Domain Page-Table Isolation

Claudio Canella*, Andreas Kogler*, Lukas Giner*, Daniel Gruss* and Michael Schwarz†
*Graz University of Technology
Email: firstname.lastname@iaik.tugraz.at
†CISPA Helmholtz Center for Information Security
Email: michael.schwarz@cispa.saarland

***Abstract*—Modern applications often consist of different security domains that require isolation from each other. While several solutions exist, most of them rely on specialized hardware, hardware extensions, or require less-efficient software instrumentation of the application.**

In this paper, we propose Domain Page-Table Isolation (DPTI), a novel mechanism for hardware-enforced security domains that can be readily used on commodity off-the-shelf CPUs. DPTI uses two novel techniques for dynamic, time-limited changes to the memory isolation at security-critical points, called memory *freezing* and *stashing*. We demonstrate the versatility and efficacy of DPTI in two scenarios: First, DPTI *freezes* or *stashes* memory to support faster and more fine-grained syscall filtering than state-of-the-art seccomp-bpf. With the provided memory-safety guarantees, DPTI can even securely support deep argument filtering, such as string comparisons. Second, DPTI *freezes* or *stashes* memory to efficiently confine potentially untrusted SGX enclaves, outperforming existing solutions by 14.6 %-22 % while providing the same security guarantees. Our results show that DPTI is a viable mechanism to isolate domains within applications using only existing mechanisms available on modern CPUs, without relying on special hardware instructions or extensions.

I. Introduction

Memory isolation is a vital primitive to ensure the security of modern systems. However, this isolation is not only necessary between different applications, but it becomes more and more important within applications as well. Often, applications consist of multiple security domains that should be isolated from each other. Such in-process isolation ensures that vulnerabilities in one domain cannot easily affect a different domain. The separation of user and kernel space is a well-established isolation mechanism, trusted execution environments (TEEs) such as Intel SGX or ARM TrustZone are more recent examples. The interface and memory model of TEEs share similarities with the kernel-user-space boundary, where the enclave is considered trusted, but the rest of the application is not. Other new ISA extensions, such as Intel MPK [41], allow setting up multiple security domains within applications to efficiently control access permissions for different memory ranges. In all these cases, the isolation is configured in software and enforced by the hardware.

However, while the isolation must provide strong security guarantees, it is still necessary that the domains can communicate with each other. This interface between low-privilege and higher-privileged domains is often exploited. For instance, the syscall interface between unprivileged applications and the kernel is often used for privilege escalation attacks [86]. Hence, state-of-the-art isolation techniques such as seccomp-bpf [20] provide developers with the ability to block syscalls, reducing the kernel's attack surface in case of exploited applications. While this works well for blocking syscalls, the performance overhead can increase linearly with the length of the blocklist [36], [87]. Still, while securing this interface as well as possible is important, there is currently no option to create sophisticated filters. For example, it is not possible to block the `exec` syscall for all but a hard-coded set of applications. This lack of support for deep argument filtering is not merely a matter of implementation but stems from the current design of the filtering mechanism [21], [22].

Existing isolation solutions are often limited to very specific scenarios, require specialized hardware, or source- or binary-level instrumentation of the application. For example, the hardware isolation of TEEs can only be applied to enclaves. However, in practice, it is infeasible to run all software in enclaves. Enclaves have limitations, such as the inability to directly interact with the kernel, and are only available on a subset of CPUs. Their asymmetric trust model was exploited in recent work [81], [94], requiring mitigations with high overheads or special hardware features. Proposals for domain isolation require hardware extensions or modifications as well [76], [94], [26]. Moreover, hardware extensions such as Intel MPK are not broadly available, e.g., MPK is only available on Xeon Scalable CPUs and some 10th-generation Intel CPUs. Furthermore, an attacker can disable Intel MPK security domains after gaining code execution within the protected application, as preventing this is not in scope for Intel MPK [88]. Hence, while these proposed isolation mechanisms are effective, they *cannot be used on commodity CPUs*.

In this paper, we introduce the notion of dynamic, time-limited changes to the memory isolation at security-critical points, a security-domain mechanism we call **DPTI** (Domain Page-Table Isolation). As DPTI-based solutions rely only on the existing hardware-enforced memory protection of the memory-management unit (MMU), we can implement memory isolation for sandboxing and domain-isolation scenarios on commodity, off-the-shelf CPUs. Still, DPTI has the same page granularity as other approaches that require dedicated hardware or hardware extensions [88], [76], [94], [68]. Similar to KPTI [61], [34], DPTI uses efficient page-table modifications to create security domains that are temporarily inaccessible from another domain. However, in contrast to KPTI, DPTI does not necessarily have to maintain a second set of page tables for every process, and it has to solve additional challenges such as multiple mappings to one physical page. In the remainder of this work, we will use *DPTI* as shorthand for both the concept and its proof-of-concept implementations.

We evaluate DPTI on Intel CPUs ranging from 2015 (for SGX support) to 2018. However, as DPTI only relies on the MMU, it runs on all CPUs with virtual-memory support, in contrast to previous works [26], [65], [88], [76], [51], [35], [68]. Contrary to software fault isolation (SFI), we do not use any instrumentation of the application. With DPTI, it is even possible to add stricter syscall filters to existing (legacy) applications via a wrapper application.

We demonstrate the versatility of DPTI in two scenarios, enhanced syscall filtering and SGX protection domains, leading to high-performance, feature-rich solutions. In these, DPTI relies on two new techniques: memory *freezing* and *stashing*.

In our first application of DPTI, we present syscall filtering that is both faster and more fine-grained than seccomp-bpf. DPTI temporarily *freezes* or *stashes* the memory ranges of complex syscall parameters from the untrusted user-space application on a syscall, preventing modification while the kernel performs the syscall. While this intuitively sounds straightforward, it requires solving multiple challenges, including process creation and replacement, multithreading support, alias mappings, and alternative memory-access interfaces. However, by solving these challenges, DPTI-based filtering can evaluate complex parameters, such as strings, without introducing time-of-check-to-time-of-use (TOCTOU) vulnerabilities into the syscall interface [79], [22], [21]. Additionally, simple, non-argument-inspecting DPTI-based allow and reject filters only incur an overhead of 22 % compared to seccomp-bpf's 33.9 %. Especially filter-heavy applications benefit from our implementation since the overhead does not depend on the number of filters, as is the case for seccomp [36], [87].

In the second use case, DPTI improves performance and security of SGX enclave confinement [81], [94]. DPTI prevents the rewriting of host memory from SGX, again by *stashing* or *freezing* host memory, making it inaccessible to enclaves. However, this is not trivial: switching to and from an enclave requires some host pages. We address this challenge by securing SGX entries and exits with a special code bridge page. Thus, we mitigate host impersonation for arbitrary syscall execution by a malicious or hijacked SGX enclave [81]. While DPTI outperforms existing SGX enclave confinement solutions [94] by 14.6 %-22 %, the most important advantage is that DPTI works on commodity systems without hardware changes and maintains full compatibility with existing enclaves.

DPTI is an efficient and easily implementable solution for isolating security domains in various scenarios. Due to its software-only implementation without specific ISA-extension dependencies [26], [65], [88], [76], [51], [35], [68], it can be readily used on commodity, off-the-shelf hardware.

To summarize, we make the following contributions:

1) We introduce DPTI,[1] an MMU-based isolation mechanism inspired by page-table isolation, enabling fine-grained security domains and security policies.
2) We use DPTI to implement extended syscall filtering, enabling sophisticated argument-inspecting filter rules not supported by seccomp or AppArmor.
3) We show that DPTI can replace existing SGX enclave confinement solutions, isolating unmodified untrusted enclaves on commodity hardware.
4) We thoroughly evaluate the security and performance of DPTI and show that it provides higher performance and security than previous solutions in both case studies.

[1]Prototype can be found at https://github.com/domain-isolation/DPTI.

Outline. Section II discusses background. Section III describes the high-level idea and threat model. Section IV details the two case studies for DPTI. Section V evaluates security and performance of DPTI. Section VI discusses future and related work. We conclude in Section VII.

II. Background

This section introduces topics required for this work.

A. Sandboxing

Sandboxing provides an extra layer of security by strictly controlling the resources an application can access [70], [32]. In many cases, sandboxing is a last line of defense that assumes that the sandboxed application was already exploited. Thus, a sandbox should drastically limit the impact of an exploit. Existing sandboxes typically restrict access to the network and file system and limit the available syscalls. Sandboxes are widespread in browsers [96], [73], [95] and on mobile operating systems [40], [4]. Linux provides sandboxing via the SELinux [97] and AppArmor [5] frameworks.

B. Linux Seccomp

A vital part of sandboxes is the ability to restrict the syscall interface. The syscall interface provides functionality from the operating system to user-space applications. An exploit with unrestricted access to the syscall interface can read, write, and execute files on the system. In the worst case, the syscall interface itself is exploitable, leading to a privilege escalation [45], [44], [46]. Hence, sandboxes try to minimize the number of exposed syscalls to a minimum required for the application to work. Secure Computing (seccomp) [20] is a syscall filter that is integrated into the Linux kernel. An application can specify allowed syscalls, and the kernel will block all others. In addition to the syscall itself, seccomp can define filters for integer parameters. Filters based on the content of strings or structures are not supported, as seccomp cannot dereference parameters [21], [22].

Due to the complexity of limiting the syscall interface, several approaches to automatically generate syscall filters have been published recently [29], [10], [18]. The common approach is to dynamically or statically analyze an application to detect all required syscalls.

C. Runtime Attacks

Many security vulnerabilities are caused by memory safety violations. Typical memory safety violations, such as buffer overflows, enable attackers to modify an application in an unintended way [86]. In many cases, attackers try to overwrite code pointers to hijack the control flow. On modern systems, data is typically not executable, and thus an attacker cannot inject so-called shellcode into an exploited application [86]. As a result, exploits fall back to reusing existing program

parts and diverting the control flow to so-called *gadgets* [67]. Shacham [83] generalized such control-flow-hijacking attacks using gadgets as return-oriented programming (ROP). By chaining multiple gadgets, it is generally possible to build arbitrary exploits. In addition to such control-flow-hijacking that overwrite pointers [83], [11], [54], [31], [77], data-only attacks [74], [42] can also violate memory safety.

Race conditions are a type of vulnerability where a data structure is accessed in parallel, and the actual order of accesses affects the correctness of the program. A special type of race condition are time-of-check-to-time-of-use (TOCTOU) vulnerabilities. A TOCTOU bug exists if a memory location is accessed multiple times, and an attacker can manipulate the data between the access. If such a TOCTOU bug is exploitable, it is also called a double-fetch bug [92], [79]. Double-fetch bugs are especially dangerous in the syscall interface as they are hard to detect [92] and often relatively easy to exploit [79].

