µSWITCH: Fast Kernel Context Isolation with Implicit Context Switches

Dinglan Peng*, Congyu Liu*, Tapti Palit*, Pedro Fonseca*, Anjo Vahldiek-Oberwagner†, Mona Vij†
*Purdue University {peng301, liu3101, tpalit, pfonseca}@purdue.edu
†Intel Labs {anjo.lucas.vahldiek-oberwagner, mona.vij}@intel.com

***Abstract*—Isolating application components is crucial to limit the exposure of sensitive data and code to vulnerabilities in the untrusted components. Process-based isolation is the de facto isolation used in practice, e.g., web browsers. However, it incurs significant performance overhead and is typically infeasible when frequent switches between isolation domains are expected. To address this problem, many intra-process memory isolation techniques have been proposed using novel kernel abstractions, recent CPU extensions (e.g., Intel® MPK), and software-based fault isolation (e.g., WebAssembly). However, these techniques insufficiently isolate kernel resources, such as file descriptors, or do so by incurring high overheads when resources are accessed. Other work virtualizes the kernel context inside a privileged user space domain, but this is ad-hoc, error-prone, and provides only limited kernel functionalities.**

We propose µSWITCH, an efficient kernel context isolation mechanism with memory protection that addresses these limitations. We use a protected structure, shared by the kernel and the user space, for context switching and propose implicit context switching to improve its performance by deferring the kernel resource switch to the next system call. We apply µSWITCH to isolate libraries in the Firefox web browser and an HTTP server, and reduce the overhead of isolation by 32.7% to 98.4% compared with other isolation techniques.

***Index Terms*—Systems security, operating systems security.**

1. Introduction

Modern software involves large and complex codebases, which significantly benefit from isolation techniques that protect sensitive data and code. Using isolation techniques, it is possible to contain the impact of vulnerabilities within application sub-components. For example, popular web browsers, such as Chromium and Firefox, have recently introduced *site isolation* [1, 2] to isolate tabs according to the site origin using independent processes. This prevents attackers from exploiting vulnerabilities in libraries and other components that could otherwise compromise the entire browser; hence, isolation prevents attackers from stealing sensitive user data associated with other websites. Isolation limits the damage to potentially vulnerable applications.

Process-based isolation is one of the strongest isolation techniques for applications because it isolates kernel resources across domains and user-level resources (e.g., process memory). However, process-based isolation incurs a significant performance overhead that can exceed several microseconds [3]. This overhead is due to expensive traps to the kernel that arise from the inter-process communication and context switches. In practice, this approach is infeasible when frequent switches between isolation domains are expected. [4] Hence, process isolation is generally used at a coarse granularity. In contrast, lightweight context switching techniques [5, 6] aim to provide similar security guarantees but with less overhead. However, the performance overhead of these techniques is still high (the median overhead reported by lwC [6] is $2.01\mu s$), preventing their wide-scale adoption for fine-grained isolation.

Taking advantage of recent CPU extensions [4, 7], such as Intel® MPK, and software-based isolation (SFI) techniques [8], several systems implement intra-process *memory isolation*. Unfortunately, memory isolation techniques are insufficient to isolate generic applications because it is also necessary to isolate the kernel context across domains [9]. Isolating the kernel context across domains ensures that a domain cannot read the files written by another or execute other system calls that could interfere with the process execution. Hence, WebAssembly (or Wasm) and other techniques often rely on virtualizing the kernel context inside a privileged userspace domain. However, this approach is ad-hoc and error-prone, given the complexity of the system call interface. Furthermore, this approach only provides very limited coverage of the system call interface in practice.

To address the limitations of prior techniques, we propose µSWITCH, a novel approach to efficiently isolate selected kernel resources of a process and application memory into multiple fine-grained domains (µContexts). Isolating kernel resources with µSWITCH ensures security guarantees similar to those of process isolation but with a significantly lower performance overhead. In particular, µSWITCH achieves sandboxed function invocation in $129ns$, which is $55\times$ faster than process isolation.

The novelty of µSWITCH lies in the *safe* use of a protected data structure shared between the process and the kernel, which allows the user space application to initiate a µContext-switch from user space, without trapping into the kernel. This constitutes an *implicit system call*—an elegant kernel abstraction that does not require major kernel changes. The actual switching of the kernel resources is deferred till the next system call. Protecting this shared data structure only requires light-weight memory protection techniques that enforce authorized updates to this structure. Indeed, we opted to use Intel's®'s Memory Protection Keys [7] to provide memory protection, but µSWITCH's approach is compatible with any memory protection technique.

Isolating intra-process µContexts poses multiple challenges. The first comes in the form of establishing the semantics of intra-process kernel resource isolation. Because µSWITCH isolates kernel resources at a granularity not considered by previous work, we had to first define the semantics of this isolation. For example, does a µContext have its own Process Control Block within the operating system kernel? Can a µContext be shared by multiple threads of the same process? We also had to reason about the implications of isolating kernel resources, such as Seccomp filters, which are traditionally applied to the entire process. Similarly, in the user space, the safe isolation of *all* in-memory application data of a µContext requires untangling shared dependencies, such as the Libc, which is typically accessed by multiple functions, and even other libraries.

To demonstrate the applicability of µSWITCH, we apply it in the context of *library isolation*, where we isolate the application core and each application library into different domains. In particular, we show that µSWITCH can be used to significantly reduce the isolation granularity of the Firefox browser. Browsers are prime use cases for isolation techniques because they use many third-party libraries that expose an extensive attack surface. For example, the libogg library used by Firefox for audio decoding was found to have a remote code execution vulnerability (CVE-2017-14632 [10]). Similarly, libvpx was found to have a remote information disclosure bug (CVE-2019-9232 [11]). These vulnerabilities show that the browser libraries are error-prone; hence, they should be isolated.

RLBox [3] proposed isolating the Mozilla Firefox libraries using WebAssembly, which has since been deployed in recent Firefox official releases [12]. However, while its original implementation provided both an SFI-based isolation approach and, a more secure, process isolation approach, the performance overhead of the latter (98% on average) was significantly higher than the former (37% on average). This forced Mozilla to use the less secure version.

Apart from the limitation of the less secure isolation mechanism, Firefox only uses one sandbox for each web domain-library pair. However, *per (web) component* isolation would provide finer-grained isolation, which would provide higher security. For instance, creating a new libjpeg sandbox for every JPEG image, instead of a libjpeg sandbox for each web domain (e.g., `*.google.com`), would prevent an attacker from stealing, within a domain, data processed by the library. This is particularly important when domains include content under the control of different actors. However, our experiments showed that implementing the more secure process-based isolation with such fine granularity causes prohibitive overheads (up to 604%).

We apply µSWITCH to two use cases, Firefox and an HTTP server, to demonstrate efficient library isolation. For Firefox, we extend RLBox [3] to support µSWITCH and isolate four libraries. Our results show an increased page load latency of only 1.5% for per-origin and 5.5% for per-component isolation across popular websites. Our approach presents a significantly lower performance overhead (1.5% for per-origin compared to 3.8% and 23.2%, for lwC [6] and WebAssembly [13] respectively). In case of the HTTP

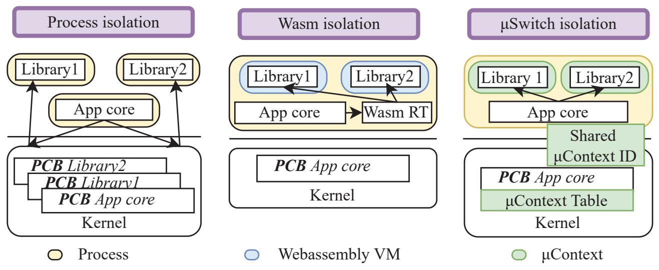


Figure 1: Kernel Context Isolation as implemented by process-based, WebAssembly, and µSWITCH.

server, we isolate the `libevent` network library with an average overhead of 5.8%, which is significantly lower than prior techniques (19.4% for lwC and 63.3% for process-based isolation).

2. Background and Motivation

Providing isolation for general-purpose applications requires isolating the process memory *and* kernel context for each domain. This section discusses the main techniques used to isolate each component and their limitations.

2.1. Memory Isolation

Software-based fault isolation (SFI) is a well-known technique that uses runtime memory access checks to protect memory across domains. For instance, WebAssembly [13] provides a linear memory abstraction and instruments all memory access instructions to enforce memory address bounds. The binary instrumentation used by SFI often requires special compiler and runtime support and can cause a significant performance overhead that depends on the memory access patterns. Hence, with the recent release of Intel® MPK on commodity CPUs [7], there has been a renewed interest in accelerating memory protection using hardware-assisted techniques across a range of use-cases [4, 14–17]. Intel® MPK provides user-level instructions that allow the application to efficiently switch page table permissions, obviating the need for expensive instrumentation or kernel invocations during switches. §5 explains in more detail Intel® MPK and how µSWITCH can leverage it.

2.2. Kernel Context Isolation

Memory isolation techniques, such as SFI or MPK, are crucial to isolate domains, but they are insufficient because they do not isolate kernel resources. Kernel resources need to be isolated across domains to ensure that domains cannot interfere with each other through the kernel. Without kernel context protection across domains, a domain could invoke system calls that, for instance, read the files written by other domains using `read()`, access the memory of other domains using `process_vm_readv()`, or even create new executable pages using the `mprotect()` system call. Figure 1 depicts the two industry standards for kernel context isolation using processes or WebAssembly, and compares them to µSWITCH.

In-process Kernel Context Virtualization. One approach to isolating kernel context consists of virtualizing kernel resources [18, 19] inside a trusted domain of the application. For instance, the WebAssembly System Interface (WASI) [18] is typically used to manage the interactions of WebAssembly domains with the operating system. WASI provides a set of POSIX-like system calls to WebAssembly domains and sanitizes the call arguments. In addition, it ensures that kernel resources, such as file descriptors, are isolated among sandboxes through trusted usermode logic in a trusted domain that interposes between the kernel and untrusted domains. Unfortunately, this approach is redundant with the kernel isolation mechanisms, and it is impractical to fully apply it to the extensive and complex kernel interface.

Process Isolation. A straightforward approach to isolating kernel contexts is to split the application across processes, such that each domain runs in a different process. In practice, this approach tends to be particularly effective because kernels traditionally rely on process isolation to ensure security in multi-tenant environments. For instance, containers (e.g., Docker [20]) typically use process isolation in addition to other kernel mechanisms that limit kernel resource scope (Linux namespaces [21]) and usage (Linux cgroups [22]).

Unfortunately, process isolation has notoriously high domain switching overhead, since the application has to make explicit calls to the kernel to switch between domains. For example, when using shared memory to synchronize the processes, the `futex` system call is often used, which operates on the wait queue of the scheduler and can cause multiple context switches for one domain switch in some cases. Other IPC methods such as pipes can also be used, which may cause even higher overhead. This process incurs in more than $7\mu s$ for a sandboxed function invocation (Table 2), which is $55\times$ slower than our system with MPK and $3238\times$ slower than a standard function call. Hence, process isolation is infeasible for fine-grained isolation in which high domain switch frequency is expected.

