox for Linux. It uses a kernel module for intercepting syscalls and the Janus policy engine in user-space for allowing or denying them. The kernel module also has very primitive syscall filters on its own. E.g., a `read` or `write` syscall is always allowed, as it can only operate on previously opened file descriptors. There is negligible overhead for syscalls that are filtered this way. The Janus system is visualized in Figure 3.3.

Janus is highly configurable. Filter modules for managing access to categories of syscalls can be loaded and configured by policy files. For example, there are modules for restricting filesystem and network resources. Policies in the configuration files are ordered from most general to least general. When a filtered syscall is executed, policies are evaluated in the same order as in the file. In contrast to Seccomp (see Section 2.6.1), a *deny* in a more general filter can still be overridden by a later *allow*. Internally, a policy registers filtering functions for each syscall affected by it. These functions are saved in a per-syscall linked list that is traversed on each invocation of a syscall. A filter has access to the syscall number and arguments. It can use the kernel module to inspect and modify buffers passed to it. This feature is necessary to implement, e.g., filtering of filesystem paths.

Janus also supports sandboxing of multi-threaded processes. The authors identify common types of pitfalls encountered when trying to implement syscall filtering for a (multi-threaded) application. Pitfalls they identified are "incorrectly mirroring state and code of the operating system" in the policy module (e.g., state of file descriptors, code for path canonicalization), "overlooking indirect paths to resources" (e.g., a core dump can write files), Time-Of-Check-Time-Of-Use (TOCTOU) race conditions between the policy decision and the syscall to be actually executed (e.g., syscall argument races, filesystem path races), and "side effects of denying syscalls" (e.g., denying privilege dropping syscalls).

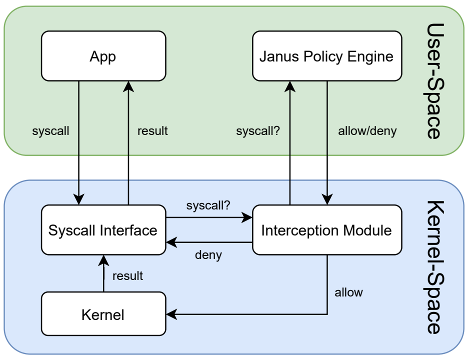


Figure 3.3: Overview of syscall filtering with Janus. Syscalls are intercepted inside a specialize kernel module. Syscall filters are executed in the *Janus Policy Engine*.

3.2.2 Ostia

In contrast to Janus, which uses a filtering architecture, *Ostia* [GPR04] features a *delegating architecture.*

Ostia uses a kernel module to allow all uncritical syscalls (e.g., `read` or `write`). All other syscalls are delegated to the emulation library in the target application. The emulation library makes an inter-process call to a so-called *agent* to access sensitive resources. The agent can then acquire sensitive resources (e.g., file descriptors) and perform syscalls on them. It decides if a specific syscall is allowed by consulting a policy similar to the one in Janus. The authors use race condition-free call sequences in the agent to overcome TOCTOU race conditions in filesystem paths as suggested in [Gar03].

Not all syscalls are implemented in the agent (trusted code) but are provided by the emulation library (untrusted code) as a combination of trusted syscalls. Thus, the Trusted Computing Base (TCB) is reduced significantly compared to the Janus sandbox.

3.2.3 Mbox

Mbox is a sandbox based only on syscall filtering in user-space. It uses Ptrace combined with Seccomp (see Section 2.6.3) for intercepting syscalls. Seccomp is used to pre-filter syscalls that are allowed or denied by default. For more complicated filter decisions, the syscall is delegated to Ptrace. As shown in Figure 3.4, Mbox overlays the host's filesystem by intercepting filesystem relevant syscalls. By default, any read access to a file is granted without the path being rewritten (path points to host "filesystem"). If a file is opened with write access, its contents are copied to the sandbox's root directory (path points to sandbox "filesystem"). All subsequent reads and writes are done on the copied file. Their design is heavily inspired by a filesystem named UnionFS. Also, the idea is similar to Linux namespaces [Ker20f]. In contrast to namespaces, Mbox does not need root privileges to operate.

Mbox also provides a feature to "hide" paths from the sandbox. As the name indicates, it is not a security feature, and files can still be accessed by symbolic links. The overhead of Mbox depends a lot on how filesystem intensive the workload is. For very compute-intensive workloads, the authors measured negligible overhead, for IO-intensive tasks, it is a lot bigger. They measured 45% overhead for compiling the Linux kernel.

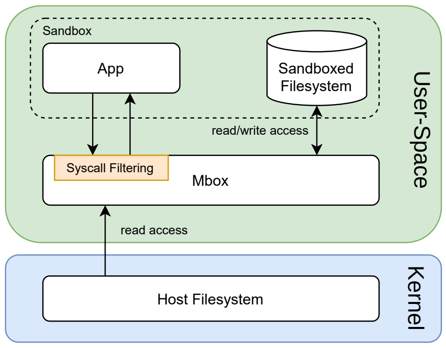


Figure 3.4: Mbox provides an overlay over the existing host filesystem by filtering syscalls.

3.2.4 gVisor

gVisor [The20a] is a sandbox for safe execution of unmodified Linux binaries. It provides an interface similarly to *runc* [Doc15] for running micro-services in more secure "containers". Instead of just filtering syscalls, gVisor introduces another layer of isolation between application and operating system. Their *Sentry* is a so-called *Unikernel* inside of the application. It completely emulates the behavior of 211 syscalls by using only 55 syscalls [You+19], thus reducing the attack surface of the host kernel. The Sentry does not have access to any filesystem resources. File access is managed by a separate process named *Gofer*, similarly to *Ostia's* [GPR04] *agent*.

Under the hood, gVisor supports two mechanisms for syscall interception: Ptrace (see Section 2.6.2) and Intel / AMD hardware virtualization (see Section 2.3.3). When using Ptrace, the tracer and Sentry are in a separate process. With hardware virtualization, the Sentry can reside in the same process as the user code. The Sentry runs in guest mode with privilege ring 0 and the user code in ring 3. Syscalls from user code are executed in the Sentry. Syscalls issued by the Sentry are delegated to the host, as they are configured to exit the guest layer.

The overhead for applications can be quite significant. Syscalls are typically 2.2x, memory allocations 2.5x, and file opens 216x slower than direct execution on the host.

3.2.5 Light-weight Contexts

Light-weight contexts (*LwC*) [Lit+16] provide OS-based in-process isolation for FreeBSD (see section Section 2.3.2 for more information on the in-process isolation employed by LwC). They offer syscalls to create isolation domains with similar semantics as the `fork` syscall. By default, when a new domain is created, all memory, file descriptors, and BSD-capabilities of the parent domain are inherited by the child domain. The parent domain can restrict these resources at the time of creation or at a later stage.

If a domain does not have the capability to execute a specific syscall, the operating system delegates the syscall to the parent domain. The parent domain can register functions for emulating syscalls of child domains. The syscall hook can emulate the syscall purely in user code, execute syscalls on its own, or execute syscalls on behalf of the child (or any other) domain. This form of syscall "impersonation" is possible for any domain that has the capability to execute a specific syscall and has access to the target domain. Syscalls executed in this way run in the same context as the original syscall, i.e., they have the target domain's file descriptor table, run in the same thread, and use the target domain's memory mappings. However, it is also possible to change parts of the context to the context of the parent domain, e.g., using a file descriptor of the parent.

Chapter 4

Our Syscall Filtering for PKU Systems

In this thesis, we present same-process syscall filtering for frameworks using Protection Keys for User-space (PKU) for in-process isolation. While PKU systems isolate memory of mutually distrusting domains, all other process resources, including kernel resources, are still shared. An attacker can use syscalls to access the same resources as any other domain. The Linux kernel enforces protection keys for buffers passed in syscall arguments. However, the threat model of protection keys in the Linux kernel is different from that of in-process isolation [Han; Ker20j]. Thus, some syscalls break the isolation provided by PKU, e.g., `mprotect`, `ptrace`, or `process_vm_readv`, to name a few.

We employ syscall filtering to ensure that isolated domains cannot interfere with each other. We implement our syscall filtering purely in *user-space*. We use the *Donky* [Sch+20] PKU framework as basis and demonstrate how to build our same-process syscall filtering on top of it (see Section 4.3). In contrast to traditional user-space syscall filtering systems [Gol+96; JS00; Pro03; Gar03; GPR04; KZ13] that need a *second process*, we can isolate the application and filter code in the *same process* with the help of PKU. Furthermore, we can not only register syscall filters for the *whole process* but also for *each domain* in the process.

We repurpose existing mechanisms like, Ptrace, Ptrace+Seccomp, and Seccomp User Trap (see Section 2.6) to support same-process filtering for multiple isolated domains (see Section 4.5). Moreover, we propose a novel form of *syscall delegation* that can filter syscalls by redirecting them into a trusted reference monitor in the same application (see Section 4.3.5).

We discuss, how we can use the hierarchical nature of Donky domains to securely support nested syscall filtering (see Section 4.4). We analyze the syscall interface to find potentially harmful syscalls (see Section 4.6). With this information, we propose comprehensive filter rules for two use-cases. For the first use-case, we provide *self-protection* for the Donky monitor (see Section 4.6.3). This use-case includes comprehensive rules to guarantee the claimed security of the isolation system.

Often, it is not sufficient to only isolate memory. E.g., if a isolated domain uses the file system to save sensitive information, other *untrusted* domains still have access to this information. Thus, in the second use-case, we further isolate non-memory kernel resources for each domain (see Section 4.6.2). We confine each domain to a *local storage*-like directory and allow a domain only to use file descriptors it opened before.

4.1 Requirements for PKU system

We build our same-process syscall filtering framework on top of an existing PKU system. We use the PKU system to isolate application from the syscall filter code. We have following requirements for the PKU system.

The system must provide ($\mathcal{R}1$) two or more isolation domains, ($\mathcal{R}2$) memory isolation that disallows direct or indirect loads and stores to memory of other domains. We use one domain for the application, the other for the syscall filtering. Additionally, the PKU system needs to have ($\mathcal{R}3$) a secure way to transition from one domain to the other. There must be a separate stack for each domain ($\mathcal{R}4$). For each syscall, we transition from the application domain to the filter domain. We need separate stacks, so the application domain cannot access local variables or change the control flow in the filtering domain.

Donky. The in-process isolation framework *Donky* [Sch+20] fulfills these requirements. A domain is described by a set of memory protection keys that allow access to all pages tagged with one of them (fulfilling ($\mathcal{R}1$) and ($\mathcal{R}2$)). Changing the PKRU is only permitted from inside the trusted monitor. The proposed hardware changes provide a secure call gate to transition from an untrusted domain to the trusted monitor. As Donky supports multiple domains, the monitor also provides a way to transition from one domain to another securely ($\mathcal{R}3$). Donky has an stack for the trusted monitor and each untrusted domain ($\mathcal{R}4$). We build our syscall filtering on top of Donky. However, we also discuss using other PKU systems, like ERIM [Vah+19] or Hodor [Hed+19] in Section 6.2.

4.2 Threat Model

Assuming an attacker inside an isolated domain, they should not be able to access or modify memory or change the control flow of any other domain. The attacker may perform arbitrary memory accesses or modifications and is allowed to jump to arbitrary code. They may execute arbitrary syscalls but must not compromise any other domains by doing so.

As Donky does not prevent a malicious domain to mount a denial-of-service attack (i.e., a domain executing a endless loop blocks the whole thread), we also do not consider denial-of-service attacks via syscalls to be part of our thread model (i.e., a domain sleeping indefinitely). Similarly to other user-space syscall filtering frameworks, we do not support syscall filtering for binaries with elevated privileges.

Our Trusted Computing Base (TCB) consists of the hardware, the operating system's kernel and the trusted Donky monitor. We assume that the trusted components are free of bugs.

4.3 Overall Design

Memory protection keys provide isolation in memory. However, as the unmodified Linux kernel does not have a notion of security domains, it cannot provide any isolation for non-memory resources. These include process-specific resources, like, page tables, file descriptors, scheduling context, or signal handlers. Global operating system resources, like, the filesystem, are shared too.

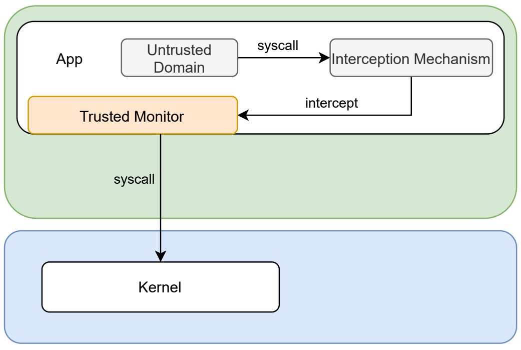
By filtering syscalls, we restrict access to specific resources. Traditional user-space syscall filtering systems would filter syscalls in another process (= *tracer* process), so their memory is isolated completely from the target application. However, a tracer process would still not have any knowledge about our in-process domains.

With the help of Donky's in-process isolation, we isolate application and filter code. Therefore, we can filter syscalls in the *same process* as they are generated. The Donky monitor manages transitions between domains and thus has full knowledge about the domain structure. We use this information to implement domain-aware syscall filtering. Another advantage of our design is that syscall filters can directly access all memory of the target application. This includes buffer arguments for syscalls. We do not need a syscall or shared memory to read or write these buffers.

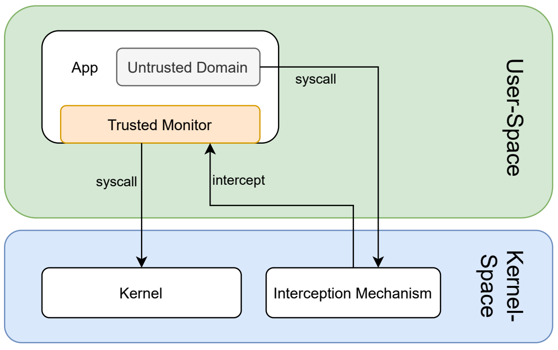
Donky has a hierarchical domain structure where parent domains can constrain the accessible memory of any child domain. Similarly, we allow an untrusted domain to restrict a child domain's non-memory resources with *domain filters*. We propose the use of *monitor filters* for the Donky monitor to protect itself.

An application starts in monitor mode, allowing it to securely register syscall filters before running any actual application code. Filter functions for monitor filters are stored as function pointers in a global syscall table. This table is protected and can only be written in monitor mode. Thus, untrusted application code cannot remove any syscall filters. Additionally, domains can register domain filters at any time. Analogous to the syscall filter table for the monitor, there exists a per-domain table for domain filters. Any child domain cannot read or write to the syscall table of their parent.

We provide multiple mechanisms for intercepting syscalls emitted by an untrusted domain of a target application. Broadly speaking, there are interception mechanisms that work purely in user-space (see Figure 4.1a) and mechanisms that intercept the syscall in the kernel (see Figure 4.1b). After the interception, both types of mechanism would redirect a syscall to the trusted monitor inside the target application.



(a) Interception purely in user-space.



(b) Interception in kernel.

Figure 4.1: Our syscall interception. The interception mechanism redirects any syscall issued by an untrusted domain to the trusted monitor in user-space.

4.3.1 Filter Functions

We use a traditional Ptrace-style filtering architecture (see Section 2.6.2) with an *enter-filter* before and an *exit-filter* after any filtered syscall. Similar filter functions can be found in the *strace* [Str] utility. The advantage of doing so is that we can reuse the same filter functions for all of our syscall interception mechanisms. For non-Ptrace mechanisms, we emulate the behavior of Ptrace-style filtering by simply calling the *enter-filter*, then the syscall, and finally the *exit-filter* with the return value of the syscall (see Listing 4.1). Filters at their core are C functions expecting a single pointer to a struct containing the following information: the syscall number, arguments, the return value, information about the domain it originated in, and information about whether we are executing an *enter-* or *exit-filter*. The struct also has a pointer to a memory area that the filter can safely use.

```
SET_SYSCALL_ENTER(&trace_info);
filter_function(&trace_info);    // executes enter-filter
trace_info.ret_value = syscall(trace_info.nr, trace_info.args[0], ...);
SET_SYSCALL_EXIT(&trace_info);
filter_function(&trace_info);    // executes exit-filter
```

Listing 4.1: Emulating Ptrace-style filtering. Executing a filter-function before and after a syscall, providing it with syscall number, arguments, and the return value.