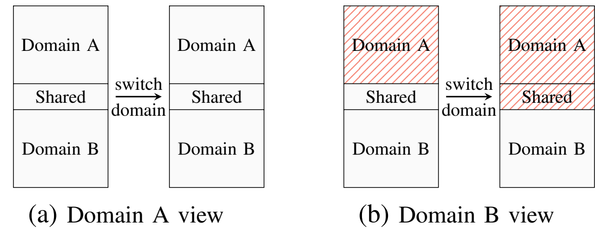
D. Memory Isolation

Memory isolation has been used to provide security for a long time. Segmentation and paging are two of the most well-known approaches to isolate memory on x86. The operating system configures the memory, and the CPU then enforces this configuration. While segmentation is no longer used to enforce access permissions on x86-64, security researchers continued to propose information hiding techniques based on this legacy feature [6], [53], [60]. Modern systems rely on paging to translate the virtual addresses of a process to physical addresses through the help of page tables. Page tables also contain access permissions which determine whether a page is read-only or user-space accessible.

Other memory isolation techniques use newer hardware features such as Extended Page Tables (EPT) [51], [59], Software Guard Extensions (SGX) [25], and Memory Protection Keys (MPK) [88], [76], [51], [35], [68]. EPTs facilitate memory virtualization, while SGX allows protecting code and data even from a compromised operating system. MPK introduces a new register containing a protection key and allows a developer to associate memory with such a key. The key itself is stored in the page tables, and an access is only allowed if the current register key matches the one stored in the page table.

E. SGX

Intel SGX [41] is a trusted execution environment (TEE) introduced with the Intel Skylake microarchitecture in 2015. Using this instruction-set extension, applications can be divided into an untrusted and a trusted part. The trusted part, the so-called enclave, is integrity- and confidentiality-protected by the CPU. Thus, even in the case of a malicious operating system, the content of the enclave is protected. Enclaves are hosted by ordinary, untrusted applications. Both enclave and untrusted application run in the same virtual address space. While the hardware prevents outside access to the address range of the enclave, the enclave can access all memory of the host application. The SGX threat model assumes that the entire software stack is malicious. However, there is no consideration that an enclave might be malicious. This asymmetry enables enclave malware that can impersonate the host application to execute arbitrary syscalls [81], [94]. Enclaves themselves cannot execute syscalls. Instead, enclaves communicate with the host application via ECALLs and OCALLs. After loading the enclave, the host application can call secure enclave functions using ECALLs via a call gate, similar to syscalls. If the enclave wants to use functionality from the operating system, such as a syscall, it has to use an OCALL to call into the host application. The ECALL/OCALL interface is defined by the developer at compile time.



(a) Domain A view (b) Domain B view

Fig. 1: The shared address space and the different views of it. No change for Domain A upon switching while parts are stashed or frozen (red pattern) for Domain B with DPTI.

III. High-Level Idea & Threat Model

On a high level, DPTI provides software-defined memory protection for more fine-grained sandboxing policies, such as deep argument filtering. This protection of memory regions can present itself in two different variants: read-only protection or entirely inaccessible. We refer to the former as DPTI-Freeze and to the latter as DPTI-Stash. Either variant guarantees that the memory cannot be modified by an untrusted domain.

Figure 1 shows the idea of DPTI. Two security domains share one virtual address space and use a dedicated memory region for communication, *i.e.*, for passing data across the security boundary. In many cases, while there is a defined memory region for the communication, Domain A has access to the entire Domain B. As both domains can execute code in parallel, Domain A cannot ensure the atomicity of the accessed data in this memory region. Hence, Domain B could modify the data while Domain A accesses it, potentially leading to data corruption or TOCTOU vulnerabilities. DPTI solves this problem by protecting the content of this memory region as long as Domain A accesses it. Any modification from Domain B is simply delayed until Domain A finishes its execution.

Such a scenario is common for operating systems, where the kernel (Domain A) is typically mapped into the upper half of virtual memory of every user application (Domain B). In Section IV-A, we present a case study with these two security domains. Another scenario is the execution of SGX enclaves, which share the address space with the host application. In Section IV-B, we present a case study in which DPTI ensures that a malicious or exploited enclave (Domain B) cannot modify data of the host application (Domain A).

A. Threat Model

For DPTI, we assume two different security domains in one application, such as user and kernel space, or untrusted application and trusted enclave. DPTI provides additional sandboxing to existing isolation mechanisms and hence assumes that the

used isolation mechanisms are reliable. In particular, DPTI presumes that the MMU-based isolation cannot be circumvented architecturally. Transient-execution attacks circumventing security domains do not undermine the security of DPTI. While Meltdown [57] showed that the US bit can be circumvented during transient execution, it only allows reading the page. Similarly, Foreshadow [89] circumvented the present bit for reading enclave memory. However, DPTI does not protect the confidentiality. Hence, Meltdown or Foreshadow do not cause a security problem. Similarly, while Spectre v1.1 [49], Store-to-Leak [78], and LVI [89] showed that values can be transiently written to inaccessible pages, applications can simply issue a serializing instruction before reading from the DPTI-isolated memory region to ensure that the architectural value is read. Fault attacks [48], [66], [47] are out of scope.

IV. Case Studies

In this section, we present two case studies showing that DPTI efficiently isolates different domains. First, we apply our method to facilitate efficient and complex syscall argument filtering that is not vulnerable to TOCTOU vulnerabilities (Section IV-A). Second, we apply DPTI to SGX, preventing attacks from untrusted or exploited enclaves [81] (Section IV-B).

A. Enhanced Syscall Filtering

To restrict syscalls, Linux provides developers with seccomp. While seccomp allows filtering static, primitive syscall arguments, it does not support complex arguments such as strings or structs due to TOCTOU vulnerabilities [21], [22]. Other approaches are based on syscall interposition, where the syscall is delegated to another process which decides whether a syscall is allowed [90], [23], [28], [30], [71]. Unfortunately, additionally to the cost of delegating the syscall, these approaches suffer from the same TOCTOU problem as seccomp [27], [93]. Hence, this is so far an unsolved problem.

We focus on deep argument filtering without introducing TOCTOU vulnerabilities while maintaining the performance of seccomp in typical scenarios. By applying DPTI, we remove an attacker's ability to modify the syscall argument while the kernel performs operations, such as checks, on it. Specifically, we prevent the attacker from rewriting a pointer or modifying a string while the kernel inspects that syscall argument. On a high level, we identify the page used by the syscall argument and make it read-only or fully inaccessible by modifying specific bits in the page-table entry. The argument is then compared to the allowed values for the respective syscall, and if the argument check succeeds, the syscall is executed. By design, this approach cannot suffer from TOCTOU vulnerabilities. While such an approach sounds straightforward, it requires solving several challenges such as process creation and replacement, multithreading, alias mappings, and other memory-access interfaces.

By providing the means for deep argument filtering, we can improve the security of the system by further limiting the post-exploitation impact of a memory safety vulnerability as we can better restrict syscalls such as *exec*. All other seccomp functionality, *i.e.*, filtering syscalls without argument checking and checking of static integer arguments, is naturally supported. We refer to this as *simple filtering*.

```
1 filter_info_t *filters = dpti_create_filters();
2 dpti_add_filter_rule(filters, SYS_read);
3 dpti_add_filter_rule_string(filters, SYS_write, 1, EQ, "
      teststring");
4 dpti_install_filters(filters);
```

Listing 1: Example of a simple allow/reject filter for *read* and a complex filter requiring deep argument filtering on the first argument of the *write* syscall using our support library.

1) Threat Model: Beyond the threat model of DPTI (cf. Section III-A), we assume that the application itself is not malicious but exploitable, e.g., due to a memory-safety violation, allowing an attacker to gain arbitrary code execution within the application. Similarly, the kernel is considered to be trusted but potentially exploitable. We assume that the post-exploitation phase targets the system and requires syscalls, e.g., to gain kernel privileges. Contrary to previous work [29], [10], [18], we can filter syscalls related to file operations as we can inspect complex data types. Our approach is orthogonal to other defenses such as CFI, ASLR, NX, or canary-based protections and improves security if these were circumvented.

2) Implementation: Our enhanced sandboxing consists of two parts: a kernel part that performs the actual task of filtering syscalls and the respective arguments, and a support library that can be linked to the application. The support library provides functionality for generating the individual syscall filters and installing them in our kernel module. As such, our support library is a similar high-level abstraction as *libseccomp* is for seccomp. However, filter generation in libseccomp is much more complex due to the usage of BPF. As our filters are not expressed in BPF, the setup is significantly easier. For our proof-of-concept implementation, we implement our filtering entirely as a standalone kernel module. The kernel module has the advantage that it can be used with any kernel version without recompiling the kernel. For a production-ready system, the filtering should be implemented directly in the kernel.

A prerequisite for all syscall-filtering approaches is to be able to intercept syscalls. Fortunately, entry points for syscalls are clearly defined, making it easy to intercept all syscalls. This includes both legacy 32-bit syscalls as well as 64-bit syscalls.

Setup. For the setup, the user-space application sends the filters to the kernel module. The filters are defined in a high-level representation similar to libseccomp, as illustrated in Listing 1. The module then performs a deep copy of the filters into kernel memory. Preventing TOCTOU vulnerabilities is not necessary at this point, as the application is still considered benign. Otherwise, if the application was already exploited, an attacker can either manipulate the filters or skip their installation entirely, rendering the filtering useless. This is in-line with other filtering approaches, such as seccomp-bpf. Once the filters have been copied, the application is considered sandboxed, and updates to the filters are no longer possible. A full implementation can consider allowing further restrictions of the filters, similar to seccomp.

Filtering Syscalls. Every requested syscall is delegated to our generic syscall function inside the module. If the syscall originates from a sandboxed application, the generic syscall function uses the syscall number to retrieve the syscall's filter rules. Contrary to seccomp, checking the filter for a specific

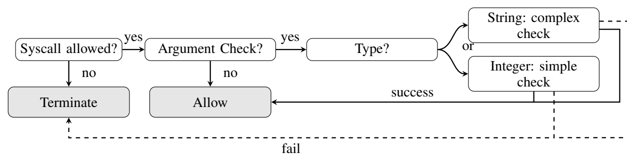


Fig. 2: Overview of the 3 DPTI syscall-filtering cases.

syscall does not require scanning all syscall filters. Instead, it is a simple array access; the lookup time for a filter does not depend on its position within the set of filters [36]. This reduces the asymptotic runtime from linear to constant.

We consider 3 different cases when checking a syscall, as illustrated in Figure 2:

(1) Syscall not allowed: The syscall fails the initial check whether it is allowed, making it unnecessary to check potential argument filters. If a syscall is executed that was not registered, the application can be terminated. Hence, we can perform an early out and kill the process.