Lightweight Context Switching. To address this problem, recent work has proposed lightweight context switching techniques, such as lwC [6], SMV [5], SeCage [23], and Virtines [24]. These approaches generally switch faster between domains than process isolation. However, they still require trapping to the operating system kernel and manipulating the page tables or flushing TLB on each domain switch, which is expensive. The high overhead of these methods motivates us to propose a faster kernel isolation technique that relies on implicit context switches.

3. Implicit Context Switching

Our approach relies on the observation that the main cost of process isolation arises from the explicit system calls required to perform context switches. Because process isolation requires two explicit calls to context switch for each back-and-forth interaction (domain A to B, and B to A), *fine-grained sandboxing* results in significant overhead. Furthermore, since the switching cost is fixed, it has to be paid even if no system call is invoked by a domain in a given transition, which further increases the relative overheads. Note that because untrusted domains can be compromised, we cannot assume that domains do not make system calls in a given transition even when they are expected to make no system calls. Hence, it is necessary to ensure a kernel context change for every domain switch.

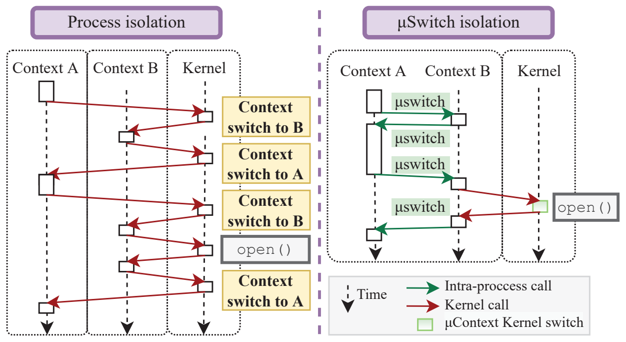


Figure 2: Process and µSWITCH isolation comparison. µSWITCH switches between domains without invoking explicit system calls (or interrupts), whereas process isolation requires explicit system calls.

This work proposes *implicit context switching* to address this problem (Figure 2). Our key insight is that the kernel does not need to be aware of the kernel context switch until the domain makes its first system call (since the last domain switch). This provides an opportunity to defer the context switch to the first system call; in particular, it allows us to batch the context switching system call with the first system call made by the domain. Our approach is particularly useful in fine-grained sandboxing where the average number of system call invocations per domain transition is very low.

Implicit context switching has another key advantage: when an application switches from domain A to B and back to A without B invoking a system call, this approach entirely avoids kernel context switching. The idea is to selectively context switch when and only if a domain makes a system call.

For implicit context switching to work there are two challenges that need to be addressed. First, it is necessary to ensure that batching of the context switch system call with the first system call occurs, even when the domain making the call is compromised. Second, it is necessary to integrate the implicit context switch, which operates asynchronously, with efficient memory protection techniques such that both the memory and kernel context are consistently isolated.

4. µSWITCH Design

µSWITCH creates multiple lightweight execution contexts within the same process to isolate kernel resources and process memory. We call these µContexts or simply *contexts*. Each µContext is represented by a unique context *descriptor*. The kernel context for each µContext is controlled by a context-descriptor that is shared between user and kernel space. Switching between the different contexts involves simply updating this shared context-descriptor to that of the new µContext. µSWITCH depends on memory

TABLE 1: Core µSWITCH API interface.

Method	Description
`int init(int flags)`	Initializes µSWITCH and creates the privileged µContext.
`int newContext(uswctx_t *ctx, int flags)`	Creates an unprivileged µContext.
`int destroyContext(uswctx_t ctx)`	Destroys an unprivileged µContext.
`int sandboxCall(uswctx_t ctx, SandboxFunc f, ...)`	Calls sandboxed function `f` in µContext `ctx`.
`int registerPrivCall(uswctx_t ctx, Handler handler)`	Registers a privcall with the given handler and returns the privcall-id.
`int privCall(int id, ...)`	Calls privileged function `id` with the given arguments.
`int dupFileDescriptor(uswctx_t ctx, int fd)`	Duplicate the file `fd` from µContext `ctx` to the prvileged µContext.

protection mechanism that isolates the in-memory data of each µContext as well as µSWITCH metadata including the shared context-descriptor, for which we use Intel® MPK as explained in §5. As discussed in §3, we use implicit context switch to avoid unnecessary expensive switches to kernel space. This section elaborates on µSWITCH's design in detail.

4.1. System Model

A µSWITCH-enabled program consists of privileged functions and sandboxed functions. The program always begins with the privileged functions in the privileged context. The privileged functions dynamically create an isolated context (µContext) for each sandbox. This divides the program into multiple isolated contexts, each of which has a set of kernel resources and a memory domain. The privileged functions run in the privileged context allowing arbitrary memory accesses and system call invocations, while a sandboxed function is limited to its own kernel resources and memory domain as defined during context creation. For example, each µContext independently opens files and network sockets and the privileged context may opt in to set a Seccomp filter [25] controlling access to system calls and system resources. In the extreme case, for instance, µSWITCH can deny the invocation of any system call to a specific function, while allowing another function to execute any system call.

Invocations between privileged and sandboxed functions are mediated by the µSWITCH API. A privileged function can invoke a *sandboxcall*, which switches to an unprivileged context, and executes the sandboxed function. After the function finishes, the execution switches back to the privileged context and continues to execute after the sandboxed function call. In the reverse direction, in order to support callback functions, a sandboxed function can invoke a *privcall*, which switches to a privileged context, executes a privileged function and switches back to the unprivileged context after the call. Similar to invoking a system call, the sandboxed function needs to provide a privcall id and the arguments. Then, the previously registered privcall handler for the privcall id is executed.

The sandboxed functions can not invoke arbitrary functions in the privileged context. For a function to be callable from the unprivileged sandboxes it has to be explicitly registered as a privcall handler. A *trusted entrypoint* marks the single entry gate into the privileged context. All entries into the privileged context including privcalls from the unprivileged contexts and returns from sandboxcalls are mediated through this trusted entrypoint. In case of a privcall from the unprivileged context, the trusted entrypoint simply dispatches the privcall to the pre-registered privcall handler depending on the privcall id specified by the sandbox context. In case of a return from the sandboxcall, it resumes control within the privileged context.

Table 1 shows the core API interface of the µSWITCH user space library. The API covers the functions for the µContext creation and destruction, sandboxcall invocation, and privcall registration and invocation.

4.2. Threat Model

µSWITCH makes no assumption about sandboxed functions. These functions execute arbitrary code and may contain memory corruption or control flow hijack vulnerabilities that an attacker could try to exploit.

We assume privileged functions, µSWITCH, the operating system, and hardware to not include vulnerabilities violating µSWITCH's security guarantees, directly leak sensitive information to sandboxed functions, or switch into sandboxes functions while still in a privileged context. In particular, we assume that the privileged function sets up µSWITCH correctly. Similarly to the approach used in process isolation, concerns with the kernel attack surface can be mitigated by filtering system calls with Seccomp. Because each µContext has its own kernel context, the system call filters can be better fitted to implement the principle of least privilege with µSWITCH.

µSWITCH relies on memory protection to isolate the memory of individual µContexts and ensure the integrity of the shared context-descriptor. Hence, µSWITCH inherits the assumptions made by the memory protection used. Our implementation uses Intel® MPK (see §5 and §6.2).

Like other intra-process isolation techniques, we consider side-channel and hardware attacks, such as rowhammer, out of the scope of this paper. µSWITCH is compatible with existing defenses [26] and can be extended to prevent such attacks. Finally, we assume the kernel is correct, which can be achieved through formal verification [27–30], testing [31–33], and hardening techniques [34].

Security and Liveness Guarantees. µSWITCH provides a similar security guarantee to process-based isolation for memory and all the isolated kernel resources. When accessing those resources, the privileged functions and the sandboxed functions run as if they were different processes.

µSWITCH also provides the liveness guarantee that the unprivileged contexts can not crash the process by faulting, such as throwing exceptions or performing unauthorized operations. Such faults generate signals such as `SIGSEGV`

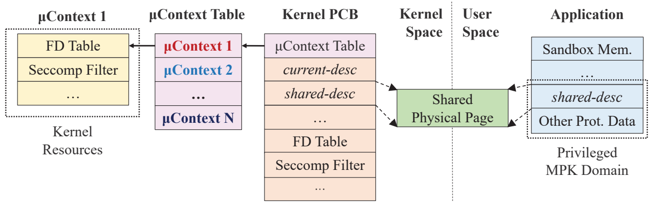


Figure 3: µSWITCH design overview.

or `SIGSYS`, which the privileged context can trap by registering signal handlers. These signal handlers when invoked allow the privileged context to handle the error instead of terminating the entire process.

4.3. Fine-grained Kernel Resource Isolation

Figure 3 presents an overview of the µSWITCH data structures. The process sends the current µContext identity via the *shared-descriptor* to the kernel. To minimize changes to the kernel, µSWITCH only extends the Process Control Block (PCB) with a pointer to the *shared-descriptor*, *current-descriptor*, and *context table*. Any context switch updates the existing resource pointers and values like the pointer to the file descriptor table, or Seccomp profiles in the PCB. As a result, µSWITCH's kernel changes are minimally invasive and do not require intense kernel refactoring (see §6.1).

This design allows most PCB resources to be isolated per µContext with minimal changes to the kernel's and µSWITCH's implementation. With our use case of isolating libraries in a browser §7.1 and an HTTP server §7.2 in mind, we decided to isolate the following resources: file descriptors, Seccomp profile, filesystem information, process credentials and user/mount/network/uts namespaces. For instance, isolating file descriptors, filesystem information (the root and working directories), and mount/network namespaces allows a privileged context to isolate its files and network connections from untrusted libraries. Whereas with isolated Seccomp profiles, a library's access to system resources can be denied to all system calls even for a single function. Prior work [20, 35–37] only achieves this level of isolation via independent processes.

Sharing Resources. µSWITCH allows µContexts to share resources. For resources that can be shared by the `clone` system call such as file descriptors, the user can share them among µContexts by specifying the corresponding flags in the `newContext` method. For other resources, the user can exclude them from the isolation domain by specifying the corresponding flags in the `init` method. µSWITCH does not touch shared resources when switching.

4.4. Implicit µContext Switch Execution

To allow kernel resources switches from user space without an explicit system call, µSWITCH maintains an object shared between the kernel and user space, which stores the context descriptor of the currently active context – the *shared-descriptor*. By updating its value, a process can switch between µContexts. The shared-descriptor is memory-protected to prevent unauthorized tampering by adversaries. If the shared-descriptor does not point to a valid µContext, the kernel resources will not be switched. Also, we maintain an object in the kernel space storing the context descriptor that the kernel observed last time the program entered the kernel, the *current-descriptor*. By comparing the current-descriptor and the shared-descriptor, the kernel knows whether there a kernel resource switch is necessary.