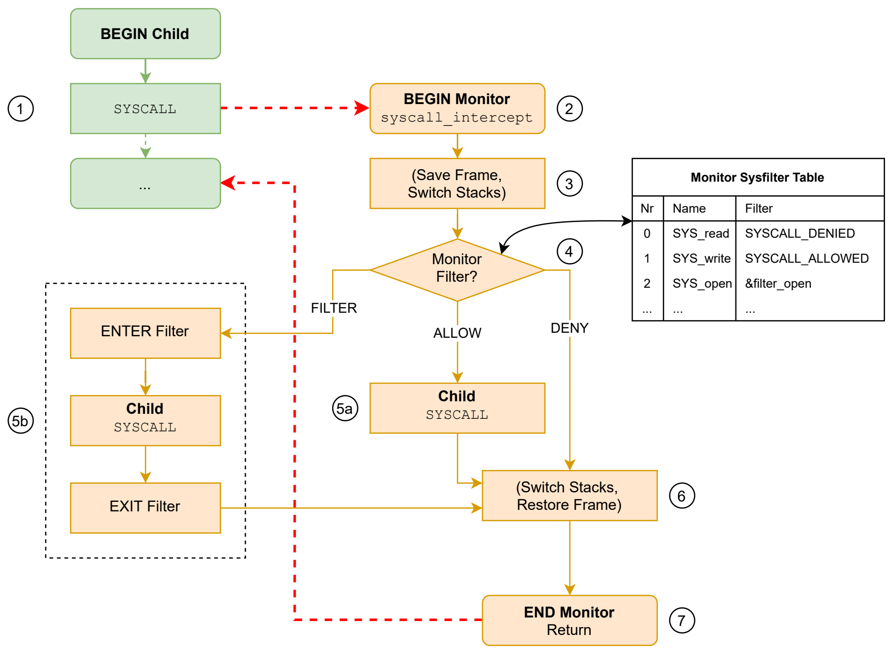
4.3.2 Overview Syscall Filtering

In this section, we give an overview of the general process of our syscall filtering based on Figure 4.2.

When a domain invokes a syscall (1), the interception mechanism delegates it to the trusted monitor (2). The monitor provides a generic syscall handler, similarly to an exception handler but in user-space. First, the monitor saves the stack frame of the current domain and switches to a secure exception stack (3). For some interception mechanisms, like Ptrace, this step is done implicitly. Based on the monitor filter installed for a specific syscall number (4), a syscall can be allowed, denied, or a filtering function is executed for it.

In case of a denied syscall, the syscall handler in the monitor aborts early and returns an error (6, 7). For an allowed syscall, it is executed as if it was not intercepted at all (5a). If a syscall filter function is registered for the syscall (5b), the *enter-filter* is called before the syscall. This filter can inspect arguments of the syscall. It can also dereference pointers in arguments and can access or modify the underlying buffers. Additionally, it can decide to block the syscall. After the syscall, a *exit-filter* is called. The *exit-filter* has access to the same information as the *enter-filter* but can also analyze and change the return value of the syscall.

After the syscall is handled, the monitor switches back to the original domain and the original stack and restores the stack frame (6) to continue execution where it was interrupted (7).



Monitor Sysfilter Table

Nr	Name	Filter
0	SYS_read	SYSCALL_DENIED
1	SYS_write	SYSCALL_ALLOWED
2	SYS_open	&filter_open
...	...	...

Figure 4.2: Our syscall filtering. A syscall is intercepted in the monitor. We execute any registered monitor filter and then return to the instruction after the syscall instruction in the original domain. The red dashed arrows indicate the monitor intercepting a syscall and then returning to the domain that issued it.

4.3.3 Buffer Arguments

We need to handle syscalls with arguments pointing to buffers in user-space with special care. If not handled correctly, decisions based on the contents of the buffers are subject to race conditions. Whereas the six scalar arguments to syscalls are passed in registers, buffer arguments are still passed by memory reference. Thus, other threads can modify these buffers during the execution of the syscall filter. Potential consequences are Time-Of-Check-Time-Of-Use (TOCTOU) race conditions. A check in the syscall filter might pass, but before the executes the syscall, a colluding thread changes the argument to have malicious content. The race condition potentially allows a domain to escape syscall filters imposed on it (see [Gar03]). For example, an attacker passes a valid path to a syscall. A colluding thread changes the path to a malicious one after the filter's execution but before the syscall in the kernel.

Disabling multi-threading trivially solves the problem of the race condition. Alternatively, related work [Gar03; Pro03; KZ13] suggests copying buffers to read-only memory so they are not accessible by other threads.

We also implement argument copying (see Figure 4.3). For our syscall filtering, the monitor copies buffer arguments to a per-thread *argument memory* area protected by a distinct memory protection key. We call this protection key the *syscall-args-key.* As we forbid any other thread to load this key, no concurrent thread can modify a buffer. After the buffer is copied, a syscall filter can safely check and even modify it. The monitor rewrites the syscall arguments to point to the copied buffers. While executing the syscall in the kernel (4), the *syscall-args-key* must be loaded (3). Additionally, we load the keys of the domain that issued the syscall. It is unloaded directly after the syscall (5). As the kernel enforces memory protection keys, not loading the *syscall-args-key* would result in the syscall failing because the kernel cannot access the buffer arguments.

As protection keys are not enforced in the monitor, we can copy the arguments in monitor mode. The monitor first checks if the memory passed in the arguments is actually accessible by the domain that initiated the syscall (1), thus preventing a confused deputy attack [Har88]. Filter functions define which arguments the monitor needs to copy and what sizes they are. After checking the access permissions of the arguments, the monitor then copies them to the *argument memory.* The page mappings (page table structures) and protection keys inside Page Table Entries (PTEs) must not change during this permission check. Otherwise, we would introduce another race condition. We use locking to prevent Donky API functions from changing any memory mappings. After copying, the *enter-* and *exit-filter* can safely access or modify these buffers and make allow/deny decisions based on their content.

Some syscalls also have buffer arguments that the kernel writes to (e.g., the `dup` syscall). The *exit-filter* might want to modify the output from the syscall before passing it back to the domain. To prevent TOCTOU race conditions for checks in the *exit-filter* function (6), the monitor allocates *argument memory* for them and rewrites arguments to point to it (1). After the *exit-filter* function, the content is copied back to the original location (7). Again, we check if the pointers passed to the syscall are accessible by the current domain.

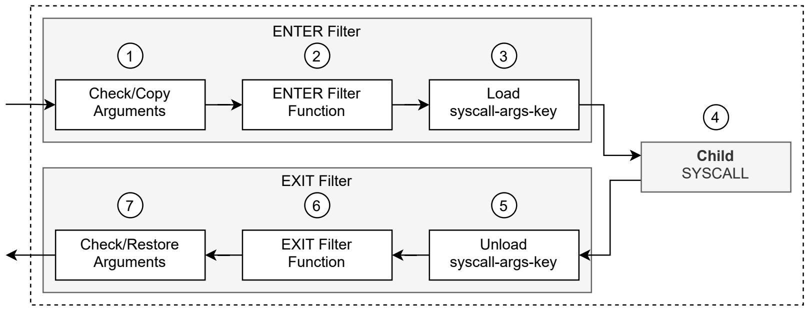


Figure 4.3: Our argument copying for filtering syscalls with buffer arguments. This is a more detailed view of (5b) in Figure 4.2.

4.3.4 Syscall Impersonation

The effect of a syscall highly depends on the context it is executed in. Some syscalls interact with the application, others with the thread that issued it. As our syscall filtering is intended for PKU systems, we also discuss the context implied by the currently loaded memory protection key. We call the procedure of executing a syscall in a different context than the current context *syscall impersonation*. A similar mechanism can be found in *Light-weight Contexts* [Lit+16] (see Section 3.2.5)

Thread/Application Context. The thread context decides for which thread or application a syscall is executed in the kernel. E.g., an `open` syscall would add a file descriptor for the newly opened file to the file descriptor table of the application that executed it. The application can then use the file descriptor for later syscalls. When a syscall interception mechanism cannot run syscalls in the original context, it can only emulate it inside the filter application.

Whereas emulating file descriptors is not very complicated, many threading syscalls cannot be emulated at all. For example, when emulating the `exit` syscall, we cannot exit the application gracefully but only kill it by sending a `SIG_KILL` to it[1].

Memory Protection Key Context. Depending on the memory protection keys loaded, some buffers passed to a syscall might be inaccessible. If we want to execute a syscall inside a syscall filter with the same context as it appears in the target code, we need to ensure that the correct memory protection keys are loaded. When entering monitor mode, the monitor gives itself access to all protection keys. For monitor filters, all syscalls part of the filter code are executed with access to all memory. For the original syscall between *enter*- and *exit-filter*, we drop privileges by just loading the protection keys of the domain that initiated the syscall. Thus, we *impersonate* the syscall for said domain.

[1]Note that there is a semantic difference between an application gracefully exiting and an application being killed. E.g., any registered exit handlers would not run for a killed application.

However, we cannot clear the monitor bit when impersonating a syscall. If we left monitor mode at this point, we would not be able to enter it again. It is only possible to change the PKRU or the monitor bit while the monitor bit is set. Thus, we execute all syscalls in monitor mode. As the PKRU is writable in monitor mode, it is important, that the kernel does not change the register as part of any syscall. For this thesis, we assume this is the case.

We can also impersonate syscalls for other domains than the current one. We do so by loading the protection key configuration of the target domain we want to impersonate before executing the syscall and restore it afterward. We use syscall impersonation for the syscall between *enter*- and *exit-filter* of a domain filter.

4.3.5 Syscall Delegation

In this section, we show that our novel method of *syscall delegation*[2] not only enables us to securely filter syscalls in the *same process* but even the *same thread.* We use Figure 4.4 to illustrate the process.

Our syscall delegation works as follows: After intercepting a syscall (1.), we redirect execution to a specific syscall handler function inside the same application as the syscall was emitted (2.). How exactly the syscall is intercepted and delegated differs for each interception mechanism. For the syscall filtering to be secure, we enter monitor mode before executing any filter code. We disable syscall delegation for all syscalls inside the monitor so that we can execute syscalls at all. We forbid disabling delegation outside of the monitor.

Syscall delegation in combination with Donky allows us to "emulate" the behavior of traditional syscall filtering, like that of Ptrace (see Section 2.6.2). For an allowed syscall, we simply execute said syscall inside of the syscall handler. As the syscall handler is in the same application and thread, the syscall executes in its original thread context. For a denied syscall, we return an error or terminate the application. We emulate Ptrace-like *enter*- and *exit-filters* by calling them before and after the syscall. Our procedure for syscall filtering with delegation is the same as described in Section 4.3.3. After all syscall handling code, we return from the monitor (3.). We directly return to the reentry point of the intercepted syscall in the target domain.

With our syscall delegation, one can filter syscalls in monitor mode. However, we also allow domains to register syscall filters. We execute these filters inside of the domain that registered it. Any syscall inside a domain filter is delegated to the syscall handler in the same way as a syscall in the application code. This allows us to do nested filtering.

[2]Note, our syscall delegation has little to do with the one used in the *Ostia* sandbox [GPR04].

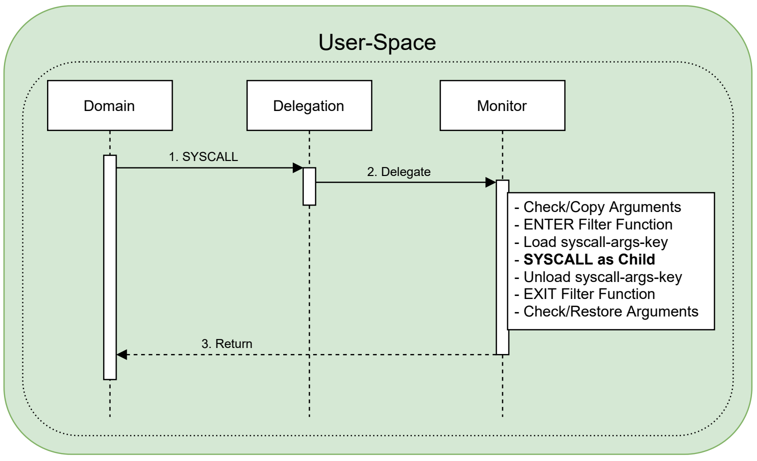


Figure 4.4: Our syscall filtering with *syscall delegation*. We execute filter functions inside the trusted monitor of the same application as filtered syscalls are issued.

4.4 Nested Filtering

Donky can create domain structures in hierarchical relationships. I.e., a parent domain can have access to the child domain's memory but not the other way around. Having this parent-child relationship, it seems natural to also support nested syscall filtering so that a parent can also manage non-memory resources of their child domains.

For our nested filtering, any domain can register syscall filters for its children. Pointers to filter functions are stored in the per-domain syscall filter table of the child. Domain filters themselves are also subject to interception. I.e., if a filter itself uses the syscall instruction, it will be filtered by its parent. Figure 4.5 shows the process of our nested filtering.

Overview. When a syscall is issued in an untrusted domain, it is intercepted by the monitor (1). We traverse the domain hierarchy (2) and distinguish between three cases:

(A) The parent of the current domain has a filter function registered.

(B) The parent denies the syscall and we return to the caller immediately (9).

(C) No parent domain exists. The monitor filters the syscall to ensure self-protection.

For (C), we proceed as described in Section 4.3.2. For (A), the monitor prepares for syscall filtering in the parent domain. It first checks and copy buffer arguments to the calling thread's *arguments memory* (3). We cannot do this outside the monitor, as we can only prevent modification of the page table by acquiring the appropriate lock inside the monitor. Additionally, the monitor loads the thread's *syscall-args-key* inside the parent domain, giving domain filters access to buffer arguments. Then the monitor switches to the parent domain and exits monitor mode (4).

The parent domain (5) executes the domain filter. It first executes the *enter-filter* function, then the syscall, and finally the *exit-filter* function. Any syscall executed in the parent is again subject to syscall filtering. Thus, it would again be intercepted by the monitor and filtered in the same way as described above. When the parent domain finishes its filtering, it returns back into the monitor.

The monitor (6) first restores any buffer arguments, as described in Section 4.3.3 (7). It then switches back to the domain where the syscall was generated initially and exits monitor mode (8, 9). The execution continues at the reentry point of the original syscall.

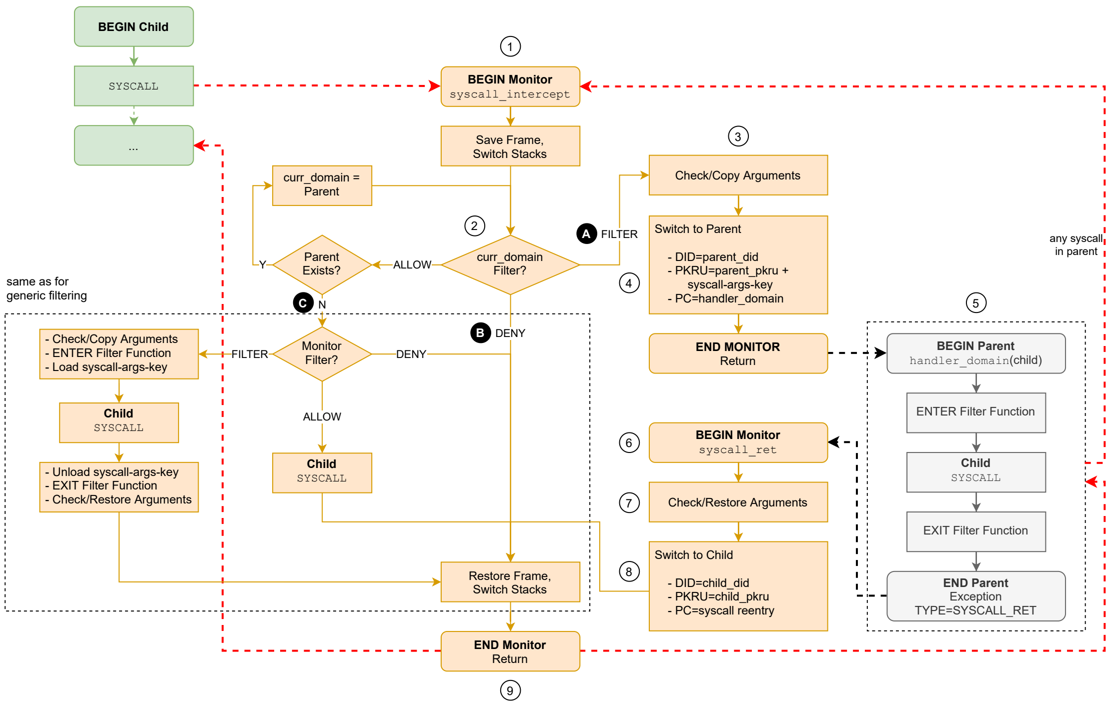


Figure 4.5: Our nested syscall filtering. For any intercepted syscall, we recursively execute any filters of parent domains first, then the monitor filter. The red dashed arrows indicate the monitor intercepting a syscall and then returning back into the domain that issued it.

Syscall Impersonation. When executing a domain filter inside of a parent domain (see Figure 4.5, (5)), a syscall filter might need to execute syscalls on its own. Hence, by default, any syscall is executed with the filter's own protection keys. However, for some syscalls, the filter might want to execute the syscall for the child domain, thus with child's protection keys loaded. I.e., it wants to *impersonate* a syscall for the child domain.

A syscall filter can control impersonation with a thread-local variable (see Figure 4.6). This variable is protected by the per-thread *syscall-args-key*. A domain filter sets the variable to the ID of the child domain it intends to filter the syscall for[3]. By default, we

[3]Note, a domain filter could set the variable to any domain ID. However, a potential attacker cannot do any harm, as the monitor checks if the domain ID for impersonation is a child of the current domain before the syscall is executed.

impersonate the syscall between the *enter*- and *exit-filter*, so it does not have access to any data structures of the filter domain.