(2) Syscall allowed, no argument filtering requested: The syscall passes the initial check, *i.e.*, the application explicitly allowed the syscall. This necessitates the check of potential argument filters. However, the syscall was allowed unconditionally as no filters are present. Therefore, DPTI calls the original syscall function with the same arguments. We refer to these first two cases as simple allow and reject filters.

(3) Syscall allowed, argument filtering requested: The syscall itself is allowed, making it necessary to check the installed argument filters. As the ABI [63] defines that syscalls can have up to 6 arguments, every argument must be checked against possible filters. We iterate over all potential arguments, checking for each whether a filter is defined. Checks then distinguish between primitive and complex data types. Primitive data types fully fit into the 64-bit register used as syscall argument. Such data types include, e.g., integers, booleans, flags, or single characters. Checking these primitive data types does not require any special handling as these arguments are already copied to the kernel. Hence, they are not vulnerable to TOCTOU and are similarly handled as they are in seccomp.

Complex data types cannot be contained fully in the 64-bit register and are typically pointers to either strings or structures. These pointers point to user-space memory and can thus be modified by a concurrently running thread during the syscall. If the modification happens between applying the filter and executing the syscall, the filter is effectively rendered useless. We focus on checking string arguments, as strings are widely used as syscall parameters. As string parameters are especially used with file names [79], they are an excellent target for sandboxing. Our method is not limited to strings but also applies to structures. For these types of arguments, we have to ensure that they cannot be modified between applying the filter and the execution of the syscall. Thus, we first resolve the page table mappings of the argument pointer. At this point, we can differentiate between our two variants of DPTI, providing different behavior and security guarantees.

DPTI-Stash: Our first variant relies on modifying the US bit, essentially making the page a kernel page and fully inaccessible to user space. It first attests whether the executable bit is not set in the resolved page-table entry as we do not want to bring user-controlled executable pages into the kernel due to security concerns. If this bit is set, our implementation kills the application. Otherwise, the US bit is cleared, *i.e.*, flushing the corresponding TLB entry from all necessary CPU cores. When an attacker tries to modify the argument in a different thread, the MMU enforces the protection by generating a page fault due to the privilege violation. Our module can now perform a simple string comparison with all developer-provided values without having to consider the possibility of a TOCTOU vulnerability. If the argument fulfills the syscall filter rule, the syscall is allowed, and the original syscall is executed. Before returning from the syscall, arguments are restored to user-space accessible if the physical page is currently not used in another, concurrently executed, syscall where string filters are used. Otherwise, the application is killed due to the violation.

DPTI-Freeze: The second variant relies on the RW bit instead of the US bit, making the page write-protected instead of fully inaccessible. By clearing it in the kernel data structures as well as in the page-table entry and flushing the page from the TLB, the page is read-only. A malicious thread can only read from it but no longer modify its content. Thus, TOCTOU vulnerabilities are no longer possible. The syscall filtering is equivalent to DPTI-Stash. After the syscall is done executing, we again set the previously cleared bits if they were set before the syscall. This requires some additional tracking of pages to determine whether this step is necessary. To prevent a concurrent thread from issuing an *mprotect* syscall to make the page writable again during the execution of another syscall, it is necessary to stall *mprotect* with write permissions on a page that had its write permissions cleared by DPTI-Freeze. The functionality for stalling is already available in the Linux kernel, as it is also necessary, e.g., for swapping pages. We can leverage the same functionality.

3) Special Cases: The approach outlined above works for most syscalls and can be handled by our generic syscall function. Only creating (*fork*, *clone*) and destroying (*exit*) processes need additional treatment as well replacing an existing process with *exec*. Linux creates a new task for a newly created process or thread, including a new process identifier. Hence, for the `fork` and `clone` syscall, we increase the reference counter of the filters and share them with the newly created process or thread. When calling `exec` after `fork`, we ensure that the copy-on-write syscall-parameter page is made accessible again for the other process(es) after the `exec` succeeds. In our proof of concept, we simply trigger a copy-on-write fault on this page before manipulating the page-table entry.

A second special case is the handling of the *exit* and *exit_group* syscalls, as they are responsible for cleaning up our filters. When a currently sandboxed application executes one such syscall, we decrease the reference counter of the filters by the number of threads in the group and mark the thread group as no longer sandboxed. If the reference counter reaches 0, *i.e.*, no thread or forked process needs the filters anymore, we free the memory before we terminate the process.

4) Multithreading: Multithreading needs special attention. DPTI works on a page-size granularity. Hence, it is likely that the string is not the only content of the associated memory page. While read accesses from a concurrent thread are no problem for DPTI-Freeze, write accesses to the page lead to a

page fault. For DPTI-Stash, both read and write accesses trigger a page fault due to the violation of the privilege boundary. These accesses might indicate a potential exploitation attempt. However, it is more likely that they are part of a legitimate access to data on the page that is unrelated to the syscall.

To handle such situations, we have to change the page-fault handler slightly. In the case of DPTI-Stash, the page-fault handler can easily determine that the access would normally be legal, as kernel pages are not found within the user-space address range. We experimentally verified this by scanning the pages of all running user-space processes on an Ubuntu 18.04 machine. If the page-fault handler determines that the faulting page is not an actual kernel page, it stalls the offending thread until the syscall has finished and the page is again available to user space. The kernel already needs to stall processes for swapping; the same functionality can be used in this case. We discuss the potential stalling times in more detail in Section V-A.

For DPTI-Freeze, we already track all pages that are modified for deep argument filtering, allowing us to easily differentiate a potentially legal write from one to a page that was never writable. If the page fault occurs on an access to such a tracked page, we again stall the offending thread until the original page access rights are restored. As stated before, read accesses cause no problem for this variant.

5) Alias Mappings: Although multiple *writable* mappings to the same physical address are rare (cf. Section V-A1), DPTI has to consider such scenarios as well. If an attacker has code execution, e.g., due to a memory safety vulnerability, the attacker can create two virtual address mappings to the same physical page, *i.e.*, by using *mmap* and other shared-memory-related syscalls. In such a case, the virtual address mapping used by the syscall is stashed or frozen, but the content of the physical page can be modified via the second, unaltered mapping. This once again facilitates a TOCTOU vulnerability. Hence, it is necessary that alias memory mappings are tracked, and if one such mapping is used during the argument check, all other mappings to the same page must be modified as well.

To cover all possible methods of creating an alias mapping, we incorporate our tracking via a probe on the page-fault handler. This allows us to track alias mappings that are allocated by pre-faulting the page, *i.e.*, *mmap* in combination with MAP_POPULATE, or a page fault upon the first access. For each physical page, we store all virtual addresses mapping this page, including meta information such as the permission and the process, independent of whether the mapping is in the same or a different process. This information is then used when checking a string argument during a syscall by iterating over the alias mappings of the currently used page and modifying the mapping as previously outlined. We discuss the impact on the performance as well as how frequently such mappings are used by real-world software in Section V-A1.

6) Modifying Memory through /proc/self/mem: Finally, an attacker can potentially circumvent our TOCTOU-free argument filtering via the */proc/self/mem* interface, which allows directly modifying the content of a physical page, ignoring missing write permissions. To protect against this possibility, it is necessary to either restrict the access to this file entirely, *i.e.*, make it read-only, or to the offset that corresponds to the physical page currently used in a syscall. In our proof-of-concept, we rely on the former by installing a probe on the *mem_write* function that prevents the write if the application is currently sandboxed. As the read-only approach is already done by REL 5 and 6, it is reasonable to assume that this does not impair the functionality of applications [62]. Moreover, to understand whether this interface is used in open-source software, we looked at all code on GitHub that opens this file for writing. After filtering out all PoCs for exploits, we only found one project (`rr`) that uses this functionality. Hence, while a production-ready implementation can implement more fine-grained control, it is questionable whether write access to this file is necessary at all.

Summary

We presented an alternative for seccomp that provides improved performance for simple filters due to less time-consuming checks. We additionally provide two variants for deep argument filtering without being vulnerable to TOCTOU vulnerabilities by efficiently and securly modifying page-table entries. The latter is currently not supported by seccomp; hence we further improved the systems' security.

B. SGX-Protection Domain

The classical SGX threat model is asymmetric, *i.e.*, it enforces strong protection guarantees for enclaves but does not protect the user application from the loaded enclave. Enclaves can lead to various security concerns, as neither the operating system nor the user application can verify the code executed inside the enclave, as intended by the design of SGX. One limitation of SGX is that it does not allow the enclave to execute code outside the enclave boundaries [14]. However, Schwarz et al. [81] showed malware inside enclaves can simply manipulate the user applications' stack to execute arbitrary code outside the enclave. To balance the protection guarantees for the user application, SGXJail [94] proposed to isolate enclaves by moving them to a separate process with strict syscall filters, or by relying on hardware modifications, *i.e.*, memory protection keys (Intel MPK).

Our DPTI-based SGX protection domain extends the memory isolation guarantees of SGXJail. However, with DPTI there is no increased overhead of process isolation, no need for MPKs, and no need for changing the SGX specification. We apply DPTI by replacing a thread's page-table mapping and isolate the user application's memory during the execution of an enclave. As in this case, most available pages, *i.e.*, the entire user-space application, have to be isolated, switching the mapping is faster than iterating over all mapped pages.

The isolated mapping only contains the original mappings for the loaded enclave, a single additional code page mapping, and a few data page mappings from the user application. The single code-page mapping is used to enter the enclave while the additional data-page mappings are used to transfer data from and to the enclave. Upon enclave exit, our protection verifies that the enclave did not tamper with the provided pages or registers and restores the non-isolated mapping for normal execution. All other pages are either protected by DPTI-Freeze or DPTI-Stash, depending on what security guarantees the user application enforces (cf. Section V-B).

In contrast to our proposed syscall filtering, the SGX protection domain implements DPTI-Stash by not mapping a page instead of clearing the US bit. Therefore, the enclave can no longer access any of the non-isolated pages as these pages are no longer mapped inside the thread's virtual address space. When using DPTI-Freeze, all of the non-isolated pages are mapped as read-only pages in the isolated mapping, allowing the enclave to read but not modify them.

To perform fast transitions, we construct the isolated mapping once and then reuse and update it upon entering. We redesign the data flow between the enclave and the user application to remove unnecessary copying of the data passed to an enclave, leading to a high-performance SGX protection domain with only an overhead of 9.9 % to 24.0 % for the worst case scenario (cf. Section V-B).

1) Threat Model: For the SGX protection domain, we reverse the classical SGX threat model and assume that the loaded enclave is untrusted and potentially malicious, whereas the app loading the enclave is trusted. We assume that the enclave tries to read or modify data of the user application, e.g., to mount a ROP attack [81]. The trusted user application does not use seccomp filtering to restrict syscalls. We assume that the code executed inside the enclave is unknown and imposes no restrictions on limiting the interfaces between the enclave and the user application in the sense of ECALLs and OCALLs.