Instead of immediately invoking a system call to switch the kernel resources when the user space application updates the shared-descriptor, we defer the switch till the next system call. The execution proceeds normally till the next system call is executed. Operating system kernels typically contain an *entrypoint* where all system calls enter the kernel. At this entrypoint, the kernel first reads the shared-descriptor to check if the shared-descriptor has been updated. If a change is observed, then it updates the process's PCB pointers to point to the kernel resources of the new context and updates the current-descriptor to the shared-descriptor. More details of the entrypoint can be found in Appendix A.

By deferring the switch, we eliminate unnecessary system call invocations to switch the kernel context. If the sandboxed or privileged function do not perform system calls, µSWITCH entirely avoids the PCB switch. Figure 4 illustrates a switch among three µContexts with only one kernel resource update when invoking a system call.

5. Memory Protection with MPK

This section describes the design of the µSWITCH memory protection component. Our current implementation uses Intel® Memory Protection Keys (MPK) [7] to isolate the in-memory data of each µContext and the shared-descriptor. We choose Intel® MPK among the hardware- (e.g., VMFUNC, page tables) and software-based isolation techniques, due to its low context switch cost and ability to execute code at native speed, unlike WebAssembly which can slow down code by 40-55% [38].

5.1. Background: Memory Protection Keys

Intel® MPK is an x86 CPU extension to restrict memory accesses of groups of memory pages in a user space application. Unlike SMV [5] and lwC [6], which split a process into multiple address spaces for memory protection, Intel® MPK partitions the memory of the same address space into several domains. Every page table entry (PTE) contains a 4-bit tag indicating the page's domain. Access permission (read or write) to each domain is controlled via a new PKRU register, which is read and written via special usermode instructions - `RDPKRU` and `WRPKRU`. In our design, each memory domain is mapped to a kernel µContext. In order to switch contexts, we update the value of the PKRU register via the unprivileged `WRPKRU` instruction, which does not update the kernel's current µContext, but ensures isolation as long as no system call occurs. The PKRU register may also be updated via the instruction `XRSTOR`. Note that execute permission cannot be disabled by MPK. However, even if attackers can jump to the privileged code, they cannot read or write to protected memory or tamper with the isolated kernel resources. Like other MPK-based systems [4], we assume that no secrets are stored in executable memory.

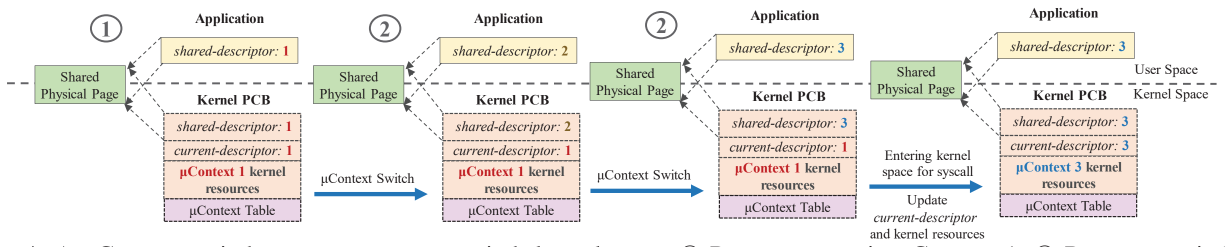


Figure 4: A µContext switch prevents context switch kernel traps. ① Program runs in µContext 1. ② Program switches to µContext 2 by updating *shared-descriptor*. ③ Program switches to µContext 3 by updating *shared-descriptor*. ④ Program invokes a syscall. In the syscall entrypoint, the kernel resources and *current-descriptor* are updated to µContext 3 before actually performing the system call.

5.2. Ensuring Integrity of PKRU Updates

The PKRU register can be updated from user space, which makes it possible to design fast switch gates without trapping to the kernel. However, for the same reason, additional defenses are needed to prevent an attacker from invoking unintended gadgets in the application binary that correspond to the `WRPKRU` instruction sequence. This would allow an attacker to circumvent the memory protection offered by Intel® MPK. Our PKRU defenses are inspired by previous work that aim to secure Intel® MPK [4, 19, 39–42]. In our design, we only allow *safe* occurrences of the `WRPKRU` and `XRSTOR` instructions, as described next. The switch gates design is based on these safe `WRPKRU` instructions.

Legitimate occurrences of `WRPKRU` instructions are present under two kinds of switch gates: (a) privileged context → unprivileged contexts and (b) unprivileged contexts → privileged context. We must ensure that the attacker can not perform unauthorized jumps to these `WRPKRU` instructions to gain unauthorized privilege. The patterns of the safe occurrences of `WRPKRU` instructions in two kinds of gates are described as follows:

In the gate that switches from the unprivileged context to the privileged context, i.e., to invoke a privcall or to return from a sandboxcall, we ensure that right after the `WRPKRU` instruction, there is *always* an instruction sequence that jumps to the trusted entrypoint, as discussed in §4.1. As the trusted entrypoint is set up by the privileged context (pre-registered privcall handlers or return points from sandbox calls), this prevents the unprivileged context from breaking out of the sandbox. We discuss the implementation details of the trusted entrypoint in §6.2.

In the gate that switches from the privileged context to an unprivileged context, i.e., to invoke a sandboxcall or to return from a privcall, we write the new PKRU value to a per-thread variable and then load it to `%eax` before the `WRPKRU` instruction. This variable is *read-only* for the unprivileged contexts. After the `WRPKRU` instruction, we load the variable and compare it with `%eax`. If they are different, the process will invoke the `exit` system call immediately. This system call can be intercepted by the Seccomp filters and raise a SIGSYS signal to allow the signal handler to return to the privileged function, which can handle the error. This use of the `WRPKRU` instruction ensures that the program has to write to the PKRU register to a protected variable before actually setting it. So when the attacker tries to make use of that `WRPKRU` instruction by jumping to it from the unprivileged context with `%eax` being set with a value that allows higher permission, the program will immediately invoke the `exit` system call as `%eax` does not match the value stored in the protected read-only memory, thus preventing the attacker from escaping from the isolation. More details of the switch gates can be found in Appendix B.

The `XRSTOR` instructions can also overwrite the PKRU register when the PKRU restore bit is enabled in the `%eax` register. Hence, a safe occurrence of the `XRSTOR` instruction is followed by a sequence that enforces the PKRU restore bit in `%eax` to be zero.

5.3. Preventing MPK Attacks with System Calls

Prir work [9, 19, 41, 42] showed that Intel® MPK's protections can be circumvented through some system calls. Accordingly, to ensure Intel® MPK is used safely, we need to disable the unsafe system calls, such as `sigreturn`, `process_vm_readv` and `pkey_mprotect`. However, some of these system calls are necessary for applications. Unlike other approaches, our fine-grained system call filtering approach ensures that we can restrict these system calls within the unprivileged contexts without breaking the functionality of the *entire* application. µSWITCH achieves this by setting appropriate Seccomp filters for the µContext of a sandboxed function while not restricting privileged functions.

6. Implementation

This section discusses the implementation details of µSWITCH. In particular, we describe the µSWITCH kernel and user space components. We implement µSWITCH in 872 lines of kernel code, 4493 lines of user space library, and modify 147 lines in Musl and 227 lines in RLBox.

6.1. µSWITCH Kernel Components

We implement a prototype of µSWITCH in Linux 5.15.2 for the x86-64 architecture. We define three new types for our purposes Figure 3. To represent an individual µContext, we define the `uswitch_context` structure, which stores a pointer to the isolated file-descriptor table and an isolated Seccomp structure. The `uswitch_context_table` stores each process' list of µContexts. Finally, we define the `uswitch_data` structure, which encapsulates all µSWITCH-related data. This structure stores the pointer to the uswitch_context_table for a process, the current-descriptor, along with a pointer to the *shared-descriptor* stored in the shared page.

The Linux kernel represents each Process Control Block (PCB) as an instance of the task_struct object. Therefore, we modify this structure to store a pointer to the uswitch_data. The task_struct structure also contains the pointer to the process file-descriptor table and a Seccomp structure which contains the mode, reference count and a pointer to the filter. By updating these pointers to refer to the per-context file-descriptor table and the Seccomp filter, we isolate these resources.

Multi-threading. To support multi-threading, an object of type uswitch_context_table is shared per process (or more precisely, a thread group in the kernel). Each thread in the process has its own uswitch_data object and can thus switch to a different µContext independently of other threads in the process. We add spin locks and atomic reference counts to the uswitch_data and uswitch_context_table structures and carefully deal with them as well as the locks and reference counts of other kernel data structures including the file descriptor tables, the Seccomp filters, and the memory management structures.

System Call Entry. For every µSWITCH-enabled process, we set the SYSCALL_WORK_ENTER flag to one. Then, the Linux kernel will invoke the syscall_trace_enter() function for all system call invocations from that process, and we can modify this function to perform the implicit context switch. We compare the *shared-descriptor* with the current-descriptor to see if *shared-descriptor* has been changed since last time the process entered the kernel. The *shared-descriptor* is stored in a page that is mapped in both the kernel virtual address and the user virtual address, so it can be directly accessed in either mode. If the *shared-descriptor* has been changed by the user space, we first update the current-descriptor in the uswitch_context_table for the process, and then, we update the files_sturct pointer and the seccomp structure in the process's task_struct to refer to the resources of the current µContext.

Cross-µContext File Access. Often the privileged context needs to access the files of the unprivileged contexts. To support this, we provide the dupFileDescriptor API function and a corresponding system call that duplicates a file descriptor from another context. This system call copies a file descriptor from another µContext's file descriptor tables to the current one, which is similar to the pidfd_getfd system call for copying files between processes. To prevent the unprivileged contexts from accessing other µContexts' files, this system call should be blocked in the unprivileged contexts.

6.2. µSWITCH User Space Components

Our implementation ensures that the heap, stack, and global memory are correctly protected using MPK. In addition, it ensures that no security-critical user space data is overwritten during the µSWITCH operations, even when there are multiple levels of nested µSWITCH invocations.