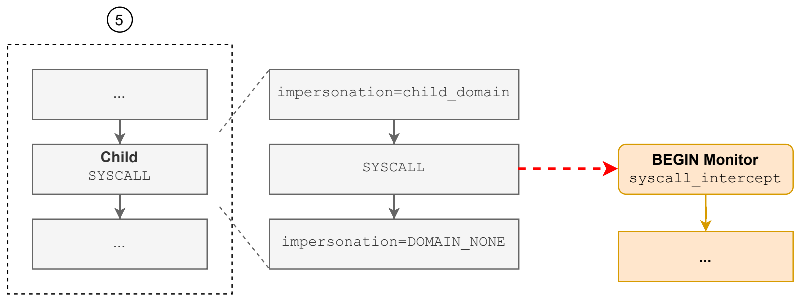


Figure 4.6: Our syscall impersonation for nested filtering. When the monitor executes a syscall for a domain (here `child_domain`), it loads its protection keys. This is a more detailed view of (5) from Figure 4.5.

4.5 Interception Mechanisms

In this section, we look at how different syscall interception mechanisms (see Section 2.6) can provide syscall filtering for Donky. We discuss traditional interception mechanisms with a tracer process (i.e., Ptrace, Ptrace+Seccomp, and Seccomp User Trap). We explore how to make them work in the same process. Moreover, we present novel interception mechanisms that support our *syscall delegation* (see Section 4.3.5).

Interception with Tracer. Traditional syscall interception mechanisms, like Ptrace, Ptrace+Seccomp, and Seccomp User Trap use a so-called *tracer* process to provide isolation between syscall filter code and the target application (the *tracee*). Without this process isolation, it would be easy for a malicious tracee to interfere with the syscall filters inside of the tracer. A tracee could even manipulate the tracer's control flow by overwriting function pointers or return addresses on the stack.

The memory isolation provided by Donky allows us to securely execute the tracer in the same process as the tracee. We isolate tracer and tracee by protecting relevant data structures and the stack of the tracer using memory protection keys. While Seccomp User Trap works with *Pthreads*[4] [Ker20n], Ptrace and Ptrace+Seccomp do not. In Linux, threads are very similar to processes. By setting the `CLONE_THREAD` flag in the `clone` syscall [Ker20a] a new thread is placed in the same thread group as the thread executing the syscall. Threads in one thread group are part of the same application and have access to the same address space. However, one cannot use a thread as a tracee for Ptrace or Ptrace+Seccomp, as one cannot use the `wait` syscall on them [Ker20a]. As waiting on a child to receive a stop signal is an integral part of the Ptrace interception mechanisms, they cannot work.

[4]Pthreads are the default abstraction for threading in the Glibc.

We overcome this limitation by using a Glibc implementation that does not use the `CLONE_THREAD` flag for their Pthread implementation. There are older versions of Glibc with *LinuxThreads* [Hon05] as backend for Pthreads that did not use the flag. However, for our evaluations, we opted for modifying a recent version of Glibc (version 2.31). We simply removed the `CLONE_THREAD` flag for the relevant `clone` syscall inside of `pthread_create` (see Listing 4.2). We additionally added `SIGCHLD` as so-called *termination signal*. We could not wait on any child without setting a termination signal. We removed the `CLONE_SIGHAND` flag, so the tracee does not inherit the same signal handlers of the tracer[5].

```
clone(..., CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SYSVSEM|
      CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID|
-     CLONE_SIGHAND|CLONE_THREAD,
+     SIGCHLD,
      ...);
```

Listing 4.2: Modified `clone` flags in `pthread_create`. Added lines are marked with (+), removed lines with (-).

For domain-aware syscall filtering, a tracer needs to be able to access Donky meta-information of the tracee. E.g., the tracer needs to know the domain the tracee thread currently is in. The monitor stores most meta-information in a monitor-protected area of the thread-local storage of each thread we call the Trusted Thread-local Storage (TTLS). Interception mechanisms like Ptrace can locate the TTLS of the tracee at a constant offset relative to the thread register (`FS` register on x86-64). For other mechanisms, we look-up the PID of the tracee in the global monitor data structures. The TTLS contains a field indicating if a thread is in the monitor or not. It includes the thread's current domain ID, caches the current PKRU configuration, and contains the *syscall-args-key* and more. The tracer adapts its behavior based on the data found in the TTLS. E.g., if the thread is in the monitor, it does not filter any syscalls. It passes the current domain ID to syscall filters so that they can make decisions based on it. Before the syscall is executed in the kernel, the tracer needs to load the *syscall-args-key* in combination with the current PKRU configuration. It cannot use any instruction usually used to set the PKRU register, as they are always directed to the currently running thread. Ptrace and Ptrace+Seccomp can load the PKRU register, as they have access to the extended register state of the processor where the PKRU register is located.

Interception without Tracer. If isolation between the target application and tracer can be provided without process isolation, no tracer process is needed. Traditionally, this is the case for syscall filtering in the kernel, like with Seccomp (see Section 2.6.1) or with custom kernel modules (see Section 2.6.5) that filter syscalls directly in the kernel.

In this thesis, we propose three novel tracer-less interception mechanisms for Donky: RISC-V-User-Mode-Syscalls, Kernel-Module-Delegation, and Indirect-Jump-Delegation (an emulation of RISC-V-User-Mode-Syscalls for x86-64). They all support efficient

[5]Note, these modifications can still cause trouble if an application relies on the fact that a thread does not signal the parent process.

delegation to the trusted monitor. Without a tracer, we do not need to perform the (at least) two context switches required for mechanisms with tracers. We trade them for much faster domain switches. As we filter the syscall in the same thread as it was issued, there is no need to retrieve the Donky meta-information of a tracee indirectly. We can access it directly via the TLS variables that are part of the *trusted TLS* of Donky. The monitor can set the PKRU register for any traced syscall by merely using the provided instructions. Our Ptrace-Delegation mechanism is a hybrid. It intercepts syscalls with a tracer but uses syscall delegation to filter syscalls in the monitor.

4.5.1 Ptrace and Ptrace+Seccomp

Ptrace+Seccomp (see Section 2.6.3) uses Seccomp (see Section 2.6.1) in the kernel to pre-filter syscalls. For each syscall, it executes a previously registered Seccomp BPF program in the kernel. The result of the BPF program decides if a syscall is either allowed, denied or forwarded to a user-space tracer. Combining Seccomp with Ptrace allows fast filtering of syscalls in the kernel with BPF code, while only forwarding more complex decisions to a tracer process. Apart from the pre-filtering, the procedure for filtering in user-space is very similar to the one of Ptrace. For the rest of this section, when we refer to "Ptrace", it is also implied the same holds for Ptrace+Seccomp.

Traditional syscall filtering with Ptrace (e.g., [KZ13]) uses a tracer process to execute filter code. As described above, we can use a tracer *thread* instead of a *process*. Thus, our tracer runs in the same address space as the target application.

We discuss our syscall filtering using Ptrace as interception mechanism based on Figure 4.7. The tracer thread runs a blocking loop, the *tracer loop* and `waits` [Ker20v] for any tracee to receive a `SIGSTOP` signal. The kernel triggers a `SIGSTOP` signal in the tracee before (2.) and after any syscall (4.). In this stopped state, the tracer has full control over the tracee and can execute further Ptrace commands. Note, for each command, the tracer issues a syscall. As we perform multiple syscalls in the tracer to trace a single syscall in the tracee, the overhead adds up fast.

Before any filtering logic, we retrieve values of all general-purpose registers in the tracee by executing the `PTRACE_GETREGSET` command. If the syscall is filtered, we copy buffer arguments of the syscall to *argument memory* (see Section 4.3.3). Next, the tracer executes the *enter-filter* function. The syscall number and arguments are passed to the filter. The filter can then modify them. Changes are propagated to the tracee's registers by a call to `PTRACE_SETREGSET`. The tracer thread runs in monitor mode, so syscall filters also have access to monitor data structures. It has access to the thread-local storage (and also the TTLS) by querying the thread pointer (`FS` register on x86-64). It retrieves information, like the tracee's domain ID or *syscall-args-key*.

As *argument memory* is protected by the *syscall-args-key*, we load the key in the tracee before it executes the syscall and unload it afterward. We do so by also requesting the `XSTATE` (extended state) [Int19] of the processor by executing another `PTRACE_GETREGSET` command. This state is highly platform-specific and can change from processor to processor. However, we can locate the PKRU register for supported x86-64 processors at a constant offset. Unfortunately, with Ptrace we cannot not read or write just parts of the `XSTATE`. Therefore, we resort to loading the whole `XSTATE` before the syscall and write it back with the modified PKRU register. After the syscall, we restore it to the original configuration. In between, the kernel executes the syscall like it was not traced (after 3.). To improve performance, we only request the extended state when a filter requires the *argument memory*, and thus, also the *syscall-args-key*.

After unloading the *syscall-args-key*, the tracer first retrieves the return value of the syscall. It provides this return value to the *exit-filter*. Again, the filter can inspect and modify the arguments. In the end, we restore buffer arguments as described in Section 4.3.3. We also restore all syscall argument registers to their original value because of the syscall calling convention. An exception to this rule is the register with the return value. When the tracer signals the tracee to continue after the syscall (5.), it returns back to the reentry point in the original domain (6.).

As the tracer runs in monitor mode, Ptrace is perfectly suitable for monitor filters. However, Ptrace does not work well with nested filtering. For nested filtering, also nested tracers are required. As each tracer would intercept syscalls from all child tracers and again would need extra syscalls to filter them, this approach is not practical. In the next section, we discuss how to combine Ptrace with our syscall delegation to support nested filtering more efficiently.

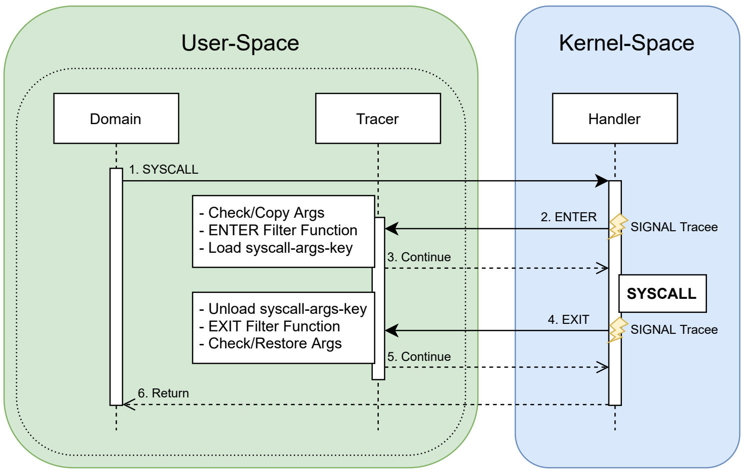


Figure 4.7: Simplified sequence diagram of our syscall filtering using Ptrace as interception mechanism.

4.5.2 Ptrace-Delegation

We use the Ptrace syscall interception to create a novel mechanism for syscall delegation that executes filter code inside the tracee instead of the tracer. We call it *Ptrace-Delegation*. The mechanism is illustrated in Figure 4.8.

Similarly to syscall filtering with Ptrace (see Section 4.5.1), we register a tracer for the target application. The tracer can be a traditional Ptrace tracer, but it can also be combined with pre-filtering using Seccomp. We configure the tracer so that the target application gets a signal before the kernel executes any syscall. The tracer waits on any thread of the target application to receive this signal (2.). In contrast to the traditional Ptrace approach, we do not execute any filter code in the tracer. Instead, we redirect execution in the tracee to continue inside of a syscall handler in the trusted monitor (3.). We do so by changing the instruction pointer to the beginning of the syscall handler. Complying with the syscall calling convention on Intel x86-64, we set the `RCX` register to point to the syscall's reentry point in the target domain. When all registers are set, we order Ptrace to continue execution in the tracee. When the tracee enters the monitor, it saves the `RCX` register to later continue execution at this address. In the monitor, we do nested syscall filtering in the same way as described in Section 4.4. For returning, we switch domain to the target domain, exit the monitor, and jump to the reentry point that we previously saved from the `RCX` register (4.).

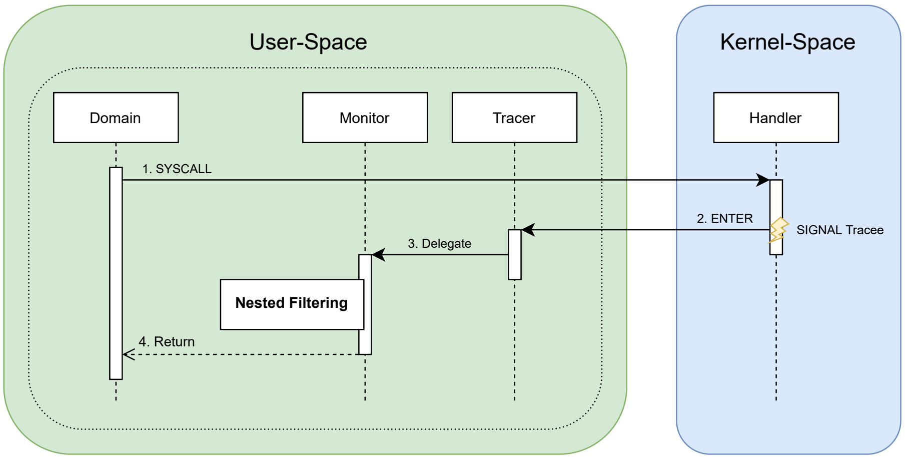


Figure 4.8: Our syscall filtering using Ptrace to delegate syscalls into the Donky monitor.

4.5.3 Seccomp User Trap

Seccomp User Trap (see Section 2.6.4) is not a mechanism for syscall filtering but syscall emulation. For this interception mechanism to work, the tracee registers a Seccomp filter that can selectively allow or deny syscalls or alternatively trap to a user-space tracer. The tracer is usually a separate application, but in our case can also be a thread.

In user-space, the tracer reads on the file descriptor returned by the successfully registered Seccomp filter of the tracee. For each syscall (1.), the Seccomp BPF filter is executed (2.) (see Figure 4.9). For any syscall that traps to user-space, the tracer gets a `seccomp_notif` struct (3.). Most importantly, this struct contains the PID (same as thread id in Linux) of the tracee, syscall number, and arguments of the intercepted syscall. We use the PID to look up the tracee in our internal data structures. The tracer determines if the tracee is in monitor mode and thus should allow the syscall. The data structures also store information about the domain the tracee currently is in. This information is is vital for domain-aware syscall filtering.

As Seccomp User Trap does not support Ptrace-style syscall filtering and only syscall emulation, generally, this means one cannot use the same syscall filter functions as for the other mechanisms. However, as our tracer resides in the same application as the tracee, all kernel resources needed in our filtering functions are shared with the tracee (e.g., file descriptors). Thus, we emulate Ptrace-style syscall filtering (4.) in the tracer similar as described in Section 4.3.3[6]. We handle some safe syscalls directly in the Seccomp BPF code, so we do not need to emulate them.

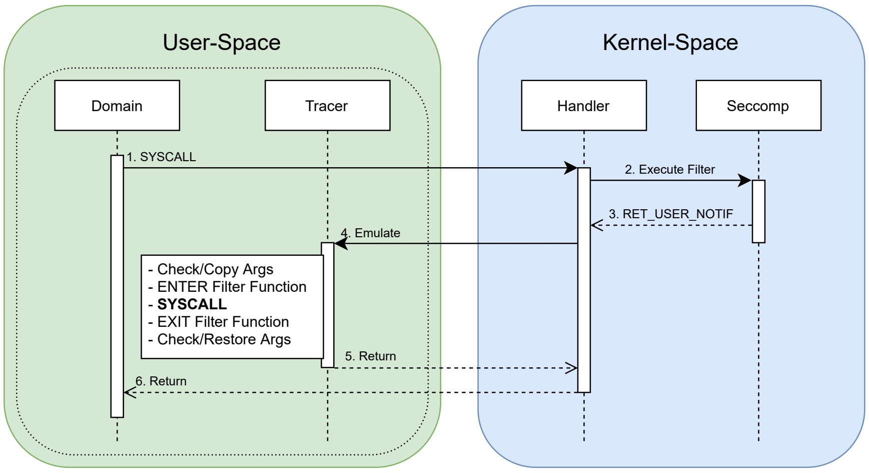


Figure 4.9: Our syscall filtering using Seccomp User Trap as interception mechanism.

4.5.4 RISC-V-User-Mode-Syscalls

Ideally, our delegation mechanism is combined with hardware that natively supports registering syscall handlers directly in user-space (see Figure 4.10). The "Standard Extension for User-Level Interrupts" (*N-extension*) [RIS20] provides us with such support on the RISC-V architecture [RIS]. One can enable certain exceptions to be delegated to user-space by setting appropriate bits in the `sedeleg` control register. Each application can configure exception handlers and communicate the addresses of them to the kernel. E.g., for Donky, we configure the control register to delegate protection key exceptions to our monitor. For our syscall filtering in Donky, we also activate the bit for syscall delegation. For convenience, we delegate syscalls to a separate exception handler.