2) Implementation: The SGX protection domain requires no changes to existing enclaves. All parts are implemented in the driver and the Untrusted RunTime System (URTS).

Isolated Mapping. For the isolated mapping, we extend the SGX driver, which provides the necessary information about pages associated with a given enclave. We create one isolated mapping per enclave. This design decision ensures that multiple threads within enclaves are supported while colluding enclaves cannot circumvent the isolation.

The enclave's isolated mapping is extended by a Code Bridge Page (CBP) and Data Bridge Pages (DBPs). The CBP contains the `EENTER` instruction used to enter enclaves inside the URTS. It is shared amongst all enclaves as the URTS library is only loaded once inside the user application. The Data Bridge Pages (DBPs) are used to transfer data between the enclave and the user application. To facilitate fast switches between two mappings, we only exchange the actual hardware mapping inside the `CR3` register and retain the VMA structures of the original mapping.

Entering and Exiting Isolation. Figure 3 depicts the control flow for switching protection domains. When a thread executes an IOCTL syscall into the SGX driver ①, the driver switches the calling thread's virtual address mappings to the isolated address mapping. DPTI ensures that the syscall instruction is the last instruction on the page before the CBP. Hence, the page containing the syscall is not part of the isolated mapping, as returning from the syscall does not require this page.

The enclave is entered and executed in isolation via the `EENTER` instruction ②. To exit an enclave, the `EEXIT` instruction is invoked from inside the enclave, with the RAX register set to 4 and the return address specified in RBX ③. While DPTI ensures that the only executable mapping is the CBP, an enclave can potentially still return to an arbitrary location

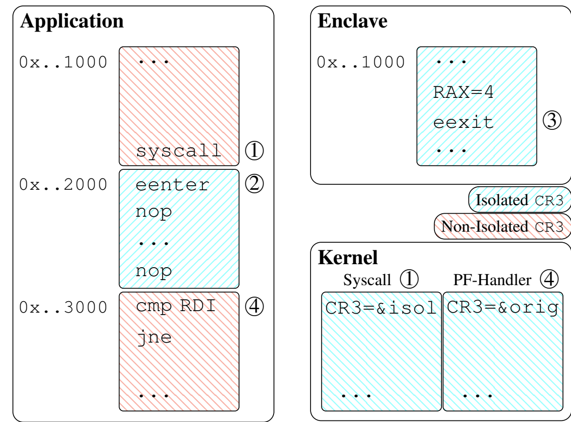


Fig. 3: Control-flow transition of entering and leaving DPTI-based SGX protection domain.

inside the CBP. Hence, the CBP must not contain exploitable code. As x86 relies on variable-length opcode and does not enforce instruction alignment, we opted for not placing any functional code on the CBP. Instead, we use the CBP as a trampoline by filling it with single-byte NOP instructions DPTI then leverages the page fault triggered by the first instruction after the CBP ④. The SGX driver catches the page fault and verifies that the fault originates from the first instruction after the CBP. If this is the case, the driver switches the address space back to the non-isolated mapping.

Interrupts. A special case is an asynchronous enclave exit due to an interrupt or fault. The CPU stores the enclave state, hands control to the interrupt or fault handler and then resumes the enclave using an `ENCLU` instruction. As this instruction has to be mapped, we place it on the CBP. We verified that a misaligned jump into this instruction cannot be exploited.

Data Bridges and Stack. DPTI must be aware of the pages used for data exchange between the enclave and host application, hence the dedicated DBPs. These DBPs are defined when setting up the isolation. Note that DPTI only works on page granularity. Therefore, the user must ensure that the data marked as DBPs is aligned and padded to the page boundary to ensure no additional data is exposed.

Before entering the isolation, DPTI aligns the stack pointer to a page boundary, allowing isolation of pages above the thread's stack pointer. The stack pages below the aligned stack pointer are accessible from within the isolation, as the SGX-SDK uses the user application's stack to pass arguments to OCALLs. However, these pages do not contain any data the application uses after the enclave returns. If already destroyed stack frames were used for sensitive data, they should be manually zeroed before entering the enclave. When returning to the user application, DPTI verifies that the stack and the base pointer match the values stored when entering the enclave during the ECALL, ensuring that the enclave did not alter the stack (cf. Section V-B).

Page Fault Handling. Page faults on EPC pages are allowed, as they can be lazily mapped. Page faults on host-application pages from isolated threads are only allowed on the first instruction of the page after the CBP, and the signal handler.

Otherwise, the enclave attempted to access data not marked as DBP or jumped to an isolated code page with the `EEXIT` instruction, leading to a termination of the enclave. Note that DPTI can distinguish a fault generated inside an enclave from a malicious attempt to execute the signal handler directly.

3) Optimizations: We propose an optimization that can be used if the application does not require the *stat* syscall, or if an untrusted application cannot abuse this syscall. Instead of switching from the SGX domain to the non-isolated domain using a relatively slow page fault (cf. Sections V-B1 and VI-A), we can rely on a syscall.

While placing a syscall instruction on the CBP could potentially lead to exploitation of this instruction, we utilize the nature of the `EEXIT` instruction. This instruction requires the `RAX` register to be set to 4. Hence, an attacker can only execute the *stat* syscall. This syscall typically does not allow any control over an application, nor does it allow to leak valuable information. To communicate with the driver, we hardcode the necessary register values before the syscall. Thus, the enclave can only execute the syscall communicating with the driver or the *stat* syscall by jumping directly to the syscall instruction.

Summary

We presented a solution to confine malicious enclaves, inverting the typical SGX threat model, in a secure way without relying on additional hardware modifications. Our solution can be implemented either via DPTI-Stash or DPTI-Freeze, each providing different security guarantees.

V. EVALUATION

In this section, we evaluate our enhanced syscall filtering (Section IV-A) and our SGX-protection domain (Section IV-B) in terms of security and performance.

A. Enhanced Syscall Filtering

We evaluate the performance (Section V-A1) and security (Section V-A2) of both variants of DPTI, clearly showing advantages and disadvantages of DPTI-Stash and DPTI-Freeze.

1) Performance Evaluation: In this section, we evaluate the performance of the two variants. First, we evaluate the performance when executing a single syscall with reject filters for other syscalls. Second, we analyze the overall performance on various real-world applications with simply allow/reject filters. In both cases, we compare the result to an unsandboxed version and one using seccomp. Hence, we only evaluate cases that are supported by seccomp. Third, we analyze the performance impact of our newly proposed method for deep argument filtering. Finally, we perform an in-depth analysis of the various steps discussed in Section IV-A, *i.e.*, resolving page tables, bit manipulation, TLB flush, and the string comparison, further substantiating the previous analysis results.

Setup. Unless stated otherwise, all experiments in this section are performed on an Intel Skylake i7-6600U running Ubuntu 18.04.1 with kernel version 5.4.0-72-generic at a stable frequency of 2.6 GHz. To ensure that unrelated mitigations do not affect our measurements, we disabled all mitigations for transient-execution attacks. For completeness, we verified the functionality of our approach in the presence of them, showing that they do not negatively affect DPTI. For our evaluation of seccomp, we rely on the state-of-the-art seccomp library *libseccomp* (2.5.1) to generate and install the respective filters. Additionally, to improve the performance of seccomp, we enabled the BPF JIT compiler.

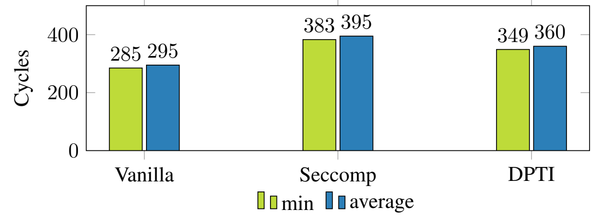


Fig. 4: Syscall latency with no, seccomp-, and DPTI-based filtering over 1 million iterations of the *getppid* syscall.

Simple Syscall-filtering Benchmark. We first microbenchmark the execution time when filtering a simple syscall (*getppid*). We compare the results to an unsandboxed (vanilla) application and one using seccomp. We choose *getppid* as it is a fast syscall without side effects used by previous work [10], [36] and by the kernel developers for their benchmarks [9]. As this experiment does not involve deep argument filtering, and hence no protection of memory contents, there is no difference between DPTI-Stash and DPTI-Freeze.

In the sandboxed versions, we allow 8 and block 341 syscalls using simple allow/reject filters. We measure the average execution time of the syscall in cycles over 1 000 000 executions. We show the results of this experiment in Figure 4.

The average execution of a vanilla *getppid* syscall requires 295 cycles and 395 cycles (+33.9 %) with seccomp active. With DPTI, the average execution time is 360 cycles (+22 %), which makes it nearly 10 % faster than seccomp. This speed up can be explained by the way the filter check is performed. While DPTI is constant in the check due to a direct access to the executed syscalls filter, seccomp either needs to traverse a BPF binary tree or a sequential list of filters. The least permissive filter for the syscall then determines whether it is allowed or not. Other work [18] has investigated a *skip-list* to improve the performance of seccomp filters, but as this approach is, to the best of our knowledge, not used in real-world applications we did not use it in our comparison.

Real-world Application Benchmark. While microbenchmarks show the specific effects of DPTI on syscalls, they do not allow reasoning about real-world application performance impact. Again, as no deep argument filtering is involved, there is no difference between DPTI-Stash and DPTI-Freeze.

For this evaluation, we need to determine the application's syscalls. Fortunately, several automated approaches have been demonstrated to get this information [18], [29], [10], which we extend to support DPTI. Our baseline is a vanilla version of the respective application. Both seccomp and DPTI are configured to only log syscall violations to ensure that syscalls not identified by the automated approach do not crash the tested application. We show the result of our evaluation in Table I. The commands used for each application are provided in Table IV in Appendix A.

TABLE I: The results of our performance evaluation of unsandboxed, seccomp- and DPTI-sandboxed applications. Overhead shows the percentage overhead compared to our baseline, *i.e.*, an unsandboxed version of the respective application. Sample sizes differs for various applications due to the difference in required runtime.