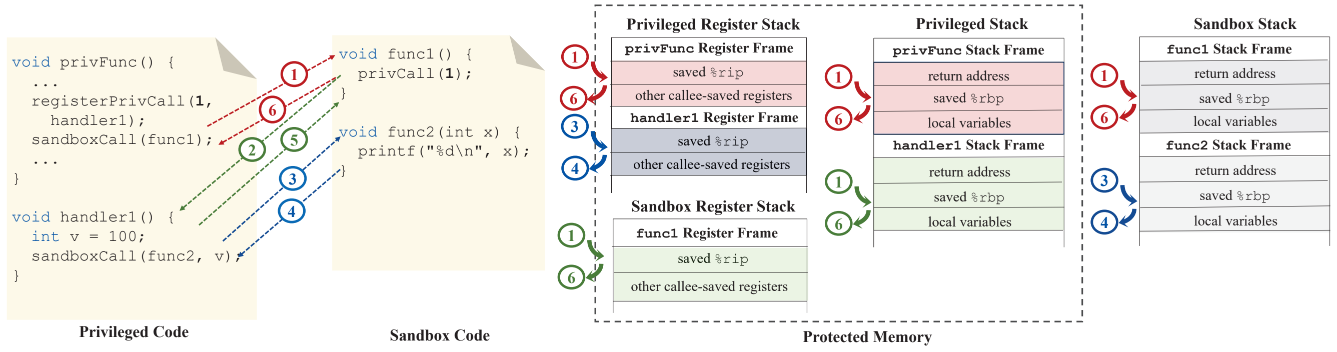
Stack Isolation. To protect the sensitive data on the stack and prevent attackers from modifying the return addresses and hijacking the control flow, we isolate both the stacks of the privileged context and the sandboxes. Each context, including the privileged context and the unprivileged contexts, has its own stacks in its own memory domain. The challenge lies in securely switching between these stacks for privcalls and sandboxcalls. When the program execution performs a sandboxcall, it must push its current state consisting of contents of all registers including %rsp and its return address to a tamper-proof, precomputed address. To achieve this, in addition to the privileged program stack, we also maintain a *Privileged Register Stack* that is located in memory, per-thread, at a fixed offset from the %fs segment. This fixed offset is decided at compile time and hardcoded in the program. Similarly, the base address of the %fs segment itself can only be updated via the system call arch_prctl, which we disable inside the sandbox contexts, or via the WRFSBASE instruction, which we disable by binary inspection discussed later.

On a return from a sandboxcall, the trusted entrypoint first updates the MPK domain via the WRPKRU instruction and then pops and restores the privileged state from the Privileged Register Stack. Restoring the %rsp register results in the restoration of the privileged stack, thus re-establishing the execution environment for the privileged context.

Similarly, we also maintain a *Sandbox Register Stack* inside the protected memory at a precomputed fixed-offset from the %fs segment. When a sandbox performs a privcall, its register state is pushed into it to protect it from corruption by other concurrently executing sandboxes. The entire process of pushing and popping the Privileged and Sandbox Register Stacks for a sequence of privcalls and sandboxcalls are shown in Figure 5. With the help of these stacks, sandboxcalls can potentially be nested as denoted by ③.

Protection Key Virtualization. Intel® MPK only supports up to 16 memory domains. Fortunately, prior work, such as libmpk [15], demonstrate how to virtualize Intel® MPK beyond 16 keys. This allows µSWITCH to support more than 16 µContexts at the cost of slower context switches when the working set of active contexts is larger than 16.

Binary Inspection. As discussed in §5.2, µSWITCH must ensure that there are only safe occurrences of the WRPKRU and XRSTOR instructions. To check that, our implementation uses a binary inspection technique inspired by ERIM[4]. When initializing the program and loading the sandboxes, we scan the whole address space to find the executable pages and search for byte sequences corresponding to WRPKRU and XRSTOR instructions in those pages. If *unsafe* occurrences of WRPKRU and XRSTOR are found, the program will abort or rewrite the instructions. In practice, our implementation aborts except for some simple cases where we can easily rewrite them, which are discussed later. Also, we disable WRFSBASE to prevent the attacker from tampering


```
void privFunc() {
  ...
  registerPrivCall(1,
    handler1);
  sandboxCall(func1);
  ...
}

void handler1() {
  int v = 100;
  sandboxCall(func2, v);
}
```
```
void func1() {
  privCall(1);
}
void func2(int x) {
  printf("%d\n", x);
}
```

Figure 5: An example of nested sandbox invocations. Same-colored arrows denote a call-ret pair. ① Privileged function `privFunc` calls sandboxed function `func1`. ② Sandboxed function invokes *privcall* 1, for which the handler is `handler1`. ③ `handler1` calls sandboxed function `func2`. ④ `func2` returns. ⑤ *privcall* 1 returns. ⑥ `func1` returns.

the base address of the `%fs` segment during the binary inspection. To prevent the attacker from injecting executable code that contains unsafe instructions, we apply standard data execution prevention (DEP), ensuring all executable pages are not writable and have to be validated before being made executable.

Our analysis of popular libraries, however, shows that unsafe occurrences are rare, which is supported by other's extensive analysis of binaries and libraries in different Linux distribution [4]. We found only one unsafe occurrence of `WRPKRU` in the `pkey_set()` function of glibc. As we do not allow the user to modify the PKRU register, we simply replace the `WRPKRU` instruction with three `NOP` instructions at runtime. Also, we found that there is one unsafe occurrence of `XRSTOR` in `ld.so`, which is necessary for the program's functionality. Therefore, we dynamically rewrite it to a safe occurrence.

C Library Isolation To ensure the correct isolation of process memory and to support the functions such as multi-threading, the isolation of C library is necessary, which we discuss in detail in Appendix D.

7. Use Case

This section discusses two use cases of µSWITCH: the Firefox web browser and an HTTP server, which demonstrate the benefits of our system in isolating the kernel resources.

7.1. Fine-grained Browser Isolation

We first applied µSWITCH to the Firefox web browser to sandbox the different libraries used for rendering a website. We ported µSWITCH to the RLBox framework, which already provides a WebAssembly-based isolation mechanism and a process-based isolation mechanism. µSWITCH improves the context switch performance compared to process-based isolation allowing isolation of libraries at even finer grain than the state-of-the-art per-origin approach. As a result, µSWITCH improves security, further limiting the capabilities of attackers.

Security Benefits for Fine-grained Isolation While per-origin sandboxing [2, 3] improves security over no sandboxing, it can still lead to information disclosure when the same origin hosts content from multiple mutually untrusted sources. For instance, a domain `upload.example.com` may store JPEG images uploaded by different users that are loaded on the same web page. If the JPEG library, on the user's web browser, has an arbitrary code execution vulnerability that can be exploited by a malicious image, the attacker can exploit it by uploading such a malicious image to `upload.example.com`. In particular, when the victim opens a page that contains a malicious image, the malicious image can exfiltrate the contents of other secret images that are loaded on the same web page. With per-component isolation, i.e., isolating every component on the web page into its own sandbox, this attack is prevented.

RLBox with µSWITCH. RLBox is Firefox's in-production library isolation framework [12]. We modify its codebase [43] to use µSWITCH when isolating libraries. RLBox uses NaCl in the original paper [44] and WebAssembly in production. Both require modifying and recompiling isolated libraries. Porting a library to use µSWITCH with RLBox only requires linking with our Musl library.

Binary Inspection for JIT Code. Firefox uses JIT compilers to accelerate Javascript code, thus introducing untrusted binary instructions in the browser's address space. To ensure that no unsafe `WRPKRU`, `XRSTOR` or `WRFSBASE` instructions occur in the emitted code, we apply binary inspection to the pages containing JIT-compiled Javascript code, before making them executable. Currently we simply abort the application if those instructions are found. While we have not observed any occurrence of these instructions in the JIT-compiled Javascript code, for the websites we have tested with, this has the potential of breaking Firefox's execution. In the future, we plan to investigate disabling the JIT compilers and falling back to the interpreter if the JIT-compiled binary contains any unsafe instructions.

7.2. Network Library Isolation in HTTP Server

HTTP servers are often externally exposed to serve the requests and need elevated permissions, such as file system access and CGI program execution. Thus, security vulnerabilities in the HTTP server components, such as the network library or script interpreters, may compromise sensitive resources. For example, a vulnerability [45] of the Apache HTTP server shows that the code in the worker processes or threads, such as a malicious PHP script, can execute arbitrary code with root privileges. Therefore, fine-grained isolation across the HTTP server components is critical for security. To this end, we applied µSWITCH to a simple HTTP web server, which is based on the sample web server [46] provided by the libevent library, an event-driven network library that provides asynchronous event notification based on the kernel functions such as `select` or `epoll` and is used in many applications such as Google Chrome and Tor. Protecting the web server based on libevent thus shows the generality and production-readiness of our system.

To apply µSWITCH to the HTTP server, we split the program into two components. One of them contains the sandboxed libevent including its event loop, and is allowed to use the network kernel interfaces including `accept`, `read` and `epoll`. The other component resides in the privileged context, and reads files from the file system to send them to the client via the libevent sandbox. Following the principle of least privilege [47], we use Seccomp filters on the HTTP server that minimize the system calls it can invoke. This use case uses and isolates multiple kernel resources, including file descriptors and Seccomp filters. Also, it uses system calls such as `select` and `epoll`, that other sandboxing techniques, such as WASM's WASI interface, do not support. This shows µSWITCH's advantage of wider kernel function coverage than WASI.

8. Evaluation

To evaluate the effectiveness of our system, we evaluate µSWITCH with both microbenchmarks and real world use-cases. In particular, we measure the performance overhead of our system and compare it with pure MPK-, lwC-, WebAssembly- and process-based isolation techniques. The results show that our system can achieve low performance and memory overhead despite higher security guarantees.

Evaluation Methodology. We use a server with two Intel® Xeon Platinum 8380 CPUs (with microcode 0xd000331) and 512 GiB RAM. We disable Turbo Boost to keep the CPU at the constant frequency of 2.3 GHz (e.g., no power throttling). To run our experiments in a reproducible way, we run a virtual machine using KVM on the server. For the guest machine, we allocate 4 cores and 32 GiB RAM for the page load experiments, and 16 cores and 32 GiB RAM for other experiments. We run Ubuntu 18.04 with our modified Linux kernel version 5.15.2. Kernel page-table isolation (KPTI) is disabled by default for the CPU. All our experiments use Glibc version 2.27, Musl version 1.2.2, Firefox 57, and WASI SDK 14 which compiles the libraries to WebAssembly. For the experiments with Firefox, we use the RLBox code released with its paper [43]. The libraries we isolate are also from the source code tree of Firefox 57. We compare µSWITCH throughout the evaluation to native execution without isolation as baseline and a pure MPK, lwC, WebAssembly, and process isolation technique, as follows:

Pure MPK To evaluate the additional overhead caused by kernel resource isolation, we remove kernel resource switching from µSWITCH to get an unsafe, pure MPK version.

TABLE 2: Sandboxcall invocation latency.

Type	Latency (ns)		
	w/o Syscall	w/ 1 Syscall	w/ 2 Syscalls
Native Function	2.2	227	437
µSWITCH	129	380	943
elwC	921	1175	1402
MPK	128	364	584
WebAssembly	2.2	233	468
Process	7124	9284	7801

Light-Weight Contexts (lwC). As lwC is implemented on FreeBSD, we cannot directly compare with it. Instead, we implement an approximation on Linux with explicit context switch, i.e., a system call that explicitly switches kernel resources. Since lwC switches page tables when switching kernel resources, we also add page table flushing to that system call to take this overhead into account. Since lwC enables the Process Context Identifier (PCID) [48] feature on modern CPUs to prevent TLB flushes on context-switches, our implementation of lwC enables PCID too. We refer to this approximated version of lwC as elwC (explicit-context-switch lwC) for simplicity. A single switch of elwC takes about $0.45\mu s$, which is smaller than $2.01\mu s$ reported in [6], so elwC is a conservative approximation for comparison.