With the original Donky hardware extension [Sch+20] the user-space exception handlers act as hardware call gate to the trusted monitor. Before the hardware delegates an exception to user-space, the monitor bit is set, after the handler returns from the exception, it is cleared. The hardware disables any delegation of exceptions in monitor mode. Thus, the monitor can make arbitrary memory accesses and syscalls.

Inside the user-mode syscall exception handler we can do nested syscall filtering exactly as described in Section 4.4.

[6]As we emulate the syscall in the tracer running in monitor mode, we do not need to load the *syscall-args-key*.

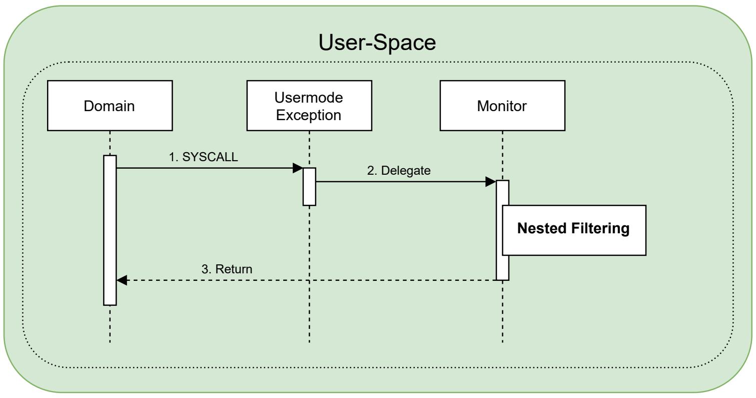


Figure 4.10: Our syscall filtering using RISC-V user-mode exception handling to delegate syscalls into the Donky monitor. For each syscall, the hardware directly jumps to the exception handler in the trusted monitor.

4.5.5 Indirect-Jump-Delegation

Similarly to user-mode syscall delegation in RISC-V (Section 4.5.4), we want to have a fast delegation mechanism on x86-64 processors. We emulate syscall delegation by rewriting the code in Glibc to call our syscall handler instead of issuing a syscall in the kernel (see Figure 4.11). As most syscalls in target applications are issued from Glibc, this is a good approximation to RISC-V delegation. Note, this form of syscall interception is not secure by itself, since an application can still invoke syscalls without going through the Glibc. However, it is very efficient and can be used in combination with a secure interception mechanism to achieve both speed and safety.

For Indirect-Jump-Delegation, we register a syscall handler at a constant offset relative to the thread pointer. We replace every occurrence of a syscall instruction in Glibc with a jump to the address of the syscall handler (see detailed code in Listing 4.3). If no handler is registered (address is 0), we resort to calling the syscall instruction as before.

When entering the syscall handler (or the monitor in general), we first disable syscall delegation by writing 0 to the handler address. When leaving the syscall handler (or the monitor), we restore the handler address. Inside of the syscall handler, we can perform nested syscall filtering as described in Section 4.4.

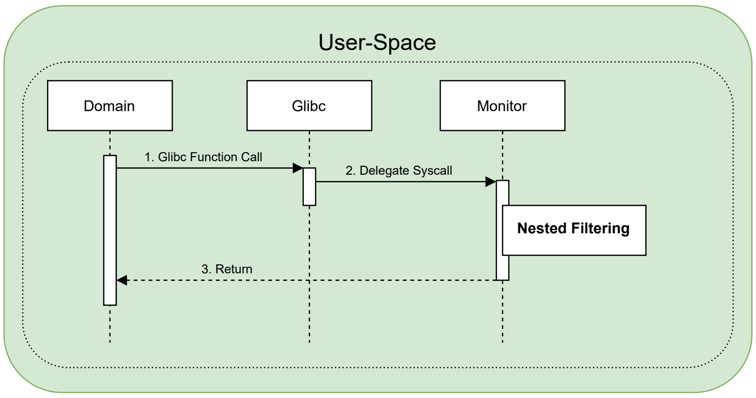


Figure 4.11: Our syscall filtering with modified Glibc. Any Glibc function call that contains a syscalls, makes an indirect jump to the user-space monitor instead.

```
    movq %fs:0x280, %r11    // handler at constant offset to FS register
    test %r11, %r11
    jz fallback             // fallback, if there is no handler
    jmp *%r11               // indirect jump to handler
fallback:
    syscall
```

Listing 4.3: Call to exception handler inside of Glibc. Note, the `FS` register is used as thread pointer in Linux on x86-64.

4.5.6 Kernel-Module-Delegation

We propose using a compact kernel module to support secure syscall delegation for x86-64. Our system is illustrated in Figure 4.12. Similarly to Ptrace-Delegation (see Section 4.5.2), the kernel module only intercepts the syscall (1.) but does not have any filtering logic per se. However, it checks if the syscall is issued from the trusted monitor or an untrusted domain. It allows any syscall issued by the monitor. It delegates any syscall from an untrusted domain to a syscall handler in user-space (2.).

It does so by writing the reentry point (instruction following syscall instruction) to the `RCX` register and the address of a previously registered syscall handler to the instruction pointer (`RIP`). Back in user-space, the monitor knows this convention and saves the reentry point. The monitor can now perform nested syscall filtering as described in Section 4.4. After the execution of all filters, the monitor returns directly to the saved reentry point (3.) without invoking the kernel again. Thus, we reduce the number of context switches compared to traditional mechanisms even further.

The delegation can be configured at program start. The target application registers the address of its syscall handler in the kernel module. Next, it specifies which syscalls the kernel module should delegate. In our proof-of-concept, the kernel module overrides the handler functions for these syscalls in the syscall table inside of the Linux kernel. All other syscalls are allowed by default. Communication between application and kernel module is done via calls to the `ioctl` syscall [Ker20d]. The application does not need to have any special privileges to execute those. We prevent untrusted domains from unregistering delegation by restricting the access to the `ioctl` syscall.

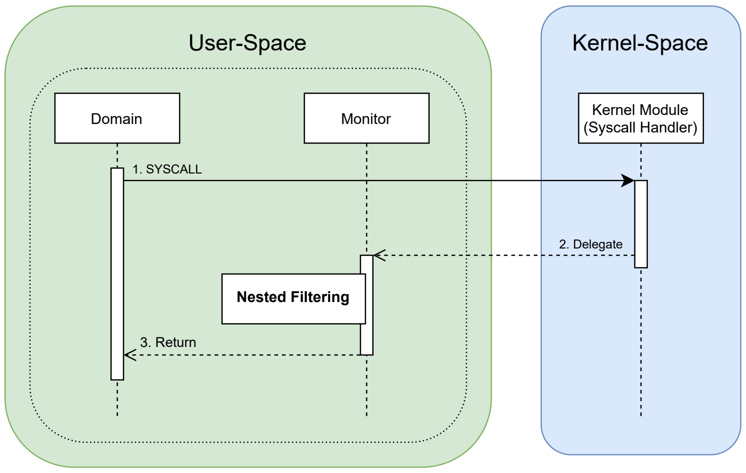


Figure 4.12: Our syscall filtering using a kernel module. For each filtered syscall, the kernel jumps to the user-space syscall handler in the monitor directly.

4.6 Filter Rules

The Linux kernel enforces memory protection keys as part of the ordinary access check for most syscalls[7]. E.g., when passing a pointer to inaccessible memory to a `read` syscall, it does not execute the syscall but returns an error. The check also fails for memory regions protected with a protection key that is not loaded in the PKRU. However, the Linux kernel developers do not treat memory protection keys as security asset [Han; Ker20j], so some syscalls allow a process to circumvent them.

Previous work already suggests that syscall filtering is needed to secure PKU-based in-process isolation. ERIM [Vah+19] (see Section 3.1.3) proposes filtering of memory management syscalls (i.e., `mmap`, `mprotect`, and `pkey_mprotect`) so that untrusted code cannot create or change (executable) memory mappings it does not own. Donky [Sch+20] (see Section 2.4) additionally suggests filtering the syscalls `munmap`, `pkey_alloc`, `pkey_free`, `clone`, and `exit`. Connor et al. [Con+20] systematically analyze the whole syscall interface for the x86-64 architecture. They identify more problematic syscalls for PKU-based sandboxes. Their list also includes `open`, `creat`, `openat`, `mremap`, `remap_file_pages`, `prctl`, `ptrace`, `process_vm_readv`, `process_vm_writev`, `sigaltstack`, and `rt_sigreturn`.

To evaluate our syscall filtering, we need a set of representative filter rules. Thus, similarly to Connor et al. [Con+20], we examined and categorized each syscall for x86-64

[7]For more information about the syscalls mentioned in this section, we refer to the Linux manual pages section 2 [Ker20e].

in Linux 5.4. As a starting point for our analysis, we used information found in the source code of the `strace` utility [Str19]. For a more in-depth analysis, we inspected the documentation in Linux manual pages [Ker20e]. If manuals did not include the needed information, we tested the syscall's behavior on a reference machine running Linux 5.4.

Based on our analysis, we provide two filter modes with comprehensive sets of syscall filter rules. One mode filters syscalls to provide *self-protection* for the whole in-process isolation scheme. The other mode contains filters for a *local storage sandbox*. The filters isolate domains by limiting file system access only to a per-domain directory. Additionally, they permit domains only to use file descriptors they have opened before.

4.6.1 Analysis of the Syscall Interface

As syscall filtering for privileged binaries is out of scope for this thesis, we only analyze syscalls that do not need any elevated privileges to run. We categorize all unprivileged syscalls into the categories process- and thread-creation syscalls, process- and thread-directed syscalls, debugging syscalls, memory management-related syscalls, path syscalls, and file descriptor syscalls.

Process/Thread-creation Syscalls. A process (parent) can create a copy of itself (child) by executing the `fork`, `vfork`, or `clone` syscalls. This copy includes the whole page table and most other kernel resources. Like in other multi-threaded applications, the `fork` syscall can lead to synchronization issues in a PKU system. E.g., before the fork, another thread in the parent process might be editing data while holding a lock. In the child process, the thread does not exist any longer, and thus the data is in an inconsistent state [Pie09]. Furthermore, if the thread in the child process wants to acquire a lock another thread holds in the parent process, this results in a deadlock [Bal14].

`exec` replaces the currently loaded executable and the process's whole address space. Since we do not have any control over the new executable, we cannot automatically initialize our in-process isolation mechanisms for the child process. Note that syscall interception mechanisms based on Seccomp could survive an `execve`. However, any tracer thread located in the same process would not, allowing a domain to execute syscalls it would not have been able to.

Thread creation with `clone` is compatible with our in-process isolation. However, we still filter the `clone` syscall for untrusted domains, as we need to initialize Donky metadata inside thread-local storage.

To summarize, we block process creation syscalls `fork`, `vfork`, `clone`, `clone3`, `execve`, and `execveat`.

Process/Thread Syscalls. Donky uses thread-local storage to store metadata about each thread. This information should only be accessible to the monitor. Thus, it is protected by a memory protection key. However, an attacker can still change the thread pointer (`FS` register on x86-64). As the register is privileged, it can only be changed via the syscalls `arch_prctl` and `set_thread_area`.

The `prctl` syscall allows an application to control various aspects of its execution. E.g., it allows registering Seccomp filters (`PR_SET_SECCOMP`) or configures if a process can be debugged or not (`PR_SET_DUMPABLE`). As noted by Connor et al., allowing a domain to register custom Seccomp filters is dangerous. A Seccomp filter could let the trusted monitor believe a syscall executed successfully, while it did not. Analyzing all other operations of `prctl` is not in the scope of this thesis. Additionally to `prctl`, we also block the dedicated `seccomp` syscall.

As identified by Connor et al. [Con+20], handling of asynchronous signals cannot be implemented securely in a PKU-based sandbox. There are multiple problems with asynchronous signal handling.

1. The kernel can send a signal to an application at any time. If allowing a domain to register signal handlers, the handler's code might run in a different domain (the one that is active when the signal is delivered). Thus, the signal handler could run arbitrary code in the context of another domain. We block the registration of signal handlers by denying the `rt_sigaction` syscall.

2. Protecting the signal stack with memory protection keys is not feasible, as the kernel would kill an application if it cannot write the signal stack based on the currently set PKRU permissions. As the signal stack contains the original register state (including the PKRU register), we cannot ensure that not another thread changes its contents.

3. Connor et al. deem the `rt_sigreturn` syscall especially problematic, as it allows to restore an arbitrary register state at any time. Thus, we block the `rt_sigreturn` syscall.

4. Another problem arises from syscalls that trigger signals in the future (e.g., `alarm`, `setitimer`, `timer_create`) or for other threads (i.e., `kill`, `tgkill`, `rt_sigqueueinfo`, `rt_tgsigqueueinfo`). As we deny registration of signal handlers, these syscalls would allow an attacker to kill a thread while it is in another domain. We block the above mentioned syscalls. Other problematic syscalls affecting signal handling are `sigaltstack` and `rt_sigprocmask`.

Additionally, some threading syscalls can potentially be used for a denial-of-service attack. E.g., `sched_yield` yields execution of the current thread, `nanosleep` pauses execution for a specified amount of time, `pause`, `rt_sigtimedwait`, and `rt_sigsuspend` stop execution until the thread receives a signal, `wait` stops execution until a child thread receives a signal, `exit` terminates the thread, and `exit_group` terminates the whole application. As a domain can also cause a denial-of-service by executing an endless loop, we do not block any of these syscalls by default.

Debugging Syscalls. As correctly identified in [Con+20], debugging syscalls are very dangerous for in-process isolation. For example, `ptrace` can be used to debug another process. When attached, a debugger has full access to the process and can read and write arbitrary memory. As we have shown (see Section 4.5), `ptrace` can also trace the same process, thus having access to the same address space. When doing so, `ptrace` does not check protection keys for any memory access in the tracee.

The `process_vm_readv` and `process_vm_writev` syscalls can be used for reading and writing memory of a child process. However, one can also read or write memory of the same process issuing the syscall. The Linux kernel also does not respect protection keys for these syscalls.

We deny all untrusted domains to use above mentioned debugging syscalls.

Memory Management Syscalls. Protection keys are associated with Page Table Entries (PTEs). Any syscall that modifies PTEs can potentially break PKU-based in-process isolation. Some syscalls, like `mprotect` or `pkey_mprotect` manipulate the protection key in the PTE directly, others indirectly. E.g., a call to `munmap` deletes a page mapping. When executing this syscall, the kernel knows that it can reuse pages part of the mapping for later calls to `mmap`. As the kernel does not wipe unmapped pages, an attacker could use `munmap` to leak information of an arbitrary domain.

File Descriptor Syscalls. File descriptors are handles to IO resources. They commonly refer to files in the file system or to network sockets. An application can get a file descriptor to a file by opening it (syscalls `open`, `creat`, `openat`, and `openat2`), to a network stream by calling `socket`. Other syscalls returning a file descriptor as an identifier for various resources are, e.g., `inotify_add_watch`, `signalfd`, `timerfd_create`, `eventfd`, `memfd_create`, and `userfaultfd`.

Some syscalls can execute operations on resources referred to by file descriptors. E.g., the syscalls `write`, `read`, `ioctl`, `fstat`, and `fchmod`. By filtering syscalls for the creation of file descriptors, one can also restrict access to these operations.

Opened file descriptors can be used in the whole process. However, assuming the Donky monitor does not use file descriptors on its own, file descriptor syscalls are inherently not dangerous. We can easily identify all file descriptor syscalls by looking for the `TD` flag inside the syscall table in the source of Strace [Str19].

Path Syscalls. Paths refer to file system resources. Some syscalls operate on paths directly[8] (e.g., `mkdir`, `access`, or `unlink`). The `*at` syscalls[9] are siblings of normal path syscalls (e.g., `mkdirat`, `faccessat`, `unlinkat`) and can be used to resolve paths relative to a directory (given as file descriptor). When supplying an absolute path, they fall back to the behavior of their sibling syscall.

Open-like syscalls (e.g., `open`, `creat`, `openat`, and `openat2`) expect a path and return a file descriptor referring to the given path. `name_to_handle_at` is an additional path syscall that returns a different kind of handle which later can be used in `open_by_handle_at` to open a file.

There are some inherently dangerous paths for a PKU system. These include the files found in the `procfs` [Ker20l] and paths to core dumps [Ker20b]. The *procfs* is a pseudo-filesystem that provides an interface to kernel data structures of a process. It is mounted at the path `/proc`. Each process has a directory inside the *procfs*. A process can access its own resources at the symbolic link `/proc/self`. By default, this path and all exposed resources are accessible for the whole process. The most problematic resource

[8]Strace [Str19] marks them with the `TF` flag.
[9]Strace [Str19] marks them by setting both `TD` and `TF` flags.

for in-process isolation is the pseudo-file `/proc/self/mem`. By reading or writing to this file, one can read or write arbitrary memory inside the current process. Similarly to the debugging syscalls `process_vm_readv` and `process_vm_writev`, protection keys are not enforced.