	Software	Sample Size	Normal Time / SEM	Seccomp Time (Overhead) / SEM	DPTI Time (Overhead) / SEM
busybox	diff	10 000	0.0025 s / 7.943 × 10^{-7}	0.011 s (340.0 %) / 1.256 × 10^{-6}	0.0033 s (32.0 %) / 7.986 × 10^{-7}
	true	10 000	0.0025 s / 6.480 × 10^{-7}	0.011 s (340.0 %) / 1.324 × 10^{-6}	0.0032 s (28.0 %) / 7.263 × 10^{-7}
	env	10 000	0.0025 s / 7.044 × 10^{-7}	0.011 s (340.0 %) / 1.291 × 10^{-6}	0.0032 s (28.0 %) / 7.022 × 10^{-7}
	ls	10 000	0.0026 s / 7.202 × 10^{-7}	0.011 s (323.08 %) / 1.300 × 10^{-6}	0.0033 s (26.92 %) / 6.983 × 10^{-7}
	dmesg	10 000	0.0025 s / 8.353 × 10^{-7}	0.012 s (380.0 %) / 2.243 × 10^{-6}	0.0041 s (64.0 %) / 1.340 × 10^{-6}
	cat	10 000	0.0025 s / 8.803 × 10^{-7}	0.011 s (340.0 %) / 1.687 × 10^{-6}	0.0032 s (28.0 %) / 6.839 × 10^{-7}
	head	10 000	0.0025 s / 9.609 × 10^{-7}	0.011 s (340.0 %) / 1.410 × 10^{-6}	0.0032 s (28.0 %) / 8.144 × 10^{-7}
	grep	10 000	0.0028 s / 9.607 × 10^{-7}	0.0113 s (303.57 %) / 1.501 × 10^{-6}	0.0035 s (25.0 %) / 8.813 × 10^{-7}
	pwd	10 000	0.0025 s / 7.664 × 10^{-7}	0.011 s (340.0 %) / 1.333 × 10^{-6}	0.0032 s (28.0 %) / 8.671 × 10^{-7}
git	diff	10 000	0.0092 s / 2.498 × 10^{-6}	0.0137 s (48.91 %) / 3.129 × 10^{-6}	0.0116 s (26.09 %) / 3.306 × 10^{-6}
	status	10 000	0.0068 s / 1.019 × 10^{-6}	0.0112 s (64.71 %) / 1.248 × 10^{-6}	0.0094 s (38.24 %) / 1.361 × 10^{-6}
ffmpeg	extract	10	12.1098 s / 8.482 × 10^{-3}	12.5605 s (3.72 %) / 8.249 × 10^{-3}	12.156 s (0.38 %) / 9.288 × 10^{-3}
	convert	10	18.6844 s / 9.543 × 10^{-3}	18.9921 s (1.65 %) / 6.754 × 10^{-3}	18.8285 s (0.77 %) / 9.704 × 10^{-3}
	remove	10	18.2757 s / 6.121 × 10^{-3}	18.4657 s (1.04 %) / 7.092 × 10^{-3}	18.4038 s (0.7 %) / 3.423 × 10^{-3}
	crop	10	12.3901 s / 1.467 × 10^{-2}	12.4444 s (0.44 %) / 2.614 × 10^{-2}	12.4219 s (0.26 %) / 4.593 × 10^{-3}
	info	10 000	0.0097 s / 9.124 × 10^{-7}	0.0567 s (484.54 %) / 2.150 × 10^{-6}	0.0551 s (468.04 %) / 3.205 × 10^{-6}
	change	10	18.2882 s / 9.466 × 10^{-3}	18.5225 s (1.28 %) / 9.164 × 10^{-3}	18.4518 s (0.89 %) / 1.333 × 10^{-2}

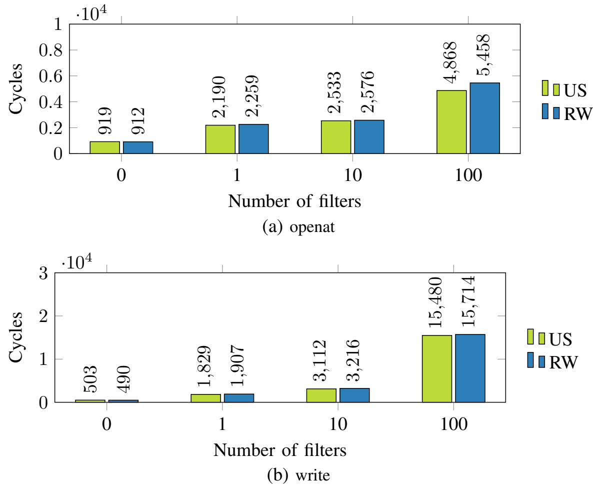


Fig. 5: Cycles required for string filtering for *openat* and *write* syscalls using DPTI, either using the US or RW bit. The x-axis indicates the position of the allowed string within the filter for the syscall, *i.e.*, the number of strings that need to be checked until a match occurs with the current argument.

We first consider the results of small applications and simple tasks of large ones, *i.e.*, busbox, git, the ffmpeg info command. For these applications, the overhead appears huge, ranging between 25 % and 468 % for DPTI. Still, for all cases, the overhead is lower than for seccomp where we measured an overhead between 48 % and 485 %. For more complex tasks in ffmpeg, the overhead never exceeds 0.9 % with DPTI, while in one case it almost reaches 4 % for seccomp.

Deep Argument Filtering. To evaluate the performance of DPTI for memory isolation, we use both variants for deep argument filtering. We consider two simple syscalls, *openat* and *write*. For both syscalls, we measure the average execution time of the syscall, including our deep argument filtering, over 100.000 invocations. As a syscall can allow multiple strings, we consider different number of strings in the filter, placing the correct one at its end, *i.e.*, for *n* strings in the filter we require *n* comparisons. As a baseline, we measure the execution time of the respective syscall without any argument checking.

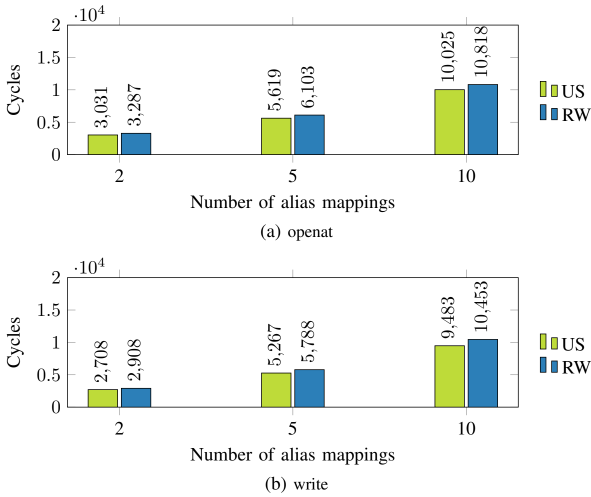
TABLE II: Cycle amounts of the individual components of our deep argument filtering over 100 000 executions of the respective syscall. Outliers were replaced with the median.

Syscall			Resolving PT	PT Manipulation	Flushing PT	Check
openat	DPTI-Stash	cycles	86	27	458	40
		SEM	0.0886	0.0028	0.0533	0.0024
	DPTI-Freeze	cycles	81	28	473	46
		SEM	0.0939	0.0041	0.0683	0.0028
write	DPTI-Stash	cycles	45	27	396	137
		SEM	0.1751	0.0028	0.0442	0.0442
	DPTI-Freeze	cycles	49	27	422	185
		SEM	0.2662	0.0034	0.1067	0.1480

Figure 5 shows the results of this experiment. Deep argument filtering has a non-negligible performance overhead: With DPTI-Stash, a single string check increases the execution time of the *openat* syscall from 920 to 2038 cycles. This increase is mostly due to the one-time overhead of isolating the memory region. With 10 string checks, the execution time is only slightly higher with 2351 cycles. In all cases, the performance of DPTI-Stash and DPTI-Freeze is about the same.

In-Depth Analysis of Filtering Components. To analyze the overhead further, we perform an in-depth analysis to determine the overhead of each individual step, *i.e.*, resolving page tables, bit manipulation, TLB flush, and the actual string comparison.

We instrument DPTI-Stash and DPTI-Freeze to measure the required cycles for each step. We use the same syscalls as in the previous experiment, but reduce the number of invocations to 100 000. We filter outliers—detected using the modified z-score—by replacing them with the median.



(a) openat

(b) write

Fig. 6: Cycles required for string filtering with alias mappings for *openat* and *write* syscalls using DPTI. The x-axis indicates the number of alias mappings for the filtered syscall argument, including the mapping used in the syscall itself.

The results of the evaluation are shown in Table II. The largest overhead is introduced by the page-table flush, which is necessary to perform twice. Recent work in Linux 5.13 has improved the performance of TLB flushing [55], which automatically improves the performance of our filtering as well. We discuss a potential way to further reduce this overhead in future work in Section VI-A. The string check differs between the two evaluated syscalls due to different string lengths used in the evaluation. Interestingly, clearing the additional bit in the kernel data structures required for DPTI-Freeze has no impact.

The execution time of all steps roughly sums up to the time difference shown for the case with no and one string filter in Figure 5. More efficient caching of page table translations can further improve the performance but was not implemented in our proof-of-concept implementation.

Alias Mappings. To evaluate the overhead of alias mappings, we build on our previous benchmark with the *openat* and *write* syscalls. Each syscall uses a single string filter, and we vary the number of alias mappings to the syscall argument, *i.e.*, 2, 5, and 10 mappings, including the mapping used in the syscall. Each syscall is executed 100 000 and the average execution time is taken. Figure 6 shows the result of this evaluation.

Unsurprisingly, the average execution time of a syscall increases with the number of alias mappings that have to be modified. In combination with the results of the previous in-depth analysis (Table II) and our performance analysis of non-alias mappings in syscalls (Figure 5), it is clear that most of the overhead is due to the additional, necessary TLB flushes. As previously mentioned does recent work on improving the performance of TLB flushes automatically improve the performance of our filtering as well [55]. Note that this overhead only materializes in the case of at least 2 alias mappings to the same page; otherwise the overhead shown in Figure 5 applies.

We also investigate the performance of our tracking itself, which shows that adding a new mapping on average adds 1222 cycles (N=100 000, +47.5 %) to the page fault. Most of this overhead, though, is not due to the tracking itself but a limitation of our proof-of-concept implementation as we need to probe (kretprobe) the page-fault handler. A full implementation can perform the tracking by extending it directly, removing on average 638 cycles of overhead from the page fault. Hence, the overhead of the tracking itself is 24.8 %. Additionally, a full implementation can modify existing kernel data structures, allowing for a more efficient tracking mechanism.

As we observe a more significant overhead when an alias mapping is used in a syscall, we now investigate whether such mappings are frequently used in syscalls in real-world applications. For this evaluation, we rely on the applications shown in Table I, *i.e.*, git, busybox, and ffmpeg, as well as visudo (Section V-A3). We track the number of alias mappings created over the execution of each one of our commands and whether such a mapping is then used in a string filter. This analysis revealed that not a single alias mapping is created and hence not used in a string filter. Therefore, the additional overhead is non-existent for at least these applications. We leave a broader analysis of alias mappings in real-world applications for future work.

2) Security Evaluation: For the security evaluation, we first evaluate whether clearing the US- or the RW-bit prevents modification of data. Then, we demonstrate that we can perform the deep argument check without interference from user space, *i.e.*, that our approach does not suffer from TOCTOU vulnerabilities. Finally, we discuss the security implications of bringing user-controlled pages into the kernel.