WebAssembly and Process Isolation For both isolation methods, we use the original implementation of RLBox. However, for the WebAssembly version, we had to backport the WebAssembly isolation of the deployed RLBox version, to its original implementation, which only used process isolation (and the obsolete NaCL). This ensures the same renderer and other Firefox components across experiments.

8.1. Microbenchmarks

8.1.1. Sandboxcall Invocation Latency

We evaluate the sandboxcall invocation overhead of µSWITCH (Table 2). To this end, we write a simple function that returns the sum of two integers, and sandbox it using five techniques: µSWITCH, WebAssembly-based SFI, MPK without resource isolation, elwC, and process isolation.

Each experiment invokes the sandboxed function 1,000,000 times and reports the average runtime. Each sandbox invocation causes two domain switches (privileged $\rightarrow$ sandbox $\rightarrow$ privileged).

As expected, compared to the heavyweight process isolation technique, the sandbox invocation latency of µSWITCH is significantly lower. In particular, µSWITCH is 55$\times$ faster (7124ns vs 129ns) when no system calls are performed by the sandbox, which is a common case in many fine-grained sandboxes (e.g., image decompression).

Interestingly, we observed that process isolation with two system calls has a lower latency than with a single system call. After tracing the kernel scheduler events, we could attribute this behavior to the different scheduler patterns exhibited by the Linux kernel's Completely Fair Scheduler [49]. In contrast, the WebAssembly based sandbox, does not involve any additional operations *upfront* at domain-switch time, and thus its overhead in this experiment is

TABLE 3: µSWITCH sandboxcall inv. latency breakdown.

	Locks	MPK	Others	**Total**
Time (ns)	30	48	51	129

closer to that of a native function invocation. However, unlike µSWITCH and process isolation, due to the instrumentation costs, SFI techniques encounter very significant overheads *inside* the execution of the sandbox functions (i.e., during user-level instruction execution).

To understand the overhead of function invocation introduced by µSWITCH and pinpoint the source of this overhead, we also measure the function invocation latency of several different variants of µSWITCH. We derive this decomposition by gradually removing each component of µSWITCH, that could potentially cause a performance overhead, and measuring the overhead without that component. First, we remove the locks for synchronization in the implementation of µSWITCH. Then, we further remove MPK memory protection from our implementation. By comparing the time of different variants, we get a decomposition of the invocation latency to show where the overhead comes from. The results are in Table 3. This experiment shows that 23% of the overhead stems from lock primitives required to synchronize memory accesses to the internal data structures. 37% stems from MPK's memory protection.

System Call Cost. Next, we evaluate the sandbox invocation overhead when the sandboxed function invokes system calls. We sandbox a function invoking the `clock_gettime` syscall and then returns the sum of two integers. We first measure the invocation latency of only invoking this function, which we denote as *1 Syscall* in Table 2. Then, we measure the invocation latency of invoking system calls, both, in the sandboxed function and the privileged function by invoking another `clock_gettime` system call on return to the privileged context, which we denote as *2 Syscalls*. As a result, every system call switches the µContext in the kernel. Even when every system call requires switching the kernel resources, the performance of implicit switching is still better than explicit switching, since each context switch additionally needs a system call to switch the kernel context. Implicit context switching batches both the system call and the context switch into the next actual system call.

8.2. Use Case: Firefox

8.2.1. Media Decoding and Gzip Decompression

We isolated four important libraries used by Firefox, libjpeg, libpng, zlib and libvpx, into their own sandboxes and depict the results in Figure 6. Each of these libraries are additionally guarded with Seccomp filters, similar to how they are used in Firefox's sandboxed render processes. These libraries perform image decoding and data decompression. To compare with RLBox, we adapted RLBox to use µSWITCH. RLBox's original implementation uses NaCl [44], requiring us to backport Firefox's WebAssembly to RLBox's original implementation. For both the image decoding experiments, we use varying image resolutions (240x160, 480x320, and 1920x1280) with file sizes of 24, 72, 523 KiB for JPEG

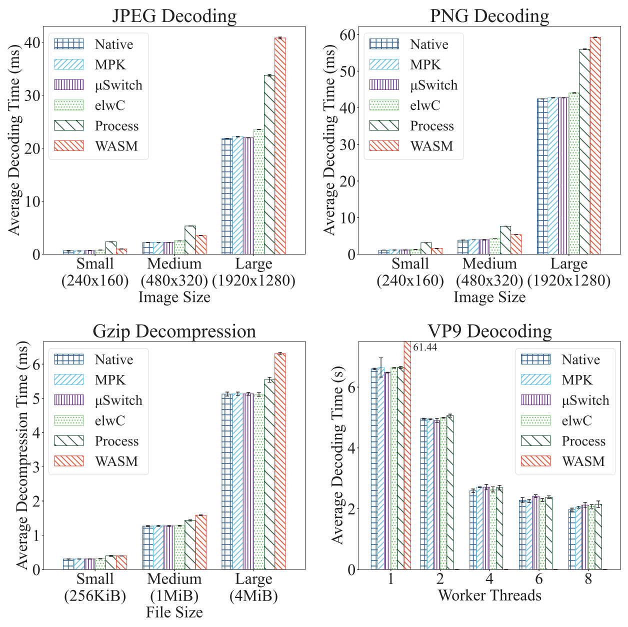


Figure 6: Decoding and decompression time for four libraries used by Firefox. In VP9, WebAssembly does not support beyond 1 worker thread.

images, and 57, 171, 1223 KiB, for PNG images, respectively. We measure the performance of Gzip decompressing three different file sizes (256 KiB, 1 MiB and 4 MiB). µSWITCH causes significantly less overhead (4.7% at maximum and 1.6% on average over native) when isolating libjpeg, libpng, and zlib when compared to other isolation techniques. In contrast, the overhead for the WebAssembly and the process-based isolation techniques range from 8.1% to 255%.

We evaluate µSWITCH's multithreading performance by isolating libvpx, a VP9 video decoding library. libvpx decodes a video file (4096x1714, 2.52 MiB) while varying the number of worker threads (1, 2, 4, 6 and 8). Since the WebAssembly runtime does not support multi-threading, we cannot report results beyond 1 worker thread. During our initial experiments (not reported here), we found that the `memcpy()` used in Musl causes a substantial performance degradation. As a result, we switched the `memcpy()` implementation to the one used by glibc. Compared to native, µSWITCH has low overhead compared with the native throughout all for different isolated libraries and always outperforms its competitor isolation techniques, WebAssembly- and process-based isolation.

We record and show the number of sandboxcalls, privcalls, and system calls invoked during decoding or decompression in Table 4. While most libraries perform a high number of sandboxcalls or privcalls, they perform only a handful system calls. In these cases, µSWITCH skips switching the kernel contexts and incurs no additional overhead apart from the MPK domain switches. This use case highlights the benefits of implicit context switching. Since these libraries use a limited set of kernel resources, unlike the browser rendered running in the privileged domain, we can apply a nearly deny-all Seccomp filter policy to the libraries. However, we note that applying this deny-all Seccomp filter policy requires fine-grained system call isolation. In particular, µSWITCH is crucial to switch the Seccomp filter efficiently. µSWITCH applies this fine-grained system call isolation by setting such filter per µContext. Other MPK-based systems, such as ERIM or Endokernel, cannot achieve this.

8.2.2. Page Loading

We measure the page load latency and resources utilization of µSWITCH and related techniques. For each web page, we define the page load latency as the total time of rendering since loading it. We isolate the same three libraries (libjpeg, libpng, and zlib) in each technique.

Page Load Latency. In web browser experiments most of the latency stems from the network and highly depends on network latency and contention. For reproducibility, we avoid depending on the unstable network by creating clones of each page locally and load the local, cloned pages during our benchmarks. We use four popular websites as our test cases: `bbc.com`, `amazon.com`, `getty.com` and `nytimes.com`. We additionally create three synthetic pages with a varying number of JPEG images (1, 10, or 100). Each experiment loads the page 200 times with the Talos testing framework of Firefox [50] Since Firefox reuses the same renderer process when reloading a page, sandboxes would be reused lowering the number of sandbox creations. Instead, we ensure all created sandboxes are destroyed before Firefox reloads the page. We report the median and the standard deviation of the measured page load latency.

In our evaluation we differentiate between a per-origin and a per-component isolation policy. As described in §7.1, the per-origin isolation policy reuses sandboxes for each network origin, and the per-component isolation policy assigns a dedicated sandbox for each web component (e.g., image). The results, shown in Figure 7, illustrate that µSWITCH outperforms the other isolation techniques for most workloads, incurring an average overhead of 1.5% for per-origin isolation and 5.5% for per-component isolation. For comparison, the average overhead of per-origin isolation and per-component isolation is 23.2% and 18.9% for WebAssembly-based sandbox and 61.7% and 346.4% for process-based sandbox. For `getty.com` with per-component isolation µSWITCH performs worse than the WebAssembly-based sandbox with 27.1% and 2.5% page load overheads, respectively. We notice that WebAssembly performs better with per-component isolation because it does not support multi-threading and requires mutual exclusion for all sandboxcalls in the case of per-origin isolation where multiple threads may invoke the same sandbox.

Number of Calls. We record the numbers of sandboxcalls, privcalls, and syscalls in Table 5 for each page load to illustrate the inner workings of µSWITCH. In case of µSWITCH per-origin isolation, we find that the frequency of system calls invoked by the sandboxed function is extremely low and are mainly performed by the per-origin heap allocator for synchronization across multiple renderer threads. In case of the per-component isolation, each component has its own sandbox with its own heap allocator, thus eliminating the need for any synchronization. Therefore, the number of system calls in per-component isolation is zero for all websites, but `bbc.com`. On the other hand, we observe between 3.5 to 61 times more sandbox creations, up to 2.5 times more sandboxcalls, and up to 1.5 times more privcalls.

TABLE 4: Media decoding and gzip decompression statistics. *Sandboxcalls* are invoked from the main program. *Privcalls* and *Syscalls* are invoked from the sandboxed functions.

	libjpeg (Large)	**libpng (Large)**	**zlib (Large)**	**libvpx (8 cores)**
Sandboxcalls	1295	15	11	2188
Privcalls	0	1282	0	16
Syscalls	0	0	0	10

TABLE 5: Page loading statistics. All the numbers are measured in one page loading cycle. *Sandboxes* is the times of creating a new sandbox or resetting the memory of an old sandbox. Each cell contains numbers for the per-origin / per-component setting.

	Sandboxcalls	**Privcalls**	**Syscalls**	**Sandboxes**
bbc.com	4578 / 4714	210 / 210	9 / 3	4 / 75
amazon.com	13441 / 13521	1024 / 1018	10 / 0	4 / 246
getty.com	3345 / 5132	158 / 200	10 / 0	2 / 70
nytimes.com	6488 / 16347	2459 / 3783	10 / 0	4 / 134
JPEG100	8412 / 7583	319 / 310	8 / 0	3 / 121
JPEG10	4128 / 4173	124 / 124	2 / 0	3 / 29
JPEG1	697 / 708	79 / 79	1 / 0	3 / 11

Memory Usage. We evaluate the memory usage for each websites. We start a fresh Firefox instance for each experiment and load the web page then collect the peak memory usage with `cgmemtime` of Firefox and all its child processes in terms of RSS and cache memory, and report the median and the standard deviation as error bars in Figure 8 over 10 runs. µSWITCH additionally uses 4.9% more memory during peak memory load compare to the baseline. This is similar to process-based isolation or elwC and higher than WebAssembly or pure MPK, which are 4.7%, 11.9%, 0.8% and 0.4% respectively.