Furthermore, reading core dumps is also problematic, as they contain the content of the whole address space of the process after a crash. If a PKU system can `fork`[10] and then crash the child application, it could access restricted information.

As filtering paths in syscall is prone to TOCTOU race conditions, we explore the implications of syscall filters for path sanitization in more detail in the next section.

4.6.2 Self-protection Filters

This section presents our so-called self-protection filters. These filters prevent untrusted domains from reading or writing memory that they are not allowed to access. As a basis, we deny all syscalls we identified as dangerous in Section 4.6.1. Furthermore, we deny all syscalls that do not fit any category in Section 4.6.1.

However, we need more convoluted filters to resolve paths to restrict access to resources in the file system. We propose three filter modes for self-protection: *self-realpath*, *self-procfd*, and *self-prctl*. We register syscall filters for self-protection as monitor filters (see Figure 4.13a).

Self-realpath. For *self-realpath*, we filter all syscalls that could give us a file descriptor inside of *procfs*. Sensitive syscalls include all variants of the `open` syscall (i.e., `open`, `openat`, `openat2`, `name_to_handle_at`).

For these syscalls, we canonicalize the path argument with help of the Glibc function `realpath` [Ker20p]. Internally, this function recursively checks if the path is a symbolic link by calling the `stat` syscall. When encountering a symbolic link, it resolves the link with a call to the `readlink` syscall. For a resolved symbolic link, the function again starts to canonicalize the path starting from the first component. The function returns if it reaches the last component. If the path points into the *procfs*, we compare the canonicalized path with a whitelist. If the path is not included in the list, the filter returns an error.

Note, this approach is prone to a TOCTOU race condition. There is a race window after the kernel resolving the last component of the path and before executing the syscall in the kernel. In this timeframe, an attacker can change any component of the canonicalized path to a symbolic link pointing to a potentially restricted file. To defend against an attacker inside of the target application, we lock path canonicalization inside of Donky. We also filter syscalls that can change the path resolution (i.e., `symlink`, `chdir`). Inside of the filters, we acquire the same lock as for the canonicalization.

Note, defending against a colluding attacker outside the target application is not part of our threat model for the *self-realpath* filters. In the following paragraphs, we present two filter modes that overcome this limitation.

[10]Note that we do not allow an application to fork.

Self-procfd. Similarly to *self-realpath*, for *self-procfd* we also filter the syscalls `open`, `openat`, `openat2`, and `name_to_handle_at`. In contrast to *self-realpath*, filters for `self-procfd` sanitize the path not before but after the syscall. We use the *procfs* interface in `/proc/fd/` [Ker20l] to resolve the path of the opened file descriptor. All returned paths are already canonicalized. Thus, we can compare the path directly to our whitelist. For a denied path, we immediately close the file descriptor and return an error to the caller.

Note, an attacker in a different thread (and domain) can easily predict the number of the opened file descriptor. The attacker could use the file descriptor after it is opened and before it is closed. Therefore, for all open-like syscalls (i.e., all syscalls that return a file-descriptor), we register the file descriptor in the monitor. We filter file descriptor-based syscalls and only accept file descriptors already registered before. In contrast to *self-realpath*, a colluding attacker outside the target application cannot interfere with our syscall filtering.

Self-prctl. For *self-prctl*, we do not register any additional syscall filters at all. We use a kernel feature that disables core dumps, debugging, and therefore also access to sensitive files inside of `procfs`. An application can enable this feature by calling `prctl(PR_SET_DUMPABLE, SUID_DUMP_DISABLE)` [Ker20k]. The files inside of `procfs` are inaccessible for any parent process that acts as a debugger and for the application itself.

4.6.3 Extended Filters

We provide extended filters that can be used on top of our self-protection. Extended filters further isolate individual domains by restricting file system access to a per-domain local storage-like folder. By default, a domain can also use any file descriptor another domain has opened. Therefore, we additionally restrict each domain only to use file descriptors it has previously opened itself.

Internally, we register a filter for each syscall expecting a path as argument (e.g., `stat`, `lstat`, `access`, `open`, `mkdir`). Inside these filters, we rewrite all path arguments. For absolute paths, we prepend the root directory of the domain's local storage. For relative paths, we prepend a domain-specific working directory to them. We ensure that a relative path cannot escape the local storage.

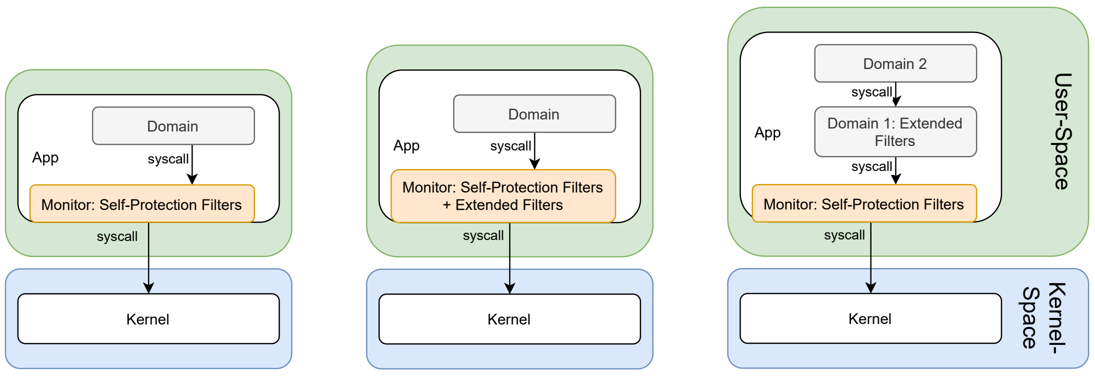
`*at`-syscalls (e.g., `openat`, `mkdirat`) are problematic for our extended filters. These syscalls expect a file descriptor to a directory as the first argument (`dirfd`). For absolute paths in the second argument (`path`), the `dirfd` is ignored, for relative paths, the path is resolved relative to this directory. Similarly to our *self-procfd* filters, we resolve the path pointed by `dirfd` using the interface provided by `/proc/fd/` [Ker20l]. Resolved paths are already canonicalized and are guaranteed to point inside the local storage. However, by moving the directory corresponding to the `dirfd`, the path for the file descriptor can change. E.g., a path previously pointing to the path `"/a/b"` inside of the local storage might change to the path `"/b"`. An attacker can escape the sandbox by specifying `"../../outside.txt"` for the `path` argument and winning the race condition.

We protect our filters from an attacker in a colluding thread by internally locking the resolution of relative paths. We also acquire the same lock for syscalls that can change paths (i.e., `rename`, `renameat`) or the working directory (i.e., `chdir`, and `fchdir`).

As symbolic links inside of any local storage could break our isolation, we disallow the creation of them. Note that our extended filters do not protect from a colluding attacker outside of the target application placing a symbolic link in any domain's local storage. We suggest using regular file system permissions to protect against this threat.

Extended filters can be registered as monitor filters (see Figure 4.13b), overriding existing self-protection filters regarding path resolution and file descriptors. As the *procfs* cannot be accessed when being restricted to a local storage directory, the original filters are not needed.

When using extended filters as domain filters (*extended-domain*, see Figure 4.13c), a parent domain can decide to constrain its child-domain(s) to a local storage sandbox. The parent domain can still access all files on the host files system unless the self-protection filters deny it. For our evaluations, we use the *extended-domain* filters in combination with *self-prctl*.


Domain
App
syscall
Monitor: Self-Protection Filters
syscall
Kernel
Domain
App
syscall
Monitor: Self-Protection Filters + Extended Filters
syscall
Kernel
Domain 2
syscall
App
Domain 1: Extended Filters
syscall
Monitor: Self-Protection Filters
syscall
Kernel
User-Space
Kernel-Space

(a) self-protection. (b) extended-monitor. (c) extended-domain.

Figure 4.13: Our filter modes for syscall filtering. (a) self-protection filters filter syscalls for all domains inside the monitor. (b) extended filters replace self-protection filters for path syscalls to implement a local storage sandbox in the monitor. (c) Domain 2 is a child domain of Domain 1. It is constrained to a local storage sandbox, while the monitor filters the syscalls of both Domain 1 and 2.

Chapter 5

Evaluation

We evaluate the performance of our syscall filtering by measuring the overhead for individual syscalls (micro-benchmarks) and for whole applications (macro-benchmarks). The micro-benchmarks give us an idea about the base overhead for each interception mechanism. The macro-benchmarks explore how this overhead affects the performance in real-world applications.

We compare our implemented syscall interception mechanisms (see Section 4.5) Ptrace, Ptrace+Seccomp, Ptrace-Delegation, Seccomp User Trap, our Kernel-Module-, and Indirect-Jump-Delegation.

We benchmark our different filter modes (see Section 4.6) for each mechanism. For self protection, our filter modes are *self-prctl*, *self-realpath*, and *self-procfd*. For the extended filters, we evaluate the performance for when they are registered as monitor filters (*extended-monitor*) or as domain filters (*extended-domain*). For evaluating extended filters, we additionally apply the filters from *self-prctl* for self-protection.

Evaluation Environment. Our evaluation system features an Intel Xeon Silver 4208 CPU, which supports Intel MPK. The used operating system is Ubuntu 20.04. This version uses the 5.4.0 long-term support Linux kernel. To reduce noise, we set the CPU frequency to a constant value of 1.8GHz for all cores. For interception mechanisms with tracer thread, we pin the tracer and tracee to distinct CPU cores. Doing so reduces noise from scheduling. For our file benchmarks, we further reduce noise by using a file system that is located in RAM (a so-called *tmpfs*) for the folder `/tmp`. We remove outliers outside the interval $[q_1 - k(q_3 - q_1); q_3 + k(q_3 - q_1)]$ with $k = 3$ [Tuk77]. q_i is the ith quartile of our data set. $q_3 - q_1$ is called the inter-quartile range. It contains 50% of the measurements around the median.

For all diagrams in this chapter, error bars indicate the standard deviation of our measurements.

5.1 Micro-benchmarks

For our micro-benchmarks, we measure the execution time of individual syscalls when filtered with our syscall filtering framework. We measure the time for 100 executions in a row to reduce the error in each measurement. We repeat this 10 000 times. Thus,

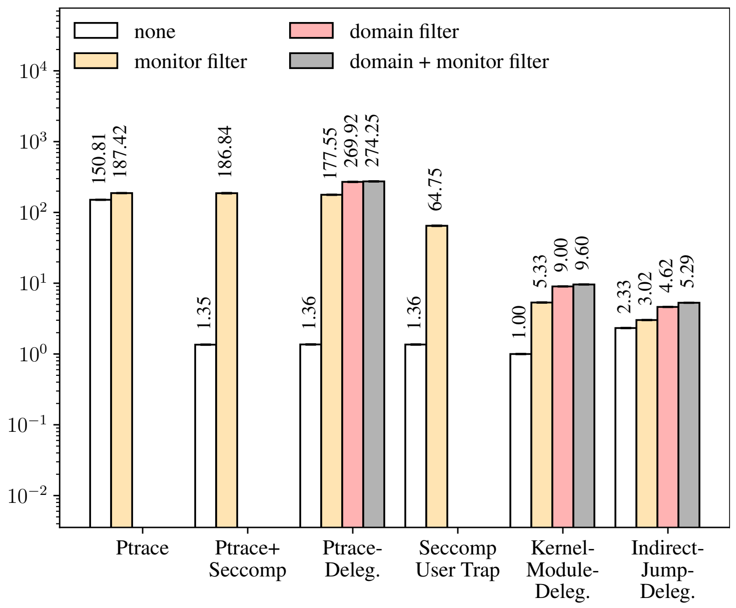


Figure 5.1: Relative execution time of a `getpid` syscall for different interception mechanisms and nesting layers. Numbers are relative to the unfiltered `getpid` syscall.

in total, we execute each syscall 1 000 000. We plot the execution time relative to the execution time of the respective unfiltered syscall.

We evaluate the syscalls `getpid` and `open`, as they show important aspects of our interception mechanisms (see Section 4.5), our nested filtering (see Section 4.4), and our proposed filter modes (see Section 4.6).

Base Overhead. The `getpid` syscall [Ker20c] returns the process ID of the process executing it. Thus, it executes very little code in the kernel. Timing it gives us an idea of the base overhead of executing a syscall. Generally, this is the overhead of issuing a `syscall` instruction in user-space, switching into the kernel, executing the respective syscall function in the kernel, and then returning back to user-space. For `getpid`, we can assume the execution time of the syscall function in the kernel is negligible.

Thus, by benchmarking the `getpid` syscall with all our mechanisms, we can calculate the base overhead for each mechanism. Additionally, for mechanisms that support nested filtering, we show the overhead for individual nesting layers. We measure the relative execution time with no filters attached, with just a monitor filter, only a domain filter, and both a monitor and a domain filter. For mechanisms that do not support nesting, we measure the relative execution time with no filters and monitor filters attached. As we do not want to benchmark any specific filtering code, the filter functions at various nesting layers are empty, thus allowing the syscall. Our results are shown in Figure 5.1.

The relative execution time of *unfiltered* syscalls is indicated by the white bars labeled with "none" in Figure 5.1. For Seccomp-based mechanisms, we measure a base overhead of 35–36%. This overhead accounts for the filtering performed in the kernel. As Seccomp reaches a filter decision by running a small BPF program, we can expect this overhead for all syscalls, not just filtered ones.

Similarly, our Indirect-Jump-Delegation also has a constant overhead for each syscall. This overhead is 133%. As the pure interception overhead for Indirect-Jump-Delegation is negligible (overhead of an indirect jump), this number reflects the time needed for domain switching. In the syscall handler, we switch from the untrusted domain to the monitor and then back to the original domain. As this is similar to an ordinary Donky call (Dcall), the overhead is also comparable. The relative execution time of a Dcall is 1.1x the `getpid` syscall.

Ptrace runs in a separate thread and needs multiple syscalls to filter a single syscall. Compared to other mechanisms, its overhead of 14981% is the highest.

Filter Overhead. When using monitor filters (yellow bars), Ptrace+Seccomp and Ptrace-Delegation perform similarly to Ptrace, as the syscall is not only filtered with the Seccomp but also in user-space. The mechanisms Ptrace and Ptrace+Seccomp use almost identical code for syscalls filtered in user-space. Thus, both have a relative execution time of approximately 187x. These mechanisms have an *enter*- and *exit-filter* in the tracer thread. They need to switch between tracer and tracee thread twice to execute these filters. As the Ptrace-Delegation mechanism only *intercepts* the syscall with Ptrace but filters it in the original thread, we can improve the execution time (177.6x). However, the overhead is still comparable to Ptrace, as Ptrace-Delegation intercepts a syscall in the tracee twice: once from the untrusted domain, the second time from the monitor that executes the allowed syscall for the domain.

For Indirect-Jump-Delegation, our relative execution time for a syscall filtered in a monitor filter is only 3.0x. Our Kernel-Module-Delegation has a relative execution time of 5.3x. Note, in contrast to the Ptrace-Delegation mechanism, for Kernel-Module- and Indirect-Jump-Delegation, we can disable delegation of syscalls while we are in monitor mode. Thus, syscalls allowed in monitor filters are only intercepted once, not twice.

Seccomp User Trap intercepts the syscall also only once. However, as it needs two thread switches, the relative execution time is much higher at 64.8x.

Nested Filter Overhead. We benchmark nested filtering for mechanisms that support delegation (Ptrace-, Kernel-Module-, and Indirect-Jump-Delegation).

For standalone domain filters (red bars in Figure 5.1), the overhead accounts for the domain switches needed for switching to the filtering domain and back to the monitor. For Kernel-Module-Delegation and Indirect-Delegation, we intercept the syscall twice: once in the original domain and once in the filtering domain. For Ptrace-Delegation, it is also intercepted a third time: for the monitor.

Again, we look at the Indirect-Jump-Delegation mechanism in more detail. In total, there are three switches from a domain to the monitor: for the original syscall, for the syscall inside the domain filter, and for returning from the domain filter. Thus, the overhead can be expected to be three times the execution time of a Donky call, i.e., $3 \cdot 1.1 = 3.3$x the `getpid` syscall. Our measured overhead is 3.62x. The difference of these values can be accounted for by the extra syscall handling code we need to execute when filtering syscalls.

For nested filters ("domain + monitor" filter in figure Figure 5.1), we also invoke filtering logic in the monitor in addition to filtering in a parent domain. As there is no further domain switch needed, the overhead for domain+monitor filtering is similar to just filtering with a domain filter.

Path Syscalls. We benchmark the `open` [Ker20g] syscall to evaluate the performance of our self-protection and extended filters. The `open` syscall is used for opening files in the file system. Thus, it is representative of other syscalls that expect a path as an argument. For our extended filters, all path-related syscalls have a similar overhead to the `open` syscall when supplied with the same path. For *self-realpath* and *self-procfd*, we only filter path syscalls that are open-like. Our results are shown in Figure 5.2.