Modifying Bits in Page Tables. To ensure that the MMU-based isolation is reliable, we create a simple program that allocates a page of memory and then accesses it, observing, as expected, no crash. In the case of DPTI-Stash, we then use PTEditor [80] to clear the US-bit and access it again. As expected, the second access now results in a crash due to a privilege boundary violation. For DPTI-Freeze, we modified PTEditor such that it clears the VM_WRITE bit in the vm_flags of the associated vm_area_struct. While reading from the page still works, storing data to the page results in a crash due to the violation of the write protection. Hence, clearing the respective bits prevents another thread from modifying the data once the protection bits have been set appropriately.

Eliminating TOCTOU. While the previous experiment shows that the underlying principle works, we verify that it can prevent the exploitation of TOCTOU vulnerabilities. In this experiment, we try to re-direct an *openat* syscall such that a wrong file is opened. We create a multi-threaded application that uses filters that only allow opening the file *file1*. Thread 1 tries to open the specified file and print its content. Simultaneously, Thread 2 tries to modify the filename while the syscall is executed. In a vanilla or seccomp-based version, the syscall continues and opens the wrong file. With DPTI, the page containing the syscall is unmodifiable for the user. Hence, Thread 2 triggers a segfault due to the illegal access, and the thread is stalled, mitigating the attack.

User-space Pages in Kernel Space. When using DPTI-Stash, our enhanced filtering effectively brings a user-controlled page into the kernel space (cf. Section IV-A). To ensure that this cannot be exploited to inject arbitrary code into the kernel,

DPTI-Stash first verifies that the page is not marked as executable. As DPTI-Stash already has to modify the page-table entry for isolating the page, it can additionally clear the executable bit. Hence, as long as the page resides in the kernel space, it is not executable anymore. A thread cannot access the page while it is isolated, hence, removing the executable permission does not require any additional handling. After the syscall is done, DPTI-Stash can simply restore the executable bit. Thus, an attacker cannot exploit DPTI-Stash to inject executable pages into the kernel. Note that the kernel has to deactivate SMAP to access the syscall parameters in any case, enabling the access to the entire user-space memory during this time. Hence, temporarily bringing one data page into the kernel during a syscall does not increase the attack surface.

We consider the possibility of DPTI-Stash potentially weakening or breaking KASLR due to the user-space page being brought into the kernel. As it is only a data page, it can only be used as a stack page in a code-reuse attack where the attacker places the return addresses of the ROP gadgets. This already requires a KASLR break as the *ret* instruction can only return to absolute addresses. Hence, we can conclude that DPTI-Stash does not weaken or break KASLR.

3) Visudo: To demonstrate how deep argument filtering can improve the security of the system, we harden the *visudo* application with DPTI. *visudo* is used to edit the *sudoers* file with automatic validity checks. It uses an editor from a pre-defined set of editors to edit the file (potentially any editor). It can also be used to open other files than the *sudoers* file by using a command-line switch. Interestingly, the tool's manpage already mentions that this behavior can be a security hole:

"if visudo is configured with the --with-env-editor option or the env_editor Default variable is set in sudoers, visudo will use any the editor defines by VISUAL or EDITOR. [..] this can be a security hole since it allows the user to execute any program they wish simply by setting VISUAL or EDITOR."

To demonstrate that our enhanced filtering prevents exactly this scenario, *i.e.*, an arbitrary editor opening arbitrary files, we manually extend *visudo* with syscall filters. We restrict *visudo* to only allow the *vi* editor to open the *sudoers* files as well as files necessary for *vi* to work, *i.e.*, configuration files, libraries, and locales. Hence, in addition to allowing syscalls required for *visudo*, we also need to allow syscalls that *vi* requires.

In total, we generate 52 simple syscall filter rules without any argument filters. We allow the *execve* syscall with a deep argument filter on */usr/bin/vi*, hence an invocation with a different editor is prevented. Naturally, the list of pre-defined editors can be extended, and the restriction on *vi* is simply done to ease our proof-of-concept implementation. We add a deep argument filter on the *openat* syscall such that it only allows opening the *sudoers* file and all strictly necessary library and file dependencies of *visudo* and *vi*. This results in 47 deep argument filters for *openat*. In total, we define 100 filters. While all the simple syscall filters are supported by seccomp, the complex filters based on string comparisons are not supported. Hence, seccomp cannot restrict access to files and editors.

To demonstrate that the resulting *visudo* binary is still able to perform its job, we use it to modify the *sudoers* file using *vi*. When doing that, we did not observe a single crash, and the task completes successfully. On the other hand, when we try to override the editor or try to manipulate another file, our module detects that the specified filter rules are being violated and kills the application. Hence, DPTI can be used to close the previously mentioned security hole.

TABLE III: Comparison of the two variants of DPTI. Write and read possible only consider the ability of the user-space application to read or write, not the kernel.

Approach	Modified Bit	Read possible	Write Possible	Tracking
DPTI-Stash	US	✗	✗	✗
DPTI-Freeze	RW VM_WRITE	✓	✗	✓

We evaluate the performance of our protected version of *visudo* over 10 000 executions and compare it to a vanilla version. For the evaluation, we measure the performance of the non-interactive verification mode of *visudo*. Nevertheless, the filtering is the same as in the interactive modification mode, *i.e.*, filters for the *openat* syscall are still checked. The vanilla version requires, on average, 12.8 ms, DPTI-Stash 13.2 ms, and DPTI-Freeze 13.0 ms.

4) Comparison DPTI-Stash and DPTI-Freeze: While both variants of DPTI solve the problem of deep argument filtering, they differ in what MMU mechanism enforces the protection. As the used bits differ in their semantics, the two variants exhibit different properties, which we discuss in more detail now. The results of our comparison are presented in Table III.

DPTI-Freeze requires manipulation of bits in two different locations, but has the advantage that a user-space thread can still read data from the modified page, which is not possible with DPTI-Stash. For DPTI-Freeze, we need to explicitly track the previous permissions of the modified page such that we can restore the correct state after the syscall completes. This is not necessary for DPTI-Stash. However, the additional bit to modify does not introduce any overhead (Table II).

In both cases, writes are not possible from user space. Additionally, DPTI-Freeze blocks writes from the kernel. While this does not provide security benefits, it can be disadvantageous if the kernel wants to write to the isolated page. However, we have not encountered such a case in our tests.

Summary

We presented a detailed performance and security evaluation, showing that DPTI outperforms seccomp in the case of simple syscall filters while providing additional security functionality in the form of deep argument filtering. We highlighted the bottlenecks of the approach and discussed the main differences between DPTI-Stash and DPTI-Freeze. Finally, we showed on the example of *visudo* how our approach can prevent a real-world security risk.

B. SGX-Protection Domain

We evaluate the performance (Section V-B1) and the security guarantees (Section V-B2) of the SGX protection domain, including the more optimized version from Section IV-B3.

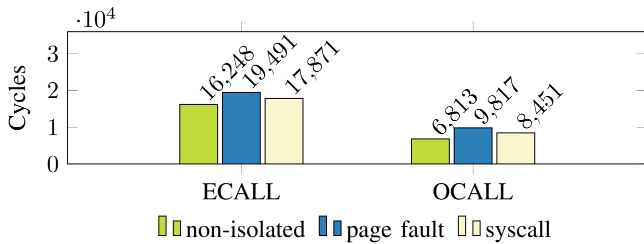


Fig. 7: Microbenchmark results for ECALLs and OCALLs for a non-isolated enclave, one using a page fault for ending isolation, and one using a syscall instead of the page fault.

1) Performance: As page table mappings are enforced by hardware, no additional performance overhead is observed during the regular execution of an enclave. The protection domain alters the page tables during enclave enter and exit, affecting the latency of enclave transitions only. We evaluate the overhead using *sgxbench* [72]. For our evaluation, we restrict ourselves to the *empty ECALL* and *empty OCALL* benchmark of *sgxbench* as these model the worst case scenario for our protection. These benchmarks simply perform ECALLs or OCALLs with no additional code inside the function.

Setup. We perform our benchmarks on an *Intel Core i5-8265U* CPU with a fixed frequency of 2.9 GHz. The operating system is Ubuntu 20.04 with Linux kernel 5.8.0. We compare DPTI-Stash to a reference URTS SGX-PSW library as baseline, and execute each benchmark 2 000 000 times. We also evaluate the performance improvements of the slightly less secure, optimized version (cf. Section IV-B3).

Results. Figure 7 shows the benchmark results. We observe a latency increase of 19.9 % for ECALLs and 44.0 % for OCALLs. When using the optimized version using the syscall instruction, we observe only an overhead of 9.9 % for ECALLs and 24.0 % for OCALLs. This improvement is solely caused by the different execution times of the page-fault handler and the syscall handler. We discuss how future work can optimize our implementation in Section VI-A. In similar microbenchmarks, SGXJail [94] had an overhead of 41.4 % for ECALLs and 45.2 % for OCALLs. Hence, as shown by the microbenchmarks, optimized DPTI outperforms SGXJail by 22 % for ECALLs and 14.6 % for OCALLS.

2) Security: In this section, we evaluate the security the DPTI-Freeze- and DPTI-Stash-based enclave isolation.

Data-Only Attacks. A non-isolated enclave can read or modify host application memory, hence allowing a malicious enclave to perform data-only attacks. DPTI prohibits data-only attacks by limiting write access to host pages, allowing the enclave to modify pages marked explicitly as DBPs only. As these pages are, by definition, used to pass data from and to a possibly untrusted enclave, we require the host to verify the correctness of the data received from the enclave.

SGX-ROP. To perform ROP-based attacks, the enclave needs to scan the host application for gadgets to build a ROP chain and then overwrite the stored return address on the stack to execute the ROP chain [81]. With both DPTI-Freeze and DPTI-Stash, we prohibit modifications to the return address on the stack by aligning the stack downwards before entering isolation and restricting access to the upper stack pages containing the return address. As the enclave can freely modify stack and base pointer upon leaving isolation, DPTI verifies that the latter was not altered. For data pushed on the stack, we have to consider the two reasons a thread can exit an enclave: First, the thread can exit the enclave to return from an ECALL. In this case, we verify that the stack pointer was restored correctly and restore the saved registers before executing the return instruction. Second, the enclave exited to perform an OCALL. The data is pushed onto the user stack, and the protection only verifies that the stack pointer grew downwards. No return instruction is executed directly after the isolation end as an OCALL adds additional call frames instead of removing them.

EEXIT Destination. The `EEXIT` instruction allows arbitrary return locations outside the SGX enclave, allowing returns to arbitrary, potentially missaligned, instructions [94]. We consider two cases: First, the enclave returns to a page different than the CBP, raising a page fault. As discussed in Section IV-B, this results in our page-fault handler raising a segmentation fault. Second, the enclave returns to the CBP, potentially misaligned. This poses no problem as the CBP is only filled with `NOP` instructions following the `EENTER`.