8.3. Use Case: HTTP Server

We evaluate the performance overhead of µSWITCH on the HTTP server to measure the efficiency of µSWITCH over other isolation techniques on applications that extensively use kernel resources. As WASI does not support this use case because it does not support network system calls, we only compare µSWITCH with the native baseline, process isolation, pure MPK, and elwC.

We use the HTTP performance benchmark tool `bombardier` to measure the requests the the server handles per second with different file sizes and isolation mechanisms. The results are shown in Figure 9. µSWITCH causes significantly less overhead (14.3% at maximum and 5.8% on average over native) than both elwC (39.0%, 19.4%) and

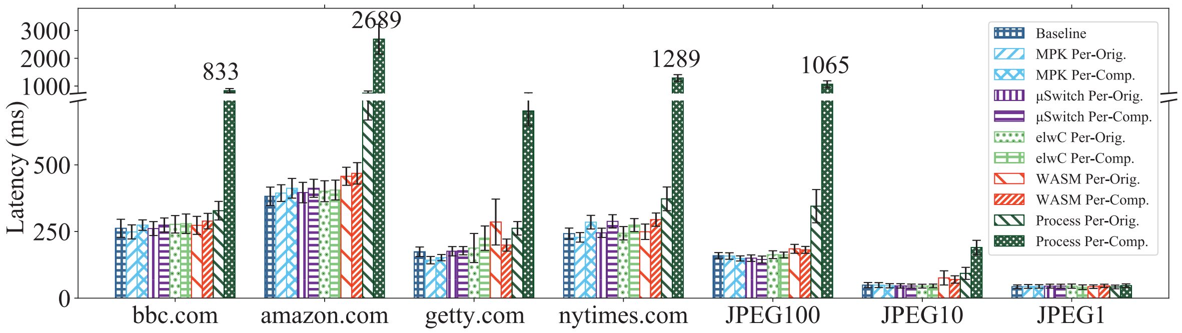


Figure 7: Web Page loading latency for the three isolation techniques, for per-origin and per-component configurations.

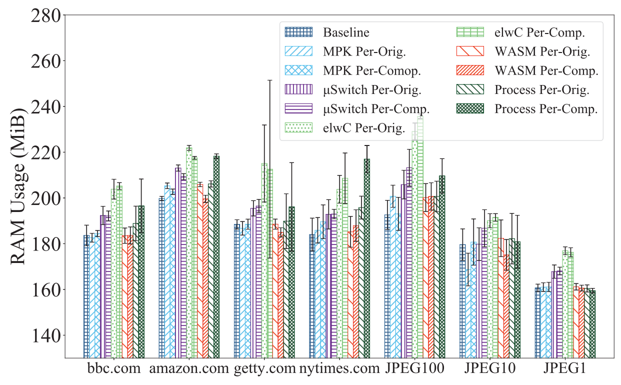


Figure 8: Memory utilization for web page loads.

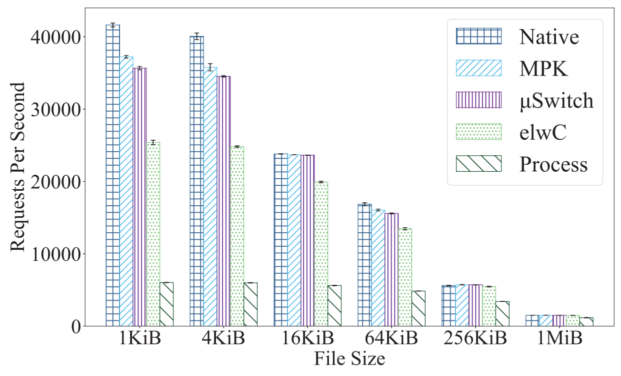


Figure 9: Throughput of the HTTP server with different file sizes and isolation techniques. (Higher is better.)

process isolation (85.5%, 63.3%), and is close to pure MPK (10.7%, 4.1%). When the file size is 1 KiB, the throughput with µSWITCH is 1.40× the one with elwC and 5.91× the one with process isolation respectively. Hence, µSWITCH achieves both high coverage of kernel services, unlike WASI or other in-process kernel context virtualization, and low overhead.

9. Security Analysis

We analyze µSWITCH's security guarantee in this section to understand how µSWITCH achieves similar isolation as process-based isolation.

Memory Isolation. In our current implementation, µSWITCH uses Intel® MPK for memory isolation. Intel® MPK provides hardware-guaranteed memory protection. If a sandboxed function tries to access the memory of the privileged context or other unprivileged contexts, the `SIGSEGV` signal will be raised and the process will be terminated. We analyze several possible methods of bypassing memory protection. First, the attacker can make use of the `WRPKRU` or `XRSTOR` instructions. However, as discussed in §5.2, only safe occurrences of them are allowed. We will later show that even if the attacker can jump to these instructions it is impossible to escape from the isolation. Second, the attacker can make use of system calls [9]. However, we will later show that the problem can be addressed by fine-grained system call filtering. Finally, the attacker may use transient execution attacks such as Spectre and Meltdown. In recent Intel® CPUs, Intel® MPK prevents certain transient execution attacks [51]. These attacks are out of the scope of this paper, and we expect that existing defenses [26] can be applied here.

Safe Occurrences of `WRPKRU` and `XRSTOR`. We first show that the `WRPKRU` instructions for switching from the privileged contexts to the unprivileged contexts are safe. If the attacker jumps to `WRPKRU` with `%eax` being a malicious value, the instructions after `WRPKRU` will immediately compare `%eax` with the correct PKRU value stored in a trusted read-only variable and terminate the program if they do not match. As the attacker does not have the permission to overwrite this variable, any attempt to overwrite PKRU in this case will cause termination of the program. In our implementation, the trusted read-only variable is stored in a fixed offset of the `%fs` segment. To prevent the attacker from tampering with the base address of the `%fs` segment, we forbid the `WRFSBASE` instruction when doing binary inspection. Also, we disable the system call `arch_prctl` as it can be used to overwrite the base address. Then, we show that `WRPKRU` instructions for switching from the unprivileged contexts to the privileged context are safe. The attacker jumping to `WRPKRU` will be forced to jump to the trusted entrypoint. Then, the control flow will continue as if a normal exit or privcall happens, which is not harmful. Finally, if the attacker tries to use `XRSTOR` to overwrite PKRU, the bit flag for restoring PKRU should be one in `%eax`. However, the instructions following `XRSTOR` will defensively terminate the program in this case.

Kernel Context Isolation. Now we discuss the security guarantee of µSWITCH for the isolated kernel resources. When the sandboxed function enters the kernel by invoking a system call, all isolated kernel resources will be updated and only the resources in the current kernel context are visible to the program. Thus, we can prevent the sandboxed function from accessing the kernel resources of other µContexts if we can protect the shared-descriptor, which is guaranteed by MPK. One important kernel resource µSWITCH isolates is the Seccomp filter. By filtering system calls in a fine-grained manner, we ensure that the attacker cannot exploit some system calls to escape isolation while the privileged context can use those system calls normally.

10. Discussion

Supported Resources. µSWITCH can isolate all kernel resources that can only be accessed via system calls. For example, file descriptors fall in this category while process memory layout does not. Hence, our implementation can be extended to other kernel resources.

Number of Memory Domains. A limitation of µSWITCH with MPK is the number of concurrent µContext's per address space as Intel® MPK only supports a maximum of 16 different memory domains in one address space, though our experiments with Firefox and the HTTP server did not run into this limitation. µSWITCH can be applied on architectures other than x86-64, which have more protection keys [40]. Also, µSWITCH can be used with other memory protection techniques that do not have this limitation, such as WebAssembly.

Refactoring Effort. µSWITCH assumes a refactoring effort from the developers . The problem of automatically applying sandboxes orthogonal to µSWITCH. µSWITCH provides a secure and fast isolation mechanism and can be combined with prior work to automatically isolate applications as demonstrated with our RLBox integration.

Use Cases. µSWITCH provides an efficient and practical intra-process isolation system that benefits various workloads. For instance, compared with process isolation, µSWITCH safely co-locates untrusted libraries and the program within the same address space, improving the fork-based snapshotting [52, 53] performance. Moreover, µSWITCH can be extended to support workloads where IPC overhead is non-trivial, e.g., serverless computing [52, 54–57]. We leave this as future work.

11. Related Work

In-process Memory Isolation. Different in-process memory isolation mechanisms have been adopted for sandboxing. SFI [8, 44, 57–59] enforces memory isolation by instrumenting memory operations. Recently, WebAssembly [13] has gained much attention as a general sandboxing platform. Hardware-assisted enforcement is often used to provide secure and efficient memory isolation. Such approach includes traditional memory management [5] and emerging hardware extensions, such as Intel® VT-x [23, 60, 61], Intel® MPK [4, 39, 62–67], ARM memory domains [68], Intel® SGX [69, 70], Intel® CET [71], and special hardware [72–74].

Many works study the safe usage of memory protection keys [4, 9, 39, 40, 75–77]. ERIM [4] proposes the design of call gates and binary rewriting to ensure the safe update of the PKRU register. PKU pitfalls [9] explores several attacks on Intel® MPK-based memory isolation mechanisms. Endokernel [19], Jenny [41], and Cerberus [42] add the isolation of kernel resources apart from MPK-based memory protection. Different from Endokernel or Jenny, which achieves the safe usage of MPK by auditing system calls with in-process trusted domain or other special handling, µSWITCH leverages fine-grained system call filtering by isolating kernel contexts, which is conceptually simpler and less error-prone.

Kernel Context Isolation. Intra-process isolation [68, 78] provides different views over memory or kernel resources for code and data within the same process. For instance, lwC [6] proposes *light-weight context*, an isolation unit that is fully decoupled from the scheduling unit. Orthogonal to our work are techniques to automatically defining isolation boundaries [79]. µSWITCH isolates a series of kernel resources and optimizes intra-process context switch latency by proposing implicit context switch.

Fast Kernel Context Switch. Process isolation could provide strong isolation but suffers from high overhead due to context switching. Providing new user space primitives to optimize process/thread scheduling is an active research field [80–83]. Prior work mainly focuses on reducing the process context switch overhead, but it still involves an explicit context switch, while µSWITCH proposes implicit context switching.

12. Conclusion

This paper presents µSWITCH, a system to efficiently isolate kernel resources and application memory into fine-grained µContexts. Using a lightweight MPK-based memory isolation technique, µSWITCH is able to perform an *implicit* switch of the kernel resources from user space, thus achieving high security guarantees with minimum performance overhead. We used µSWITCH to isolate libraries in the Firefox web browser and an HTTP server, and reduced the overhead of isolation by 32.7% to 98.4% compared with other isolation techniques.