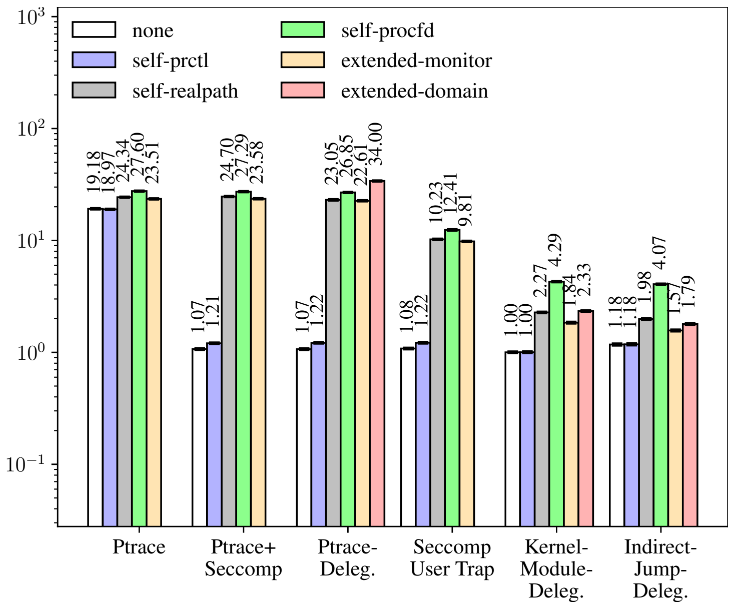


Figure 5.2: Relative execution time of the `open` syscall for different interception mechanisms and filter modes. Numbers are relative to the unfiltered syscall. The exact command is: `open("/tmp", O_RDONLY)`.

The *self-prctl* filters are fastest for self-protection, as they do not need to filter any additional path syscall. The BPF code for pre-filtering syscalls in Seccomp-based mechanisms adds overhead for every syscall. For open, this overhead is 21–22%.

self-realpath and *self-procfd* are both consistently slower, as they need to perform additional syscalls to sanitize the path in open-like syscalls. For the short paths used in the evaluation, *self-realpath* is faster. However, as *self-realpath* needs a syscall for each path component (e.g., the `/a/b` has two components: "a" and "b"), it gets slower when encountering longer paths. Additionally, it would resolve any encountered symbolic link and start traversing the path from the beginning.

We tested the code used inside of our syscall filters for *self-realpath* and *self-procfd* in isolation outside of our framework. The results are shown in Figure 5.3. For each path component, the `realpath` library function gets slower much faster than when resolving a path via the `/proc/fd` interface. Thus, for paths with more than four components, *self-procfd* would also be slower than *self-realpath* in Figure 5.3.

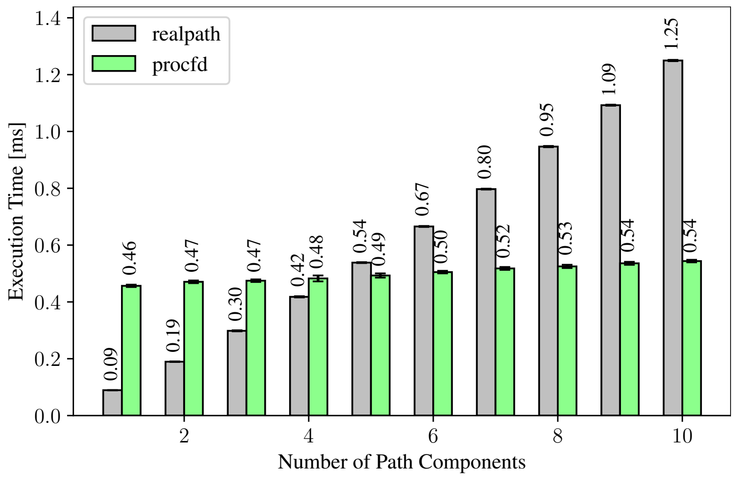


Figure 5.3: Effect of path length on performance. Comparing absolute execution times of syscalls used for sanitizing paths inside our syscall filters.

Extended filters do not need additional syscalls for the `open` syscall because they only prepend the directory of the local storage of the current domain to the specified path[1]. Thus, when combining extended filters with fast interception mechanisms, we achieve low overheads. For our Kernel-Module-Delegation, *extended-monitor* filters have 84%, *extended-domain* filters 133% overhead. For our Indirect-Jump-Delegation, the respective overheads for these filter modes are 57% and 79%.

5.2 Macro-benchmarks

Our macro-benchmarks consist of a wide range of compute- and IO-heavy applications. Compute-heavy benchmarks include signing the contents of a file with `openssl` (see Figure 5.4f) and using `zip` to compress the contents of a folder (see Figure 5.4d). IO-heavy benchmarks are `ls` (see Figure 5.4a), `grep` doing a full-text search in a folder (see

[1]Note, for relative paths in `*at`-syscalls, we still need an additional `readlink` syscall to resolve the path of the file descriptor of the directory the path should be relative to. If the special file descriptor `AT_FDCWD` is specified, we do not need any extra syscalls, as it represents the current working directory.

Figure 5.4b), `dd` filling a file with zeros (see Figure 5.4c), `git status` (see Figure 5.4e), and `sqlite3` inserting 100 entries into a database (see Figure 5.4g). Note that there are no benchmark results for `git status` with Seccomp User Trap because our implementation for this mechanism does not support multi-threading.

For benchmarks targeting the file system, we use a predefined commit version of the Git repository found at `https://github.com/git/git.git`. We clone this repository with a commit depth of 10 000[2].

All evaluated applications are *unmodified.* We initialize our syscall filtering by preloading the target application with our framework as library. I.e., we start the application with the `LD_PRELOAD` environment variable set to a shared library containing our framework. We initialize Donky and the specified syscall filtering mechanism inside our framework's library constructor. Depending on the filtering mode, the target application starts in the root domain or, in the case of nested filtering, another untrusted domain. The syscall filtering mechanism is configured to intercept the syscalls of the target application and delegate them to the Donky monitor.

For benchmarking, we start measuring the execution time of the target application after our framework is completely initialized. We stop our timer in the destructor after the application finished executing. Thus, the variable initialization overhead of each interception mechanism is not included in figures 5.4a to 5.4g. We discuss the initialization overhead separately in Section 5.3. We repeat all our benchmarks 1000 times.

Base Overhead. As already discussed in the micro-benchmarks (see Section 5.1), each interception mechanism (except Kernel-Module-Delegation) also has overhead for unfiltered syscalls. We can measure this base overhead also in our macro-benchmarks. It is illustrated in figures 5.4a to 5.4g by white bars.

As Ptrace (without pre-filtering) always needs to intercept all syscalls, its base overhead is consistently the biggest for all evaluated applications. Indirect-Jump-Delegation also intercepts all syscalls. However, as the interception is much faster than Ptrace, the base overhead for our macro-benchmarks is similar to the one of Seccomp-based mechanisms that pre-filter syscalls directly in the kernel. I.e., the base overhead for Indirect-Jump-Delegation is 0–42%, for Seccomp-based mechanisms 4–56% (depending on syscall frequency in the target application). There is no base overhead for Kernel-Module-Delegation, as no syscall is intercepted.

Self-protection. For self-protection (see Section 4.6.2), only a small subset of syscalls needs to be filtered. As Ptrace without pre-filtering always needs to intercept all syscalls, its overhead is substantially higher than for all other mechanisms.

The *self-prctl* filter mode is consistently the fastest form of self-protection for our framework. As most self-protection filters are simple allow/deny decisions and *self-prctl* does not filter any path syscalls, the overhead is comparable to the base overhead. For

[2]The full command to fetch the exact directory structure is: `git clone --branch v2.30.0-rc0 --depth 10000 https://github.com/git/git.git /tmp/git`

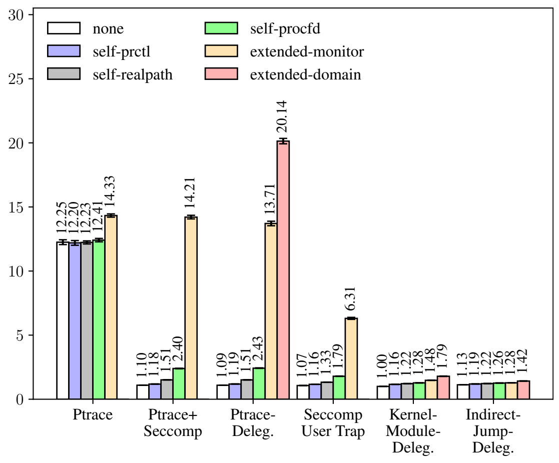
most evaluated applications (except `git` and `sqlite3`) and interception mechanisms (except Ptrace), the overhead is 0%–30%.

Self-protection with the *self-realpath* filter mode additionally needs to filter open-like syscalls. However, for applications that do not open many files (e.g., `ls`, `dd`, `zip`, `openssl`), this form of self-protection is still very efficient. This is also true for Seccomp-based interception mechanisms (3–51% overhead).

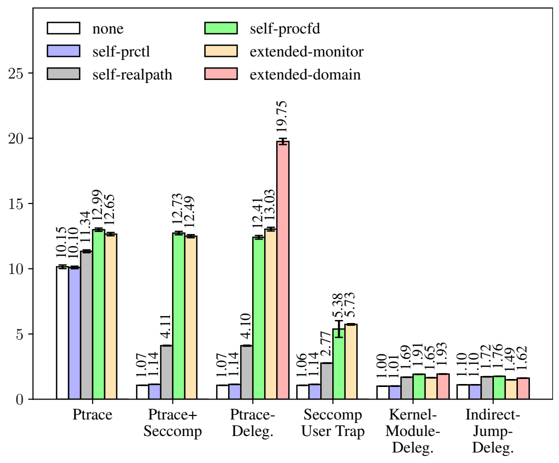
self-procfd also filters all syscalls that use file descriptors. Thus, the overhead for this filter mode is higher for all mechanisms. For Seccomp-based mechanisms, it is dramatically higher. I.e., for Ptrace+Seccomp the relative execution time for `ls` is 2.4x, for `grep` 12.7x, and for `dd` 26.3x. For Kernel-Module- and Indirect-Jump-Delegation, the performance is similar to the one with *self-realpath*, as the syscall filters for the file descriptor syscalls are very simple.

Extended Filters. Similarly to the *self-procfd* filter mode, the extended filters (see Section 4.6.3) intercept all syscalls that expect a file descriptor. Additionally, they also filter all syscalls that have a path as an argument. As this filter mode intercepts a big percentage of commonly used syscalls, extended filters are slow for both Ptrace and Seccomp-based mechanisms. However, as extended filters do not need an additional syscall to rewrite path arguments, they are quite efficient for fast interception mechanisms. For Indirect-Jump-Delegation with *extended-monitor* filters, the overhead is 1% for `openssl` and `zip`, 49% for `grep`, and 136% for `git status`. Note that for our `openssl` benchmark, the relative execution time for both *extended-monitor* filters and *extended-domain* filters is smaller than 1. This is because the local storage sandbox blocks all access outside the local storage directory. Thus, any syscall trying to access any global system configuration files e.g., for localization inside of `/etc`, would fail. As a result, subsequent syscalls might not be executed. However, denying access to these paths did not change the fundamental behavior of the benchmarked applications.

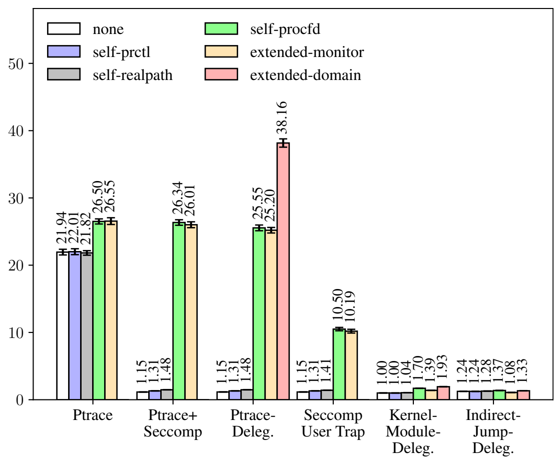
For IO-heavy benchmarks with Ptrace-Delegation, the execution time is about 1.5x higher if the extended filters are registered as domain filters instead of monitor filters. For the faster interception mechanisms, using the extended filters as domain filters is comparable to using them as monitor filters.



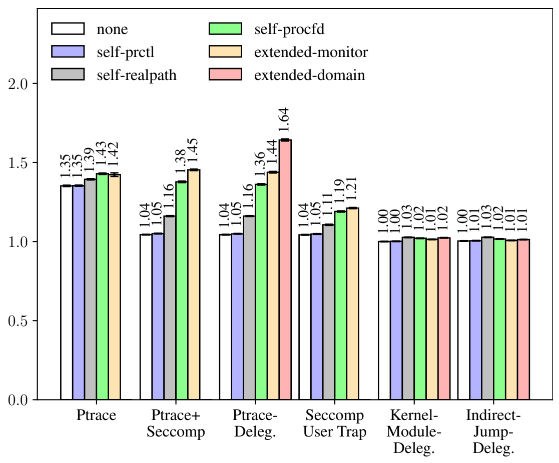
(a) Retrieving a directory listing: `ls -lah /tmp/git`.



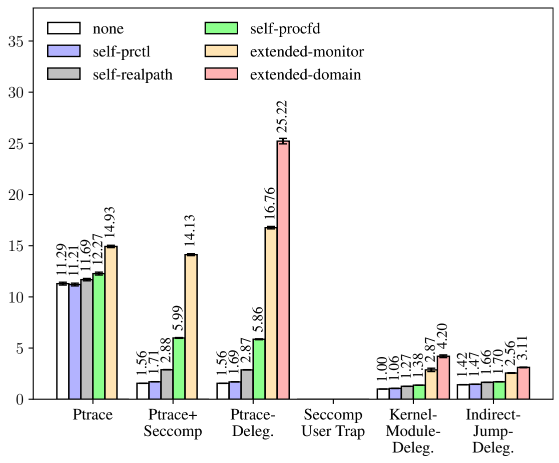
(b) Performing full-text search in a directory: `grep sshd -r /tmp/git/Documentation`.



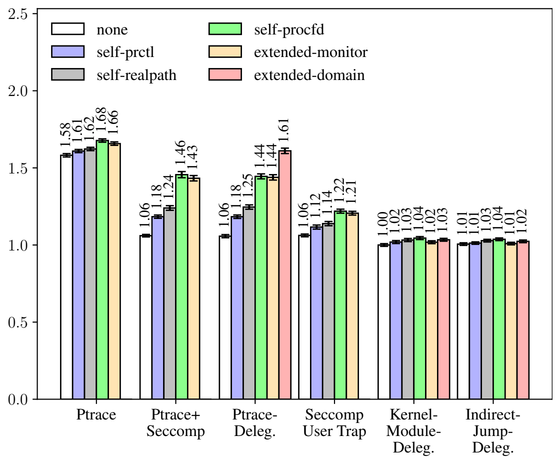
(c) Writing a file: `dd if=/dev/zero of=/tmp/file.bin bs=1024 count=1024`.



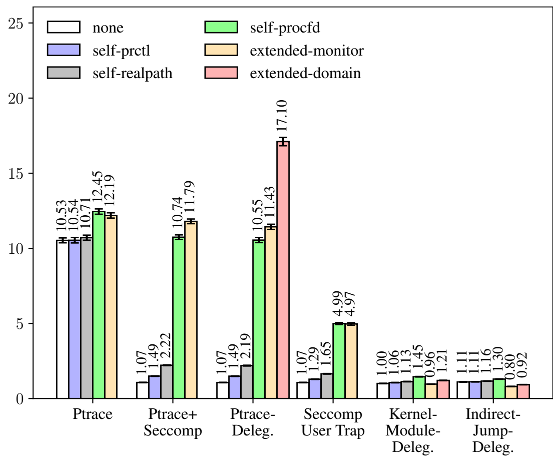
(d) Compressing a folder: `zip -r /tmp/gitweb.zip /tmp/git/gitweb`.



(e) Retrieving the status of a Git repository: `git -C /tmp/git status`.



(f) Signing a file with OpenSSL: `openssl dgst -sha256 -sign /tmp/key.pem -out /tmp/signature.bin /tmp/file.bin`.



(g) Inserting 100 rows into a database: `sqlite3 /tmp/db.sqlite '.read /tmp/insert100.sql'`.

Figure 5.4: Results of our application benchmarks. We plot the execution time of the protected application relative to that of the respective unfiltered application.

5.3 Initialization Overhead

The initialization overhead of our framework highly depends on the interception mechanism and the number of registered filters. Figure 5.5 shows various initialization overheads compared to the the native execution of `/bin/true` (0.29ms). We measure 0.64ms overhead for preloading the Donky framework. For Ptrace, there is an additional 0.37ms overhead for attaching the tracer. For Ptrace and Indirect-Jump-Delegation, the initialization overhead does not scale with the number of registered filters, as these mechanisms do not need to register filters explicitly. The initialization time for Seccomp-based filters also includes the initialization of a tracer thread. There is a small overhead depending on the number of syscall filters registered because of the verification of the Seccomp code in the kernel [Lina]. We use a single BPF program to specify allowed and denied syscalls. Similarly, our kernel module has overhead for registering the filters in the kernel for each filtered syscall. For Indirect-Jump-Delegation, we only measure the overhead for registering the syscall filters in the Donky monitor. This overhead exists for all mechanisms but is negligible. The filter modes *extended-monitor* and *extended-domain* filter the same number of syscalls. Yet, there is an additional overhead of 0.03–0.04ms for *extended-domain* filters, as our framework also creates an additional domain for the target application. Similarly to the extended filters, *self-procfd* also filters file descriptor syscalls. Thus the overhead is higher than *self-realpath*.