Data Confidentiality. DPTI-Stash provides data confidentiality as host application pages are not mapped in the isolated address space. DPTI-Freeze does not guarantee data confidentiality as pages are just mapped as read-only, allowing the enclave to read host data, but it still prevents malicious modification.

Multithreading and TOCTOU. As described in Section IV-B, multiple threads inside the same enclave share an isolated mapping. This does not pose a security problem, as threads within an enclave already share the enclave memory. Nevertheless, we ensure that multiple threads running inside the enclave cannot alter each other's stack during an OCALL by removing that thread's stack from the isolated mapping.

> **Summary**
> We presented a microbenchmark evaluation of our SGX protection domain, showing the performance overhead for ending the isolation via a page fault or syscall. The benchmarks demonstrate that DPTI outperforms previous solutions in this field without requiring additional hardware features. Our security evaluation showed that we improve the security of the system in the presence of a malicious enclave.

VI. Discussion

In this section, we discuss future work and related work.

A. Future Work

Enhanced Syscall Filtering. The largest bottleneck for efficient deep argument filtering is the necessity of TLB flushes. Future work can investigate the possibility of employing protection keys in the kernel, completely removing the necessity of modifying and flushing page tables. Previous work has already explored the possibility of using such keys in the kernel [33]. A more recent patch set has even provided protection-key functionality to the kernel [13]. As the reliance on protection keys contradicts the aim of this work for providing a method for commodity, off-the-shelve systems, they were not

used. Similarly, IMIX [26] could improve the performance of our deep argument filtering. However, currently, no hardware supports IMIX. Future work can investigate whether seccomp can be extended to include our approach directly. To generate tight filters for complex arguments, the automated generation of syscall filter rules [29], [10], [18] can be extended to extract arguments automatically.

SGX-Protection Domain. The benchmarks in Section V-B1 show that the overhead between raising a page fault and handling it is quite high. Merging the proof-of-concept directly into the kernel would increase the performance by greatly reducing the amount of code executed.

B. Related Work

Syscall Filtering. Multiple works have been published on syscall checking [32], [90], [71], [43], [28], [2], [3], [69], [24], [56], [15], [30]. Some rely on kernel tracing [32], [90], [71], [43], [28], [2], [3], but the performance suffers from additional context switches. Hence, Linux relies on seccomp to perform syscall-filtering if it is requested by a developer. As discussed, seccomp requires a developer to manually identify the syscalls required by an application. Several recent works have investigated the feasibility of automatically identifying and generating these filter rules, eliminating the need for manual analysis [18], [29], [10]. While seccomp improves the security, it negatively effects the performance (cf. Section V-A). Recent work has therefore proposed changing how seccomp handles filters to improve the performance [36], [85]. Nevertheless, deep argument filtering is still not supported as of Linux 5.11. Salaün [75] restricts ambient rights, such as global filesystem accesses, for a set of processes. This is an orthogonal approach as it does not attempt to filter syscalls, but instead focuses on access control on kernel objects directly [19].

Memory Isolation. Memory isolation is a well-researched field where proposals can be grouped into OS-, virtualization-, hardware-, language- and runtime-based techniques. While memory isolation can be easily achieved on the OS level by simply placing the necessary parts into separate processes, this does incur a significant overhead. To prevent this significant overhead, recent work provides additional OS abstractions [16], [37], [58] that together with compiler support [12] or runtime analysis tools [8] make it feasible to isolate long-term signing keys in a web server. Other work has proposed to use a hypervisor for isolating memory. Dune [7] allows a user-space process to use the Intel VT-x virtualization extensions to isolate compartments. Other work demonstrated how the *VMFUNC* instruction can be used to switch extended page tables to achieve in-process isolation [51], [59]. SIM [84] uses VT-x for isolating a security monitor in an untrusted VM.

Memory isolation can also be enforced by the hardware, for instance, by Intel SGX or ARM TrustZone. Previous work used SGX to protect internal data structures of just-in-time compilers to prevent code-injection attacks [25]. While the hardware enforces this isolation, the switching overhead is similar to other approaches [51]. Other works proposed additional x86 ISA extensions to add load and store instructions that must be used to access data in a safe region [26], [65]. These approaches additionally require CFI [1] to protect against control-flow hijack attacks [86]. Several works have relied on Intel Memory Protection Keys (MPK) to facilitate more efficient memory isolation [88], [76], [51], [35], [68]. However, Intel MPK is not used in the kernel and not available for most commodity systems. It is only available in a limited subset of shipped processors since the Intel Xeon Skylake microarchitecture. Creative use of page-table entries for in-process isolation is a technique that has been explored [26], [51], but these solutions require modifications of the ISA or the re-purposing of ignored or reserved bits in page tables. Consequently, they can only be applied to future hardware but not to commodity systems.

Memory isolation can also be ensured through static checks in memory-safe languages. In unsafe languages, this isolation can be provided through software fault isolation, where runtime checks are added by the compiler or through binary rewriting [91], [64]. Naturally, this imposes a performance overhead while not protecting against control-flow hijack attacks, making it necessary to combine it with CFI. Recent work has explored the possibility of so-called *zero-cost transitions* between normal and sandboxed code for well-structured code, but still requires CFI [50]. Contrary to previous solutions for memory isolation [26], [65], [88], [76], [51], [35], [68], DPTI does not require ISA extensions or re-purposing of ignored bits in the page table. Instead, we solely rely on existing functionality and bring our enhanced filtering to widely available commodity systems. DPTI retrofits sandboxing mechanisms with strategies that previous works have explored against microarchitectural attacks [34], [38].

Intel SGX. The memory isolation of SGX is asymmetric, *i.e.*, while enclave memory is inaccessible for the operating system and the host application, the enclave has full access to the data and code of the host application. Schwarz et al. [81] showed that untrusted SGX enclaves can rewrite the host application memory to impersonate the host and execute arbitrary syscalls. To mitigate such attacks, Weiser et al. [94] proposed SGXJail, which contains two ways to isolate enclaves, either isolation through another process or through a slightly modified variant of Intel MPK. Other works have proposed to monitor the I/O behavior of enclaves to detect a potential attack [17], [14]. Another proposed approach is to analyze enclave code before running it, but this is not possible for generic loaders [14]. Hence, Costan and Devadas [14] proposed to force generic loader enclaves to embed malware-analysis code into the enclave, but it is unknown how effective this approach is.

Other approaches try to improve the security of the enclave itself [52], [82], [98], but such approaches are orthogonal to our approach as they assume a malicious host or OS. A SFI-based approach has been proposed by Ryoan [39], but this requires recompilation of the enclave using SFI, which might not be possible and is not necessary in our approach.

VII. Conclusion

In this paper, we proposed DPTI, a security-domain isolation mechanism for commodity off-the-shelf CPUs. We presented two novel techniques for dynamic, time-limited changes to the memory isolation at security-critical points, called DPTI-Freeze and DPTI-Stash. While DPTI-Freeze makes memory temporarily read-only, DPTI-Stash temporarily removes access permissions. We evaluated the versatility of DPTI in two

scenarios. In the first scenario, DPTI enables faster and more fine-grained syscall filtering than seccomp-bpf while providing stronger memory-safety guarantees that allow supporting deep argument filtering. In the second scenario, DPTI efficiently confines SGX enclaves, outperforming existing solutions by 14.6 %-22 % and removing hardware requirements. Our results show that DPTI is a viable mechanism that can be used to isolate domains within applications without relying on special hardware instructions or extensions.

ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 681402). Additional funding was provided by generous gifts from ARM, Amazon, and Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-Flow Integrity," in *CCS*, 2005.

[2] A. Acharya and M. Raje, "MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications," in *USENIX Security Symposium*, 2000.

[3] A. Alexandrov, P. Kmiec, and K. Schauser, "Consh: Confined Execution Environment for Internet Computations," The University of California, Santa Barbara, Tech. Rep., 1999.

[4] Android, "Application Sandbox," 2021. [Online]. Available: https://source.android.com/security/app-sandbox

[5] AppArmor, "AppArmor: Linux kernel security module," 2021. [Online]. Available: https://apparmor.net/

[6] M. Backes and S. Nürnberger, "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing," in *USENIX Security Symposium*, 2014.

[7] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe User-level Access to Privileged CPU Features," in *OSDI*, 2012.

[8] A. Bittau, P. Marchenko, M. Handley, and B. Karp, "Wedge: Splitting Applications into Reduced-Privilege Compartments," in *NSDI*, 2008.

[9] D. Bueso, "tools/perf-bench: Add basic syscall benchmark," 2019. [Online]. Available: https://lore.kernel.org/patchwork/patch/1048777/

[10] C. Canella, M. Werner, D. Gruss, and M. Schwarz, "Automating Seccomp Filter Generation for Linux Applications," in *CCSW*, 2021.

[11] S. Checkoway, L. Davi, A. Dmitrienko, A. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *CCS*, 2010.

[12] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu, "Shreds: Fine-Grained Execution Units with Private Memory," in *S&P*, 2016.

[13] J. Corbet, "Memory protection keys for the kernel," 2020. [Online]. Available: https://lwn.net/Articles/826554/

[14] V. Costan and S. Devadas, "Intel SGX Explained," *Cryptology ePrint Archive, Report 2016/086*, 2016.

[15] A. Dan, A. Mohindra, R. Ramaswami, and D. Sitaram, "Chakravyuha (CV): A sandbox operating system environment for controlled execution of alien code," Tech. Rep., 1997.

[16] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation," in *ASPLOS*, 2015.

[17] S. Davenport and R. Ford, "SGX: the good, the bad and the downright ugly," January 2014. [Online]. Available: https://www.virusbulletin.com/virusbulletin/2014/01/sgx-good-bad-and-downright-ugly

[18] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis, "sysfilter: Automated System Call Filtering for Commodity Software," in *RAID*, 2020.

[19] K. development community, "Landlock: kernel documentation," 2020. [Online]. Available: https://landlock.io/linux-doc/landlock-v21/security/landlock/kernel.html

[20] J. Edge, "A seccomp overview," 2015. [Online]. Available: https://lwn.net/Articles/656307/

[21] ——, "Deep argument inspection for seccomp," September 2019. [Online]. Available: https://lwn.net/Articles/799557/

[22] ——, "Seccomp and deep argument inspection," June 2020. [Online]. Available: https://lwn.net/Articles/822256/

[23] B. Ford and R. Cox, "Vx32: Lightweight User-level Sandboxing on the x86," in *Usenix ATC*, 2008.

[24] T. Fraser, L. Badger, and M. Feldman, "Hardening COTS software with generic software wrappers," in *DISCEX*, 2000.

[25] T. Frassetto, D. Gens, C. Liebchen, and A.-R. Sadeghi, "JITGuard: Hardening Just-in-Time Compilers with SGX," in *CCS*, 2017.