Acknowledgment

We thank the anonymous reviewers and our shepherd for their feedback, which greatly improved the paper. We are also grateful to Michael LeMay for the insightful discussions. This work was funded in part by NSF under grants CNS-2140305 and CNS-2145888, and Intel.

References

[1] Google, "Site isolation – the chromium projects," https://www.chromium.org/Home/chromium-security/site-isolation/.

[2] C. Reis, A. Moshchuk, and N. Oskov, “Site isolation: Process separation for web sites within the browser,” in *Proceedings of the USENIX Conference on Security Symposium*, 2019.

[3] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan, “Retrofitting fine grain isolation in the Firefox renderer,” in *Proceedings of the USENIX Conference on Security Symposium*, 2020.

[4] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, “ERIM: Secure, efficient in-process isolation with protection keys (MPK),” in *Proceedings of the USENIX Conference on Security Symposium*, 2019.

[5] T. C.-H. Hsu, K. Hoffman, P. Eugster, and M. Payer, “Enforcing least privilege memory views for multithreaded applications,” in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[6] 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 *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2016.

[7] J. Corbet, “Memory protection keys,” 2015, https://lwn.net/Articles/643797/.

[8] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, “Efficient software-based fault isolation,” *SIGOPS Oper. Syst. Rev.*, vol. 27, no. 5, p. 203–216, dec 1993.

[9] R. J. Connor, T. McDaniel, J. M. Smith, and M. Schuchard, “PKU pitfalls: Attacks on PKU-based memory isolation systems,” in *Proceedings of the USENIX Conference on Security Symposium*, 2020.

[10] The MITRE Corporation, “CVE-2017-14632,” https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1463.

[11] ——, “CVE-2019-9232,” https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-9232.

[12] B. Holley-Mozilla, “WebAssembly and Back Again: Fine-Grained Sandboxing in Firefox 95,” 2019, https://hacks.mozilla.org/2021/12/webassembly-and-back-again-fine-grained-sandboxing-in-firefox-95/.

[13] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, “Bringing the web up to speed with webassembly,” *SIGPLAN Not.*, vol. 52, no. 6, p. 185–200, jun 2017.

[14] A. Ahmad, S. Lee, P. Fonseca, and B. Lee, “Kard: Lightweight data race detection with per-thread memory protection,” in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[15] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, “libmpk: Software abstraction for Intel Memory Protection Keys (Intel MPK),” in *Proceedings of the USENIX Conference on USENIX Annual Technical Conference (ATC)*, 2019.

[16] V. A. Sartakov, L. Vilanova, and P. Pietzuch, “CubicleOS: A library OS with software componentisation for practical isolation,” in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[17] X. Wang, S. Yeoh, P. Olivier, and B. Ravindran, “Secure and efficient in-process monitor (and library) protection with Intel MPK,” in *Proceedings of the European Workshop on Systems Security (EuroSec)*, 2020.

[18] L. Clark, “Standardizing WASI: A system interface to run WebAssembly outside the web,” 2019, https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface.

[19] B. Im, F. Yang, C.-C. Tsai, M. LeMay, A. Vahldiek-Oberwagner, and N. Dautenhahn, “The endokernel: Fast, secure, and programmable subprocess virtualization,” *arXiv preprint arXiv:2108.03705*, 2021.

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

[21] M. Kerrisk, “namespaces(7) — linux manual page,” https://man7.org/linux/man-pages/man7/namespaces.7.html.

[22] ——, “cgroups(7) — linux manual page,” https://man7.org/linux/man-pages/man7/cgroups.7.html.

[23] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, “Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation,” in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[24] N. C. Wanninger, J. J. Bowden, K. Shetty, A. Garg, and K. C. Hale, “Isolating functions at the hardware limit with virtines,” in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2022.

[25] M. Kerrisk, “seccomp(2) — linux manual page,” https://man7.org/linux/man-pages/man2/seccomp.2.html.

[26] J. Behrens, A. Cao, C. Skeggs, A. Belay, M. F. Kaashoek, and N. Zeldovich, “Efficiently mitigating transient execution attacks using the unmapped speculation contract,” in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2020.

[27] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, “Sel4: Formal verification of an os kernel,” in *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2009.

[28] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy, “An empirical study on the correctness of formally verified distributed systems,” in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2017.

[29] L. Nelson, H. Sigurbjarnarson, K. Zhang, D. Johnson,

J. Bornholt, E. Torlak, and X. Wang, "Hyperkernel: Push-button verification of an os kernel," in *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2017.

[30] E. Gershuni, N. Amit, A. Gurfinkel, N. Narodytska, J. A. Navas, N. Rinetzky, L. Ryzhyk, and M. Sagiv, "Simple and precise static analysis of untrusted linux kernel extensions," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2019.

[31] S. Gong, D. Altinbüken, P. Fonseca, and P. Maniatis, "Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis," in *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2021.

[32] P. Fonseca, R. Rodrigues, and B. B. Brandenburg, "SKI: Exposing kernel concurrency bugs through systematic schedule exploration," in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2014.

[33] C. Liu, S. Gong, and P. Fonseca, "KIT: Testing os-level virtualization for functional interference bugs," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[34] M. Abubakar, A. Ahmad, P. Fonseca, and D. Xu, "SHARD:Fine-Grained kernel specialization with Context-Aware hardening," in *Proceedings of the USENIX Conference on Security Symposium*, 2021.

[35] Canonical Ltd, "Linux containers," https://linuxcontainers.org/.

[36] Containers Organization, "Podman," https://podman.io.

[37] Google, "Open-sourcing gvisor, a sandboxed container runtime," https://cloud.google.com/blog/products/identity-security/open-sourcing-gvisor-a-sandboxed-container-runtime.

[38] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not so fast: Analyzing the performance of WebAssembly vs. native code," in *Proceedings of the USENIX Conference on USENIX Annual Technical Conference (ATC)*, 2019.

[39] 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 *Proceedings of the USENIX Conference on USENIX Annual Technical Conference (ATC)*, 2019.

[40] 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 *Proceedings of the USENIX Conference on Security Symposium*, 2020.

[41] D. Schrammel, S. Weiser, R. Sadek, and S. Mangard, "Jenny: Securing syscalls for PKU-based memory isolation systems," in *Proceedings of the USENIX Conference on Security Symposium*, 2022.

[42] A. Voulimeneas, J. Vinck, R. Mechelinck, and S. Volckaert, "You shall not (by)pass! practical, secure, and fast pku-based sandboxing," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2022.

[43] S. Narayan and C. Disselkoen, "Rlbox github repository," https://github.com/shravanrn/LibrarySandboxing.

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

[45] The MITRE Corporation, "CVE-2019-0211," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-0211.

[46] Nick Mathewson, "Sample http server," https://github.com/libevent/libevent/blob/master/sample/http-server.c.

[47] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, pp. 1278–1308, 1975.

[48] Intel, "Intel architecture instruction set extensions and future features," https://www.intel.com/content/www/us/en/develop/download/intel-architecture-instruction-set-extensions-programming-reference.html.

[49] C. S. Pabla, "Completely fair scheduler," *Linux Journal*, vol. 2009, p. 4, 2009.

[50] Mozilla, "Talos," https://wiki.mozilla.org/TestEngineering/Performance/Talos.

[51] Intel, "Intel analysis of speculative execution side channels," https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf.

[52] X. Wei, F. Lu, T. Wang, J. Gu, Y. Yang, R. Chen, and H. Chen, "No provisioned concurrency: Fast RDMA-codesigned remote fork for serverless computing," 2022.

[53] K. Zhao, S. Gong, and P. Fonseca, "On-demand-fork: A microsecond fork for memory-intensive and latency-sensitive applications," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2021.

[54] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Menezes Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing," Tech. Rep. UCB/EECS-2019-3, Feb 2019.

[55] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ml workflows," in *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, 2019.

[56] J. Carreira, S. Kohli, R. Bruno, and P. Fonseca, "From warm to hot starts: Leveraging runtimes for the serverless era," in *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2021.

[57] A. Vahldiek-Oberwagner and M. Vij, "Meshwa: The case for a memory-safe software and hardware architecture for serverless computing," *Proceedings of the Workshop On Resource Disaggregation and Serverless Computing (WORDS)*, 2022.

[58] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, "Fast byte-granularity software fault isolation," in *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*.

[59] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software guards for system address spaces," in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2006.

[60] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe user-level access to privileged CPU features," in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012.

[61] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested kernel: An operating system architecture for intra-kernel privilege separation," *SIGARCH Comput. Archit. News*, vol. 43, no. 1, p. 191–206, mar 2015.

[62] A. Ghosn, M. Kogias, M. Payer, J. R. Larus, and E. Bugnion, "Enclosure: Language-based restriction of untrusted libraries," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[63] P. Kirth, M. Dickerson, S. Crane, P. Larsen, A. Dabrowski, D. Gens, S. Volckaert, and M. Franz, "PKRU-safe: Automatically locking down the heap between safe and unsafe languages," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2022.

[64] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No need to hide: Protecting safe regions on commodity hardware," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2017.

[65] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu, "Faastlane: Accelerating Function-as-a-Service workflows," in *Proceedings of the USENIX Conference on USENIX Annual Technical Conference (ATC)*, 2021.

[66] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis, "xMP: Selective memory protection for kernel and user space," in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2020.

[67] J. Gu, H. Li, W. Li, Y. Xia, and H. Chen, "EPK: Scalable and efficient memory protection keys," in *Proceedings of the USENIX Conference on USENIX Annual Technical Conference (ATC)*, 2022.

[68] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu, "Shreds: Fine-grained execution units with private memory," in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2016.

[69] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer, and P. Pietzuch, "Glamdring: Automatic application partitioning for Intel SGX," in *Proceedings of the USENIX Conference on USENIX Annual Technical Conference (ATC)*, 2017.

[70] A. Ahmad, J. Kim, J. Seo, I. Shin, P. Fonseca, and B. Lee, "CHANCEL: efficient multi-client isolation under adversarial programs," in *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2021.

[71] M. Xie, C. Wu, Y. Zhang, J. Xu, Y. Lai, Y. Kang, W. Wang, and Z. Wang, "Cetis: Retrofitting intel cet for generic and efficient intra-process memory isolation," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[72] T. Frassetto, P. Jauernig, C. Liebchen, and A.-R. Sadeghi, "IMIX: In-Process memory isolation EXtension," in *Proceedings of the USENIX Conference on Security Symposium*, 2018.

[73] B. Davis, R. N. M. Watson, A. Richardson, P. G. Neumann, S. W. Moore, J. Baldwin, D. Chisnall, J. Clarke, N. W. Filardo, K. Gudka, A. Joannou, B. Laurie, A. T. Markettos, J. E. Maste, A. Mazzinghi, E. T. Napierala, R. M. Norton, M. Roe, P. Sewell, S. Son, and J. Woodruff, "Cheriabi: Enforcing valid pointer provenance and minimizing pointer privilege in the posix c run-time environment," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[74] M. LeMay, J. Rakshit, S. Deutsch, D. M. Durham, S. Ghosh, A. Nori, J. Gaur, A. Weiler, S. Sultana, K. Grewal, and S. Subramoney, "Cryptographic capability computing," in *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021.