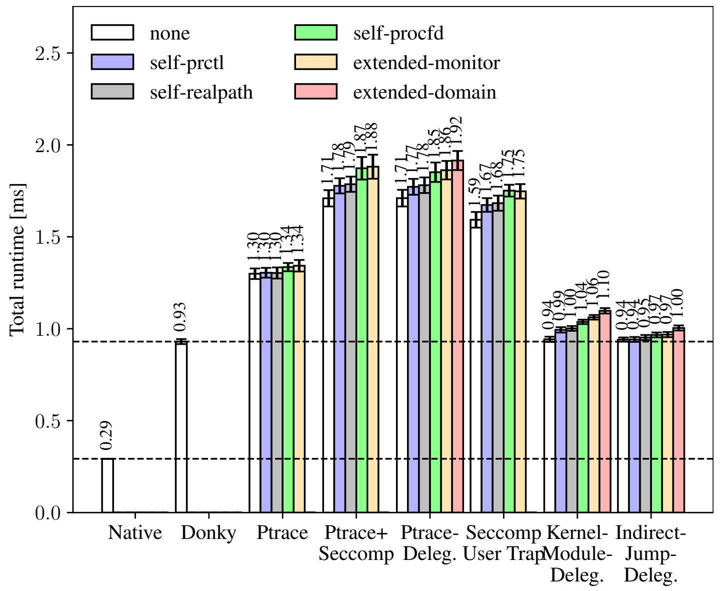


Figure 5.5: Initialization overhead of different interception mechanisms and filter modes. We measure the execution times for `/bin/true` without ("Native") and with ("Donky") our framework preloaded and for each interception mechanism and filter mode combination.

Chapter 6

Discussion

In this chapter, we look at software development aspects for creating syscall filters for our same-process syscall filtering framework (see Section 6.1). Furthermore, we explain how other PKU systems are compatible with our syscall filtering (see Section 6.2). In Section 6.3, we discuss future work to improve the effectiveness and efficiency of PKU sandboxes.

6.1 Software Development Aspects

Syscall filters might change the way syscalls operate in a non-intuitive manner. The semantics of a syscall might differ from those defined in the documentation. For applications relying on the semantics, this can have unforeseen consequences. In this section, we highlight some aspects of the development of syscall filters for our same-process syscall filtering framework.

Side-effects of Syscall Filters. As already discussed by Garfinkel in [Gar03], problems may arise when incorrectly mirroring operating system code or state inside of syscall filters. The state inside the filter might get out of sync with the state in the operating system. Thus, the effects of a syscall might be inconsistent with the ones described in its documentation. There are over 300 syscalls for x86-64 in Linux 5.4 [Str19]. For a developer of syscall filters to know each syscall that modifies or queries a specific piece of state in the kernel is unlikely. Garfinkel recommends avoiding replicating OS state. However, this might not be possible, as not all OS state is exposed to user-space.

When developing both the syscall filter and the program code in parallel, inconsistent syscalls can cause trouble. Every syscall can have a hidden state that the application developer might not consider. When encountering a bug in application code, it can be attributed both to application code and the syscall filter (or the kernel). When using a big library like the Glibc, one cannot be sure about which syscalls are called for a given library function. If a filtered syscall inside the Glibc does not behave as the Glibc expects it to do, this results in errors that are hard to track down.

To summarize, user-space filtering allows the developer to write complex filter code. However, writing good filter code is not as straightforward as it seems.

Same-process Filtering and Glibc. Similar to Donky, in our framework, all global state inside the Glibc is shared between application and syscall filtering code. As global variables inside the Glibc need to be accessible by all domains, they cannot be protected by any specific protection key. For Donky, this is problematic, as an attacker can leak information by, e.g., reading global buffers for file IO.

For our syscall filtering, it is also problematic for another aspect: The Glibc executes syscalls as part of their library functions. As the state in the Glibc might not be consistent at the point the syscall is executed, the syscall handler must only use reentrant Glibc functions (i.e., functions that can safely be called again at any point of their code). Similar restrictions are for code that runs in signal handlers. There is a list of so-called *async-signal-safe* library functions in the Linux manuals [Ker20r]. Note that this list includes not only reentrant functions but also ones that can suppress signal handling. However, if a function is not on this list, one can be sure it is not reentrant and not suitable for executing in a syscall filter.

Additionally to the reentrant Glibc functions, a syscall filter can also safely execute any syscall. However, for nested filtering, one needs to ensure that syscall filters of a parent domain do not use the same state as filters of child domains. A filter function can execute a syscall at any point in the filter code. When the syscall is intercepted and filtered in the parent domain, the shared state might be inconsistent.

Thus, we recommend that code for Donky domains, for the Donky monitor, and code for syscall filters should not use any Glibc code that relies on any global state.

6.2 Support for Other PKU System

In this section, we discuss the compatibility of our syscall filtering with the PKU systems ERIM [Vah+19] and Hodor [Hed+19]. We analyze which of our requirements (see Section 4.1) they fulfill.

ERIM. *ERIM* [Vah+19] (see Section 3.1.3) fulfills most requirements. There is an untrusted and trusted domain ($\mathcal{R}1$). Memory of these domains is protected by memory protection keys ($\mathcal{R}2$). One can only switch domain via trusted trampoline code ($\mathcal{R}3$). Binary scanning is used to disallow changing the PKRU without changing control flow to the trusted monitor. While not implemented, the authors propose using a secure stack when switching to the trusted monitor ($\mathcal{R}4$).

Additionally to our Donky self-protection filters (see Section 4.6.2), ERIM needs to ensure that it cannot create executable memory mappings containing a `WRPKRU` instruction. Connor et al. give a comprehensive overview of these additional filters [Con+20].

Hodor. *Hodor* [Hed+19] (see Section 3.1.2) fulfills our requirements. Hodor supports multiple domains ($\mathcal{R}1$). Memory for domains is protected by memory protection keys ($\mathcal{R}2$). In contrast to Donky, Hodor does not have a trusted monitor. Nevertheless, we can set up an “ordinary” domain to have access to all memory (PKRU = 0). We can configure our interception mechanisms to switch to this domain to filter syscalls. Hodor uses trampoline code for switching between domains ($\mathcal{R}3$). The `WRPKRU` instruction cannot be used outside the trampoline (see Section 3.1.2). Each domain and thread combination has a dedicated stack, thus fulfilling $\mathcal{R}4$.

The authors modified the kernel to allow memory management syscalls like `mmap` and `mprotect` to only modify memory mappings of the current domain. With our framework, this policy could be implemented in syscall filters.

6.3 Future Work

In this section, we look at future work to improve syscall filtering for PKU sandboxes.

Kernel Support. PKU-based in-process isolation could benefit from more support in the kernel. For example, providing the PKRU register to Seccomp filters would make it much easier to distinguish if syscalls originate from the trusted monitor or an untrusted domain. For a PKRU system with only two domains, a simple policy could allow the trusted domain to make all syscalls and deny all syscalls for the untrusted domain.

A further improvement would be to support the more advanced eBPF filters in Seccomp (see [Lina]). These filters can have key/value stores in the kernel to introduce state into a filter. The `bpf` syscall allows to refer to a key/value store by a file descriptor and access it from user-space. With eBPF, one could implement syscall filters for Seccomp that are both complex (e.g., deep argument inspection) and stateful.

Furthermore, as more syscalls are added to the kernel, our syscall filters for self-protection might not be sufficient anymore. Syscall filters for PKU sandboxes would benefit from an officially maintained list of syscalls that could circumvent the isolation provided by memory protection keys. For the time being, we recommend using a whitelisting approach that denies new syscalls by default.

Openat2 for Extended-filters. `openat2` is a new syscall introduced in Linux 5.6 [Ker20i]. Similarly to `openat` [Ker20h], this syscall expects a file descriptor pointing to a directory (`dirfd`) and a path (`path`). Additionally, it allows controlling the process of opening files with advanced options. New flags allow changing the behavior of path resolution inside the `path` argument. For the `RESOLVE_IN_ROOT` flag, paths are resolved as if the directory pointed by the `dirfd` was the root directory. I.e., paths are confined to this directory and cannot escape it.

We can use this syscall to implement our extended filters. For each syscall expecting a path, we would first open the path with `openat2`, passing the local storage directory as `dirfd`. Then, we execute a sibling syscall of the original path syscall that expects a file descriptor instead of a path. For most path syscalls, such sibling syscall exists. E.g., the `stat` syscall has the `fstat` syscall, `truncate` has `ftruncate`. In the end of the filter, we `close` the opened file descriptor.

In contrast to our extended filters with path rewriting (see Section 4.6.3), extended filters with `openat2` do not have any assumption about an attacker outside the target application. However, as every path syscall (except open-like syscalls) needs two additional syscalls for filtering, implementing a local storage with `openat2` is not practical. As our path rewriting does not require any additional syscalls for filtering path syscalls, it is much more efficient.

We cannot provide exact performance numbers, as the kernel used on our evaluation system did not yet implement the `openat2` syscall.

6.3.1 Interception Mechanisms using Signals

In this section, we discuss issues and potential solutions for signal handling for PKU sandboxes. Furthermore, we discuss two additional interception mechanisms that could benefit from secure signal handling.

Signal Handling. PKU sandboxes cannot support secure signal handling (see Section 4.6.1). The main problem is that we cannot protect the signal stack as the kernel might refuse to write to it when delivering a signal. A potential solution is to remove the check in the kernel. By only allowing the trusted monitor to register signal handlers, one could support per-domain signal handlers similarly to the ones described in the original Donky paper [Sch+20]. However, this change might not be compatible with the kernel's threat model of memory protection keys.

A solution without modification of the kernel might be to use Ptrace [Ker20o]. By design, the tracer is informed about every signal before the tracee receives it. We can load the appropriate memory protection keys for the protected signal stack (see Section 4.5.1) and delegate execution to a protected signal handler in the monitor (see Section 4.5.2). This signal handler further dispatches the signal to the handler in the correct domain.

Another approach for handling signals are potentially the syscalls `sigwaitinfo` [Ker20u] or `signalfd` [Ker20t]. Instead of handling signals asynchronously, these syscalls allow to wait on pending signals synchronously. A thread using these syscalls receives both signals directed to itself and process-directed signals. It cannot receive any signals directed to another thread.

Seccomp Trap. Similar to Seccomp User Trap (see Section 2.6.4), Seccomp Trap [Linb] filters syscalls in the kernel with a Seccomp filter. For syscalls that return the result `SECCOMP_RET_TRAP` the kernel generates a `SIGSYS` signal in the process that issued the syscall. If there is a signal handler registered for this signal, the handler can do syscall filtering. The signal handler can access and modify the whole register state. On x86-64, this state also includes the extended state, containing the PKRU register. The kernel restores the register state when returning from the signal handler.

Assuming that signal handling was secure, there is another problem with Seccomp Trap: As we cannot disable the delegation in the Seccomp filter, we cannot actually execute any delegated syscall. As Seccomp has access to the instruction pointer, it would be possible to allow syscalls originating from a specific code location (i.e., a syscall function in monitor code). However, as Intel MPK cannot protect memory from code execution, an untrusted domain can execute any syscall by jumping to the syscall instruction.

Syscall User Dispatch. Syscall User Dispatch [Linc] is a new syscall interception mechanism that dispatches syscalls into user-space. It is not intended as a security feature. It provides compatibility layers like Wine [Win] with an efficient way to intercept syscalls originating in certain memory areas. Furthermore, it quickly allows disabling syscall filtering by writing to a previously negotiated memory address. With secure signal handling, this mechanism can potentially be used for PKU sandboxing.

Chapter 7

Conclusion

In this thesis, we analyzed how to secure existing PKU systems with syscall filtering. As PKU systems are generally implemented purely in user-space, the kernel does not have any notion about the isolation structure inside a protected application. Consequently, there is no isolation for non-memory resources inside of an application. E.g., a file descriptor is accessible for all domains. As the kernel does not treat memory protection keys as a tool for in-process isolation, some syscalls can violate the isolation. E.g., a process can read any memory, irrespective of the protection key it is protected with, via the `process_vm_readv` syscall.

Similar to PKU being a user-space concept, we used user-space syscall filtering to further isolate memory- and non-memory resources of a process. Existing user-space syscall filtering systems need a second process to filter syscalls of the target application. We proposed same-process syscall filtering to limit the number of context switches required to filter syscalls. We isolate the memory for application and syscall filtering code with the help of memory protection keys. The application runs in an untrusted domain, the filter code in the trusted monitor. Inspired by the RISC-V user-mode syscall handling presented in the original Donky paper, we proposed a new form of *syscall delegation*. It allows us to run filter code not only in the same process but even in the same thread. Moreover, filter code can reference data and execute syscalls in the same way as the application code.

Based on syscall delegation, we proposed *nested syscall filtering* as a way for untrusted domains to filter the syscalls of their child domains. We allow a parent domain to *impersonate* syscalls for their child domains. Syscalls inside of syscall filters are again subject to filtering in the respective parent domain.

We repurposed the Ptrace, Ptrace+Seccomp, and Seccomp User Trap syscall interception mechanisms to provide syscall filtering in the same process. Besides these mechanisms, we proposed three new interception mechanisms that support syscall delegation and nested filtering. We call them Ptrace-, Indirect-Jump-, and Kernel-Module-Delegation. To emulate the RISC-V user-mode syscall handling, Indirect-Jump-Delegation replaces all syscall instructions inside the Glibc with indirect jumps to a syscall handler in the trusted monitor. Note, this mechanism on its own is not secure. However, combined with a secure interception mechanism that filters syscalls made outside the Glibc, this mechanism could be both fast and secure.

Our Ptrace-Delegation uses Ptrace to intercept syscalls. However, it does not filter the syscalls inside the tracer but *delegates* them into the trusted monitor of the target application. Additionally, we implemented a kernel module. For intercepted syscalls, the module does not execute any filter code in the kernel but delegates them back to user-space to a previously registered handler.

Based on our categorization of syscalls, we proposed two filter modes with distinct sets of syscall filter rules. One set contains syscall filters for self-protection of the PKU framework. The other one also isolates non-memory resources by confining each domain to its own local storage-like folder.

We evaluated the interception mechanisms and filter modes with micro- and application-benchmarks. Even for the fastest syscalls, like `getpid`, there is only a 2x overhead when using our novel mechanisms. The syscall filter for the open syscall is one of the slowest for our local storage sandbox. However, its overhead for our Indirect-Jump-Delegation is only 57%, for Kernel-Module-Delegation 84%.

In our application benchmarks, we just have 0–20% overhead for self-protection with most interception mechanisms and benchmarked applications. As the local storage sandbox needs to filter many syscalls, traditional interception mechanisms do not perform well (21–2555% overhead). Indirect-Jump-Delegation only has 0–156% overhead for our local storage sandbox.

To summarize, we showed that sandboxing with PKU can be both fast and secure by employing our novel efficient syscall filtering.

Abbreviations

BPF	Berkeley Packet Filter	17, 18, 21, 59
CISC	Complex Instruction Set Computer	6
eBPF	Extended Berkeley Packet Filter	18, 68
MPK	Memory Protection Keys	9, 10, 13, 24, 25, 26, 56, 69
OS	Operating System	3, 19, 29, 66
PID	Process ID	21, 42, 45
PKRU	Protection Key Register for User-space	9, 10, 11, 12, 32, 38, 42, 43, 44, 49, 51, 67, 68, 69
PKU	Protection Keys for User-space	9, 11, 14, 25, 31, 32, 37, 49, 50, 51, 52, 53, 66, 68, 69, 70
PTE	Page Table Entry	8, 9, 10, 14, 36, 52
SFI	Software-Fault Isolation	5, 6
TCB	Trusted Computing Base	28, 32
TLB	Translation Lookaside Buffer	5
TLS	Thread-Local Storage	12
TOCTOU	Time-Of-Check-Time-Of-Use	27, 28, 36, 53
TTLS	Trusted Thread-local Storage	12, 42, 43

Bibliography

[AD14] Thomas Anderson and Michael Dahlin. *Operating systems: principles and practice.* Vol. 2. Recursive books, 2014.

[Adv20] Advanced Micro Devices Inc. *AMD64 Architecture Programmer's Manual Volume 2: System Programming.* `https://www.amd.com/system/files/TechDocs/24593.pdf`. Oct. 2020. (Visited on 03/2021).

[AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java programming language.* Vol. 2. Addison-wesley Reading, 2000.

[AKS98] Albert Alexandrov, Paul Kmiec, and Klaus Schauser. *Consh: A confined execution environment for internet computations.* 1998.

[AR00] Anurag Acharya and Mandar Raje. "MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications". In: *USENIX Security Symposium.* 2000.