[26] T. Frassetto, P. Jauernig, C. Liebchen, and A.-R. Sadeghi, "IMIX: In-process memory isolation extension," in *USENIX Security Symposium*, 2018.

[27] T. Garfinkel, "Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools," in *NDSS*, 2003.

[28] T. Garfinkel, B. Pfaff, M. Rosenblum *et al.*, "Ostia: A delegating architecture for secure system call interposition." in *NDSS*, 2004.

[29] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal System Call Specialization for Attack Surface Reduction," in *USENIX Security Symposium*, 2020.

[30] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson, "SLIC: An Extensibility System for Commodity Operating Systems," in *USENIX ATC*, 1998.

[31] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *S&P*, 2014.

[32] I. Goldberg, D. Wagner, R. Thomas, E. A. Brewer *et al.*, "A secure environment for untrusted helper applications: Confining the wily hacker," in *USENIX Security Symposium*, 1996.

[33] S. Gravani, M. Hedayati, J. Criswell, and M. L. Scott, "IskiOS: Lightweight Defense Against Kernel-Level Code-Reuse Attacks," *arXiv:1903.04654*, 2019.

[34] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is Dead: Long Live KASLR," in *ESSoS*, 2017.

[35] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty, "Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries," in *Usenix ATC*, 2019.

[36] T. Hromatka, "seccomp and libseccomp performance improvements," 2018.

[37] T. C.-H. Hsu, K. Hoffman, P. Eugster, and M. Payer, "Enforcing Least Privilege Memory Views for Multithreaded Applications," in *CCS*, 2016.

[38] Z. Hua, D. Du, Y. Xia, H. Chen, and B. Zang, "EPTI: efficient defence against meltdown attack for unpatched VMs," in *USENIX ATC*, 2018.

[39] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," in *Usenix OSDI*, 2016.

[40] G. Inc., "Seccomp filter in Android O," 2017. [Online]. Available: https://android-developers.googleblog.com/2017/07/seccomp-filter-in-android-o.html

[41] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide," 2019.

[42] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block Oriented Programming: Automating Data-Only Attacks," in *CCS*, 2018.

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

[44] V. Kemerlis, "Protecting commodity operating systems through strong kernel isolation," Ph.D. dissertation, Columbia University, 2015.

[45] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking kernel isolation," in *USENIX Security Symposium*, 2014.

[46] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kguard: Lightweight kernel protection against return-to-user attacks," in *USENIX Security Symposium*, 2012.

[47] Z. Kenjar, T. Frassetto, D. Gens, M. Franz, and A. Sadeghi, "V0LTpwn: Attacking x86 Processor Integrity from Software," in *USENIX Security Symposium*, 2020.

[48] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.

[49] V. Kiriansky and C. Waldspurger, "Speculative Buffer Overflows: Attacks and Defenses," *arXiv:1807.03757*, 2018.

[50] M. Kolosick, S. Narayan, C. Watt, M. LeMay, D. Garg, R. Jhala, and D. Stefan, "Isolation Without Taxation: Near Zero Cost Transitions for SFI," *arXiv:2105.00033*, 2021.

[51] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No Need to Hide: Protecting Safe Regions on Commodity Hardware," in *EuroSys*, 2017.

[52] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer, "SGXBOUNDS: Memory Safety for Shielded Execution," in *EuroSys*, 2017.

[53] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-Pointer Integrity," in *OSDI*, 2014.

[54] B. Lan, Y. Li, H. Sun, C. Su, Y. Liu, and Q. Zeng, "Loop-oriented programming: a new code reuse attack to bypass modern defenses," in *IEEE Trustcom/BigDataSE/ISPA*, 2015.

[55] M. Larabel, "Concurrent TLB Flushing For Linux 5.13 Provide A Small Performance Benefit," 2021. [Online]. Available: https://www.phoronix.com/scan.php?page=news_item&px=Linux-5.13-Concurrent-TLB-Flush

[56] C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, and J. H. Hartman, "Protecting against Unexpected System Calls," in *USENIX Security Symposium*, 2005.

[57] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *USENIX Security Symposium*, 2018.

[58] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhattacharjee, and P. Druschel, "Light-weight contexts: An OS abstraction for safety and performance," in *OSDI 16*, 2016.

[59] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting Memory Disclosure with Efficient Hypervisor-Enforced Intra-Domain Isolation," in *CCS*, 2015.

[60] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "Aslr-guard: Stopping address space leakage for code reuse attacks," in *CCS*, 2015.

[61] LWN, "The current state of kernel page-table isolation," 2017. [Online]. Available: https://lwn.net/SubscriberLink/741878/eb6c9d3913d7cb2b/

[62] P. Matousek, "CVE-2016-5195 kernel: mm: privilege escalation via MAP_PRIVATE COW breakage," 2016. [Online]. Available: https://bugzilla.redhat.com/show_bug.cgi?id=1384344#c16

[63] M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell, "System V Application Binary Interface," *AMD64 Architecture Processor Supplement, Draft v0.99.7*, 2014.

[64] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC Architecture," in *USENIX Security Symposium*, 2006.

[65] L. Mogosanu, A. Rane, and N. Dautenhahn, "MicroStache: A Lightweight Execution Context for In-Process Safe Region Isolation," in *RAID*, 2018.

[66] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based Fault Injection Attacks against Intel SGX," in *S&P*, 2020.

[67] Nergal, "The advanced return-into-lib(c) explits: PaX case study," 2001.

[68] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK)," in *USENIX ATC*, 2019.

[69] D. S. Peterson, M. Bishop, and R. Pandey, "A Flexible Containment Mechanism for Executing Untrusted Code," in *USENIX Security Symposium*, 2002.

[70] V. Prevelakis and D. Spinellis, "Sandboxing applications," in *USENIX ATC*, 2001.

[71] N. Provos, "Improving host security with system call policies," in *USENIX Security Symposium*, 2003.

[72] R. Quinonez, "SGXBENCH framework for benchmarking SGX enclaves," 2018. [Online]. Available: https://github.com/sgxbench/sgxbench

[73] C. Reis, A. Moshchuk, and N. Oskov, "Site Isolation: Process Separation for Web Sites within the Browser," in *USENIX Security Symposium*, 2019.

[74] R. Rogowski, M. Morton, F. Li, F. Monrose, K. Z. Snow, and M. Polychronakis, "Revisiting Browser Security in the Modern Era: New Data-Only Attacks and Defenses," in *EuroS&P*, 2017.

[75] M. Salaün, "Landlock: unprivileged access control," 2016.

[76] D. Schrammel, S. Weiser, S. Steinegger, M. Schwarzl, M. Schwarz, S. Mangard, and D. Gruss, "Donky: Domain Keys–Efficient In-Process Isolation for RISC-V and x86," in *USENIX Security Symposium*, 2020.

[77] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. Sadeghi, and T. Holz, "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications," in *S&P*, 2015.

[78] M. Schwarz, C. Canella, L. Giner, and D. Gruss, "Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs," *arXiv:1905.05725*, 2019.

[79] M. Schwarz, D. Gruss, M. Lipp, C. Maurice, T. Schuster, A. Fogh, and S. Mangard, "Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features," in *AsiaCCS*, 2018.

[80] M. Schwarz, M. Lipp, and C. Canella, "misc0110/PTEditor: A small library to modify all page-table levels of all processes from user space for x86_64 and ARMv8," 2018. [Online]. Available: https://github.com/misc0110/PTEditor

[81] M. Schwarz, S. Weiser, and D. Gruss, "Practical Enclave Malware with Intel SGX," in *DIMVA*, 2019.

[82] J. Seo, B. Lee, S. M. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "Sgx-shield: Enabling address space layout randomization for sgx programs." in *NDSS*, 2017.

[83] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *CCS*, 2007.

[84] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure In-VM Monitoring Using Hardware Virtualization," in *CCS*, 2009.

[85] D. Skarlatos, Q. Chen, J. Chen, T. Xu, and J. Torrellas, "Draco: Architectural and Operating System Support for System Call Security," in *MICRO*, 2020.

[86] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," in *S&P*, 2013.

[87] Tizen, "Security:Seccomp," 2018. [Online]. Available: https://wiki.tizen.org/Security:Seccomp

[88] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure and Efficient In-process Isolation with Memory Protection Keys," in *USENIX Security Symposium*, 2019.

[89] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *S&P*, 2020.

[90] D. A. Wagner, "Janus: An Approach for Confinement of Untrusted Applications," University of California at Berkeley, Tech. Rep., 1999.

[91] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *ACM SOSP*, 1993.

[92] P. Wang and J. Krinke, "How Double-Fetch Situations turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel," in *USENIX Security Symposium*, 2017.

[93] R. N. M. Watson, "Exploiting Concurrency Vulnerabilities in System Call Wrappers," in *WOOT*, 2007.

[94] S. Weiser, L. Mayr, M. Schwarz, and D. Gruss, "SGXJail: Defeating enclave malware via confinement," in *RAID*, 2019.

[95] M. Wiki, "Project Fission," 2019. [Online]. Available: https://wiki.mozilla.org/Project_Fission

[96] ——, "Security/Sandbox," 2019. [Online]. Available: https://wiki.mozilla.org/Security/Sandbox

[97] S. Wiki, "FAQ — SELinux Wiki," 2009. [Online]. Available: http://selinuxproject.org/w/?title=FAQ&oldid=729

[98] W. Zhao, K. Lu, Y. Qi, and S. Qi, "MPTEE: Bringing Flexible and Efficient Memory Protection to Intel SGX," in *EuroSys*, 2020.

APPENDIX

In Table IV, we show the exact commands that were used for the evaluation of the real-world applications in Section V-A.

TABLE IV: The specific commands which were used for evaluating the performance of an unsandboxed, seccomp-, and DPTI-sandboxed version of the respective application.

	Software	Command
busybox	diff	busybox diff cat.sh grep.sh
	true	busybox true
	env	busybox env
	ls	busybox ls
	dmesg	busybox dmesg
	cat	busybox cat test
	head	busybox head -n 100 test
	grep	busybox grep -ir python3 .
	pwd	busybox pwd
git	diff	git diff
	status	git status
ffmpeg	extract	ffmpeg_g -i video.mp4 -r 1 -f image2 image-%2d.png -y -hide_banner
	convert	ffmpeg_g -i video.mp4 video.avi -y -hide_banner
	remove	ffmpeg_g -i video.mp4 -an output.mp4 -y -hide_banner
	crop	ffmpeg_g -i video.mp4 -filter:v 'crop=640:480:200:150' output.mp4 -y -hide_banner
	info	ffmpeg_g -hide_banner -i video.mp4
	change	ffmpeg_g -i video.mp4 -filter:v scale=1280:720 -c:a copy output.mp4 -y -hide_banner