[ARMa] ARM Limited. *ARM Developer Suite Developer Guide.* `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0056d/BABBJAED.html`. (Visited on 03/2021).

[ARMb] ARM Limited. *Arm Morello Program.* `https://developer.arm.com/architectures/cpu-architecture/a-profile/morello`. (Visited on 03/2021).

[Bag+18] Mojtaba Bagherzadeh, Nafiseh Kahani, Cor-Paul Bezemer, Ahmed E. Hassan, Juergen Dingel, and James R. Cordy. "Analyzing a decade of Linux system calls". In: *ICSE.* 2018, p. 267. ISBN: 978-1-4503-5638-1.

[Bal14] Thorsten Ball. *Why threads can't fork.* `https://thorstenball.com/blog/2014/10/13/why-threads-cant-fork/`. 2014. (Visited on 03/2021).

[Bel+12] 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.* 2012, pp. 335–348. ISBN: 978-1-931971-96-6.

[Bit+08] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. "Wedge: Splitting Applications into Reduced-Privilege Compartments". In: *NSDI.* 2008, pp. 309–322. ISBN: 978-1-931971-58-4.

[Cas+09] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. "Fast byte-granularity software fault isolation". In: *SOSP.* 2009, pp. 45–58. ISBN: 978-1-60558-752-3.

[Che+16] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. "Shreds: Fine-Grained Execution Units with Private Memory". In: *S&P*. 2016, pp. 56–71. ISBN: 978-1-5090-0824-7.

[Cho16] Han Choongwoo. *Case Study of JavaScript Engine Vulnerabilities*. `https://github.com/tunz/js-vuln-db`. 2016. (Visited on 03/2021).

[Con+20] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. "PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems". In: *USENIX Security Symposium*. 2020, pp. 1409–1426. ISBN: 978-1-939133-17-5.

[Cor18] Jonathan Corbet. *Deferring seccomp decisions to user space*. `https://lwn.net/Articles/756233/`. June 2018. (Visited on 03/2021).

[Cor19] Jonathan Corbet. *The end of the 5.5 merge window*. `https://lwn.net/Articles/806576/`. Dec. 2019. (Visited on 03/2021).

[CVE14] CVEdetails.com. *CVE-2014-0160*. Available from MITRE, CVE-ID CVE-2014-0160. Apr. 2014. URL: `https://www.cvedetails.com/cve/CVE-2014-0160/` (visited on 03/2021).

[CVE18] CVEdetails.com. *CVE-2018-1000810*. Available from MITRE, CVE-ID CVE-2018-1000810. Oct. 2018. URL: `https://www.cvedetails.com/cve/CVE-2018-1000810` (visited on 03/2021).

[CVE19] CVEdetails.com. *CVE-2019-12083*. Available from MITRE, CVE-ID CVE-2019-12083. May 2019. URL: `https://www.cvedetails.com/cve/CVE-2019-12083` (visited on 03/2021).

[Doc15] Docker Inc. *Introducing runC: a lightweight universal container runtime*. `https://www.docker.com/blog/runc/`. July 2015. (Visited on 03/2021).

[Edg20] Jake Edge. *Seccomp and deep argument inspection*. `https://lwn.net/Articles/822256/`. June 2020. (Visited on 03/2021).

[Fab74] Robert S. Fabry. "Capability-Based Addressing". In: *Commun. ACM* 17 (1974), pp. 403–412.

[Fra+18] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. "IMIX: In-Process Memory Isolation EXtension". In: *USENIX Security Symposium*. 2018, pp. 83–97.

[Gar03] Tal Garfinkel. "Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools". In: *NDSS*. 2003. ISBN: 1-891562-16-9.

[Gol+96] Ian Goldberg, David A. Wagner, Randi Thomas, and Eric A. Brewer. "A Secure Environment for Untrusted Helper Applications". In: *USENIX Security Symposium*. 1996.

[GPR04] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. "Ostia: A Delegating Architecture for Secure System Call Interposition". In: *NDSS*. 2004. ISBN: 1-891562-18-5.

[Han] Dave Hansen. *PATCH 01/33 mm: introduce get_user_pages_remote()*. `https://lkml.org/lkml/2016/2/12/612`. (Visited on 03/2021).

[Har88] Norman Hardy. "The Confused Deputy (or why capabilities might have been invented)". In: *ACM SIGOPS Oper. Syst. Rev.* 22 (1988), pp. 36–38.

[Hed+19] 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*. 2019, pp. 489–504.

[Hew94] Hewlett Packard. *PA-RISC 1.1 architecture and instruction set reference manual, third edition.* 1994.

[Hon05] Zhang Hong-xin. "Implementation of Linuxthreads". In: *Computer and Modernization* (2005).

[IBM17] IBM Coorporation. *Power ISA version 3.0b.* 2017.

[IBM64] IBM Coorporation. *IBM System/360 Principles of Operation.* IBM Press, 1964.

[Int00] Intel Corporation. *Intel IA-64 architecture software developer's manual, revision 1.1.* 2000.

[Int19] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developers Manual.* 2019.

[Jon93] Michael B. Jones. "Interposition Agents: Transparently Interposing User Code at the System Interface". In: *SOSP*. 1993, pp. 80–93. ISBN: 0-89791-632-8.

[JS00] K. Jain and R. Sekar. "User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement". In: *NDSS*. 2000. ISBN: 1-891562-07-X.

[Ker20a] Michael Kerrisk. *clone(2) — Linux manual page.* `https://man7.org/linux/man-pages/man2/clone.2.html`. 2020. (Visited on 03/2021).

[Ker20b] Michael Kerrisk. *core(5) — Linux manual page.* `https://man7.org/linux/man-pages/man5/core.5.html`. 2020. (Visited on 03/2021).

[Ker20c] Michael Kerrisk. *getpid(2) — Linux manual page.* `https://man7.org/linux/man-pages/man2/getpid.2.html`. 2020. (Visited on 03/2021).

[Ker20d] Michael Kerrisk. *ioctl(2) — Linux manual page.* `https://man7.org/linux/man-pages/man2/ioctl.2.html`. 2020. (Visited on 03/2021).

[Ker20e] Michael Kerrisk. *Linux manual pages: section 2.* `https://man7.org/linux/man-pages/dir_section_2.html`. 2020. (Visited on 03/2021).

[Ker20f] Michael Kerrisk. *NAMESPACES(7) Linux Programmer's Manual.* `http://man7.org/linux/man-pages/man7/namespaces.7.html`. 2020. (Visited on 03/2021).

[Ker20g] Michael Kerrisk. *open(2) — Linux manual page*. `https://man7.org/linux/man-pages/man2/open.2.html`. 2020. (Visited on 03/2021).

[Ker20h] Michael Kerrisk. *openat(2) — Linux manual page*. `https://man7.org/linux/man-pages/man2/openat.2.html`. 2020. (Visited on 03/2021).

[Ker20i] Michael Kerrisk. *openat2(2) — Linux manual page*. `https://man7.org/linux/man-pages/man2/openat2.2.html`. 2020. (Visited on 03/2021).

[Ker20j] Michael Kerrisk. *pkeys(7) — Linux manual page*. `https://man7.org/linux/man-pages/man7/pkeys.7.html`. 2020. (Visited on 03/2021).

[Ker20k] Michael Kerrisk. *prctl(2) — Linux manual page*. `https://man7.org/linux/man-pages/man2/prctl.2.html`. 2020. (Visited on 03/2021).

[Ker20l] Michael Kerrisk. *proc(5) — Linux manual page*. `https://man7.org/linux/man-pages/man5/procfs.5.html`. 2020. (Visited on 03/2021).

[Ker20m] Michael Kerrisk. *process_vm_readv(2) — Linux manual page*. `https://man7.org/linux/man-pages/man2/process_vm_readv.2.html`. 2020. (Visited on 03/2021).

[Ker20n] Michael Kerrisk. *pthreads(7) — Linux manual page*. `https://man7.org/linux/man-pages/man7/pthreads.7.html`. 2020. (Visited on 03/2021).

[Ker20o] Michael Kerrisk. *ptrace(2) — Linux manual page*. `https://man7.org/linux/man-pages/man2/ptrace.2.html`. 2020. (Visited on 03/2021).

[Ker20p] Michael Kerrisk. *realpath(3) — Linux manual page*. `https://man7.org/linux/man-pages/man3/realpath.3.html`. 2020. (Visited on 03/2021).

[Ker20q] Michael Kerrisk. *seccomp(2) — Linux manual page*. `https://man7.org/linux/man-pages/man2/seccomp.2.html`. 2020. (Visited on 03/2021).

[Ker20r] Michael Kerrisk. *signal-safety(7) — Linux manual page*. `https://man7.org/linux/man-pages/man7/signal-safety.7.html`. 2020. (Visited on 03/2021).

[Ker20s] Michael Kerrisk. *signal(7) — Linux manual page*. `https://man7.org/linux/man-pages/man7/signal.7.html`. 2020. (Visited on 03/2021).

[Ker20t] Michael Kerrisk. *signalfd(2) — Linux manual page*. `https://man7.org/linux/man-pages/man2/signalfd.2.html`. 2020. (Visited on 03/2021).

[Ker20u] Michael Kerrisk. *sigwaitinfo(2) — Linux manual page*. `https://man7.org/linux/man-pages/man2/sigwaitinfo.2.html`. 2020. (Visited on 03/2021).

[Ker20v] Michael Kerrisk. *wait(2) — Linux manual page*. `https://man7.org/linux/man-pages/man2/waitpid.2.html`. 2020. (Visited on 03/2021).

[KN] Steve Klabnik and Carol Nichols. *Unsafe Rust*. `https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html`. (Visited on 03/2021).

[Kon+17] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. "No Need to Hide: Protecting Safe Regions on Commodity Hardware". In: *EUROSYS*. 2017, pp. 437–452. ISBN: 978-1-4503-4938-3.

[KZ13] Taesoo Kim and Nickolai Zeldovich. “Practical and Effective Sandboxing for Non-root Users”. In: *USENIX ATC.* 2013, pp. 139–144. ISBN: 978-1-931971-01-0.

[Lina] Linux Kernel Organization. *Linux Socket Filtering aka Berkeley Packet Filter (BPF).* `https://www.kernel.org/doc/html/v5.9/networking/filter.html`. (Visited on 03/2021).

[Linb] Linux Kernel Organization. *SECure COMPuting with filters.* `https://www.kernel.org/doc/html/v5.9/userspace-api/seccomp_filter.html`. (Visited on 03/2021).

[Linc] Linux Kernel Organization. *Syscall User Dispatch.* `https://www.kernel.org/doc/html/latest/admin-guide/syscall-user-dispatch.html`. (Visited on 03/2021).

[Lit+16] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. “Light-Weight Contexts: An OS Abstraction for Safety and Performance”. In: *OSDI.* 2016, pp. 49–64.

[Liu+15] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. “Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation”. In: *CCS.* 2015, pp. 1607–1619. ISBN: 978-1-4503-3832-5.

[LSK18] Hojoon Lee, Chihyun Song, and Brent ByungHoon Kang. “Lord of the x86 Rings: A Portable User Mode Privilege Separation Architecture on x86”. In: *CCS.* 2018, pp. 1441–1454. ISBN: 978-1-4503-5693-0.

[Mad+13] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David J. Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. “Unikernels: library operating systems for the cloud”. In: *ASPLOS.* 2013, pp. 461–472. ISBN: 978-1-4503-1870-9.

[Mer14] Dirk Merkel. “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux journal* 2014.239 (2014), p. 2.

[MJ93] Steven McCanne and Van Jacobson. “The BSD Packet Filter: A New Architecture for User-level Packet Capture”. In: *USENIX ATC.* 1993, pp. 259–270.

[Moz] Mozilla Corporation. *Security/Sandbox.* `https://wiki.mozilla.org/Security/Sandbox`. (Visited on 03/2021).

[MRD18] Lucian Mogosanu, Ashay Rane, and Nathan Dautenhahn. “MicroStache: A Lightweight Execution Context for In-Process Safe Region Isolation”. In: *RAID.* Vol. 11050. LNCS. 2018, pp. 359–379. ISBN: 978-3-030-00469-9.

[Pie09] Damian Pietras. *Threads and fork(): think twice before mixing them.* `http://www.linuxprogrammingblog.com/threads-and-fork-think-twice-before-using-them`. 2009. (Visited on 03/2021).

[Pro03] Niels Provos. “Improving Host Security with System Call Policies”. In: *USENIX Security Symposium.* 2003.

[RFB16] Inge Alexander Raknes, Bjørn Fjukstad, and Lars Ailo Bongo. “nsroot: Minimalist Process Isolation Tool Implemented With Linux Namespaces”. In: *CoRR* abs/1609.03750 (2016).

[RG09] Charles Reis and Steven D. Gribble. “Isolating web programs in modern browser architectures”. In: *EUROSYS*. 2009, pp. 219–232. ISBN: 978-1-60558-482-9.

[RIS] RISC-V International. *RISC-V: The Free and Open RISC Instruction Set Architecture*. `https://riscv.org/`. (Visited on 03/2021).

[RIS20] RISC-V International. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, document version 1.12-draft*. `https://github.com/riscv/riscv-isa-manual/releases/download/draft-20200212-c3d1f07/riscv-privileged.pdf`. 2020. (Visited on 03/2021).

[Sch+20] 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: *USENIX Security Symposium*. 2020, pp. 1677–1694. ISBN: 978-1-939133-17-5.

[Seh+10] 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*. 2010, pp. 1–12.

[SN19] Remo Schweizer and Stephan Neuhaus. “Downright: A Framework and Toolchain for Privilege Handling”. In: *Cybersecurity Development, SecDev*. 2019, pp. 76–88. ISBN: 978-1-5386-7289-1.

[Str] Strace Developers. *strace*. `https://strace.io/`. (Visited on 03/2021).

[Str19] Strace Developers. *strace - syscallent.h*. `https://github.com/strace/strace/blob/v5.4/linux/x86_64/syscallent.h`. 2019. (Visited on 03/2021).

[Sze+13] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. “SoK: Eternal War in Memory”. In: *S&P*. 2013, pp. 48–62. ISBN: 978-1-4673-6166-8.

[The] The FreeBSD Project. *The FreeBSD Project*. `https://www.freebsd.org/`. (Visited on 03/2021).

[The20a] The gVisor Authors. *gVisor*. `https://gvisor.dev/docs/`. 2020. (Visited on 03/2021).

[The20b] The Rust team. *Rust programming language*. `https://www.rust-lang.org/`. 2020. (Visited on 03/2021).

[Tuk77] John W. Tukey. *Exploratory data analysis*. Addison-Wesley series in behavioral science : quantitative methods. 1977.

[Uhl+05] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kägi, Felix H. Leung, and Larry Smith. "Intel Virtualization Technology". In: *Computer* 38 (2005), pp. 48–56.

[Vah+19] 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.* 2019, pp. 1221–1238.

[Vil+14] Lluís Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. "CODOMs: Protecting software with Code-centric memory Domains". In: *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014.* 2014, pp. 469–480.

[Wah+93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. "Efficient Software-Based Fault Isolation". In: *SOSP.* 1993, pp. 203–216. ISBN: 0-89791-632-8.

[Wan+20] Zhe Wang, Chenggang Wu, Mengyao Xie, Yinqian Zhang, Kangjie Lu, Xiaofeng Zhang, Yuanming Lai, Yan Kang, and Min Yang. "SEIMI: Efficient and Secure SMAP-Enabled Intra-process Memory Isolation". In: *S&P.* 2020, pp. 592–607.

[Wat+10] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. "Capsicum: Practical Capabilities for UNIX". In: *USENIX Security Symposium.* 2010, pp. 29–46.

[Wat+12] Robert NM Watson, Peter G Neumann, Jonathan Woodruff, Jonathan Anderson, Ross Anderson, Nirav Dave, Ben Laurie, Simon W Moore, Steven J Murdoch, Philip Paeps, et al. "CHERI: a research platform deconflating hardware virtualisation and protection". In: Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE). 2012.

[Wat+16] Robert N. M. Watson, Robert M. Norton, Jonathan Woodruff, Simon W. Moore, Peter G. Neumann, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Michael Roe, Nirav H. Dave, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Ed Maste, Steven J. Murdoch, Colin Rothwell, Stacey D. Son, and Munraj Vadera. "Fast Protection-Domain Crossing in the CHERI Capability-System Architecture". In: *IEEE Micro* 36 (2016), pp. 38–49.

[Win] Wine Developers. *WINE HQ.* `https://www.winehq.org/`. (Visited on 03/2021).

[Yee+09] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. "Native Client: A Sandbox for Portable, Untrusted x86 Native Code". In: *S&P.* 2009, pp. 79–93. ISBN: 978-0-7695-3633-0.

[You+19] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. “The True Cost of Containing: A gVisor Case Study”. In: *11th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2019, Renton, WA, USA, July 8, 2019*. 2019.

[Zho+14] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. “ARMlock: Hardware-based Fault Isolation for ARM”. In: *CCS*. 2014, pp. 558–569. ISBN: 978-1-4503-2957-6.