[75] H. Lefeuvre, V.-A. Bădoiu, c. Teodorescu, P. Olivier, T. Mosnoi, R. Deaconescu, F. Huici, and C. Raiciu, "FlexOS: Making os isolation flexible," in *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2021.

[76] L. Delshadtehrani, S. Canakci, W. Blair, M. Egele, and A. Joshi, "Flexfilt: Towards flexible instruction filtering for security," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2021.

[77] S. Gravani, M. Hedayati, J. Criswell, and M. L. Scott, "Fast intra-kernel isolation and security with IskiOS," in *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2021.

[78] T. Kim and N. Zeldovich, "Practical and effective sandboxing for non-root users," in *Proceedings of the USENIX Conference on USENIX Annual Technical Conference (ATC)*, 2013.

[79] A. Bittau, P. Marchenko, M. Handley, and B. Karp, "Wedge: Splitting applications into Reduced-Privilege compartments," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[80] Google Inc., "UMCG early preview/RFC patchset," https://lore.kernel.org/lkml/

20210520183614.1227046-1-posk@google.com.
[81] M. Hao, H. Li, M. H. Tong, C. Pakha, R. O. Suminto, C. A. Stuardo, A. A. Chien, and H. S. Gunawi, "Mittos: Supporting millisecond tail tolerance with fast rejecting slo-aware os interface," in *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2017.
[82] J. T. Humphries, N. Natu, A. Chaugule, O. Weisse, B. Rhoden, J. Don, L. Rizzo, O. Rombakh, P. Turner, and C. Kozyrakis, "Ghost: Fast & flexible user-space delegation of linux scheduling," in *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2021.
[83] K. Kaffes, J. T. Humphries, D. Mazières, and C. Kozyrakis, "Syrup: User-defined scheduling across the stack," in *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2021.
[84] musl libc, https://git.musl-libc.org/cgit/musl.

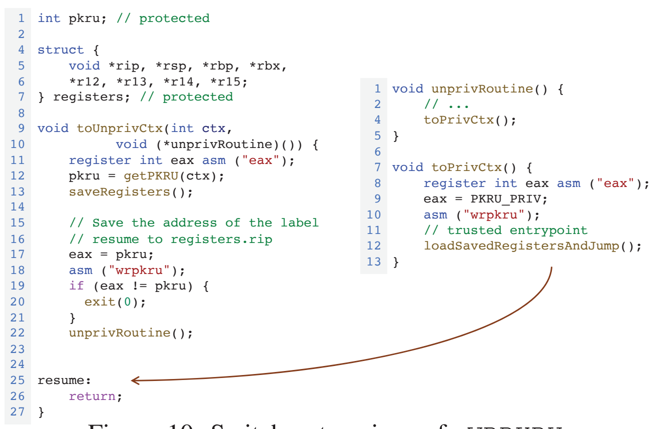
Appendix A. System Call Entrypoint

Algorithm 1 illustrates how the system call entrypoint works. In addition to the shared-descriptor, we add a *next-descriptor* for signal handling discussed in Appendix C.

Algorithm 1 In-process Kernel Context Switch.

```
Input: $\mathcal{P}$: Kernel PCB.
1: if $\mathcal{P}.shared\_desc \neq \mathcal{P}.curr\_desc$ then
2:     $\mathcal{P}.contexts[\mathcal{P}.curr\_desc] \leftarrow \mathcal{P}.resources$
3:     $\mathcal{P}.resources \leftarrow \mathcal{P}.contexts[\mathcal{P}.shared\_desc]$
4:     $\mathcal{P}.curr\_desc \leftarrow \mathcal{P}.shared\_desc$
5: if $\mathcal{P}.next\_desc \neq -1$ then
6:     $\mathcal{P}.shared\_desc \leftarrow \mathcal{P}.next\_desc$
```

Appendix B. Details of µContext Switch Gate



```
int pkru; // protected

struct {
    void *rip, *rsp, *rbp, *rbx,
    *r12, *r13, *r14, *r15;
} registers; // protected

void toUnprivCtx(int ctx,
         void (*unprivRoutine)()) {
    register int eax asm ("eax");
    pkru = getPKRU(ctx);
    saveRegisters();

    // Save the address of the label
    // resume to registers.rip
    eax = pkru;
    asm ("wrpkru");
    if (eax != pkru) {
      exit(0);
    }
    unprivRoutine();


resume:
    return;
}
```

```
void unprivRoutine() {
    // ...
    toPrivCtx();
}

void toPrivCtx() {
    register int eax asm ("eax");
    eax = PKRU_PRIV;
    asm ("wrpkru");
    // trusted entrypoint
    loadSavedRegistersAndJump();
}
```

Figure 10: Switch gate using safe `WRPKRU`.

Figure 10 shows an example of the switch gates using safe `WRPKRU` that are used to switch to both the privileged context and the unprivileged contexts. When the privileged function calls the sandboxed function `unprivRoutine()`, it saves the registers including the stack pointer and other callee-saved registers, and the label `resume` as the exit point in `saveRegisters()`. Then after it switches to the unprivileged memory domain by `WRPKRU`, the value of `%eax` is checked. When the sandboxed function `unprivRoutine()` returns to the privileged context, it calls `toPrivCtx()`, which sets the PKRU register to the value of the privileged context, then jumps to the trusted entrypoint `loadSavedRegistersAndJump()`. `loadSavedRegistersAndJump()` loads the saved registers. Then, it jumps to the label `resume`, the exit point of the privileged function `toUnprivCtx()`, which is illustrated as the arrow in the figure. After that, the privileged function continues its execution.

Appendix C. Signal Handling and Fault Isolation

µSWITCH allows the privileged context to register signal handlers. These signal handlers always execute in the privileged context even if the signals are raised in the unprivileged contexts. However, securely performing signal handling presents several challenges because the default signal handling code in the kernel is not aware of µContexts. Therefore, we insert signal trampolines that insert prologues and epilogues for each signal handler and perform additional steps, as described next.

Access to Unprivileged Domains. By default, before transferring control to the signal handler, the kernel will set the PKRU register to the default value, which only grants the permission to the privileged domain 0. However, in µSWITCH, we want the signal handlers to be able to access data from unprivileged contexts, for example, to inspect and sanitize the arguments when trapping any system calls invoked by the unprivileged contexts. To allow the signal handlers to access all domains, we update the PKRU register in the signal trampoline prologue.

Restoring the shared-descriptor Value on sigreturn. Because the `sigreturn` system call is unaware of µContexts, additional steps are required to restore the shared-descriptor value on a `sigreturn`. The `sigreturn` system call is blocked in the unprivileged contexts. Therefore, when the signal happens in an unprivileged context, the signal handler has to perform an implicit context switch to the privileged context. This requires updating the shared-descriptor value. However, when the signal handler returns via the `sigreturn` system call, all CPU state, including the PKRU register value, are restored. Because the `sigreturn` system call is unaware of the shared-descriptor, it does not restore its value, which will cause a mismatch between the kernel's view of µContext (the shared-descriptor value) and the memory protection domain (PKRU value). The PKRU register will be updated with the correct unprivileged value, but the shared-descriptor will still contain the old, incorrect privileged value.

To address this problem, we add a *next-descriptor*, which stores the value that the shared-descriptor must be updated to when the signal returns. This next-descriptor is also stored in the protected shared page between the kernel and user space. Our signal trampoline epilogue performs two tasks. It (1) updates the next-descriptor with the unprivileged value and (2) updates the shared-descriptor with the privileged value. Then, at the next system call (in this case `sigreturn`), the kernel updates the shared-descriptor with the value of next-descriptor, after validation, i.e., only when next-descriptor is not -1. The -1 value means that shared-descriptor will not be updated at the next system call, which is the typical case for the system calls other than `sigreturn`. This allows the shared-descriptor to be updated with the correct value.

Securing the Signal Frame. When a signal occurs, the signal frame, which contains security-critical register information, must be stored on a dedicated per-thread secure signal stack until the `sigreturn` system call restores the frame. However, when the signal occurs in an unprivileged context, additional steps must be taken to ensure the integrity of the signal frame. As the original call stack is writable by the unprivileged context, we cannot directly set the secure signal stack in the protected memory domain as the signal stack using the `sigaltstack` system call. Instead, we need to switch to the secure signal stack in the the signal trampoline prologue. After switching to the secure signal stack, the signal trampoline prologue also copies the signal frame from the original call stack to the secure signal stack.

Fault Isolation. Fault isolation for sandboxes can be implemented by handling signals such as SIGSEGV, SIGILL, and SIGSYS. Instead of simply terminating the process when a fault occurs, μSWITCH forces the execution to return from the sandboxcall that invoked the μContext. μSWITCH installs signal handlers for these faults. These signal handlers run in the privileged context, and manipulate the signal frame to force a return from the sandboxcall. This ensures the liveness of the program. In other words, if these faults happen in the unprivileged contexts, the privileged function can trap and handle them without terminating the program. We support handling all faults that can be caught by signals including unauthorized memory access or system call, bus error, illegal instruction and floating point error.

To guarantee that the unprivileged functions cannot block the program by a Denial-of-service attacks such as running a infinite loop, we allow the privileged context to set an interval timer and handle the SIGALRM signal to implement timeouts for sandboxcalls. Similarly, transparent system call interception can be implemented by using Seccomp filters and handling SIGSYS signals. This allows μSWITCH to sandbox the binary code containing `syscall` instructions without rewriting the binary.

Appendix D. C Library Isolation

Isolating sandbox functions from privileged functions also requires isolating any library that is shared by both the privileged and sandbox code. While it is possible to isolate most of the libraries into its own sandbox, sandboxing Libc presents further challenges as it is shared by most libraries and our current prototype does not support *sharing* a sandbox with other sandbox contexts. Therefore, instead of sandboxing the Libc library, we provide each sandbox its own *copy* of the Libc library, thus isolating its global and static variables, and thread local storage.

We link each sandbox with its own copy of the lightweight Musl C library [84]. To allow the different copies of the same functions to reside in the same address space and isolate the GOT table, PLT section and global variables, we use `dlmopen` to load each sandbox in its own namespace. Providing each sandbox with its own copy of the C library also simplifies the task of separating the heap memory for each sandbox into its own MPK protection domain. We simply preallocate 1 GiB of memory for each sandbox, protected by its own MPK protection key, and use a dlmalloc-based pool allocator to create heap objects within it. Alternatively, we support a virtualized memory management API, which is discussed later.

Multi-threading. μSWITCH allows a sandbox context to create threads, but it must first switch to the privileged context to do so. We modify the implementation of pthread in Musl to achieve this. We replace the function `pthread_create()` with a function that calls a privcall. The privcall handler in the privileged context will then create a thread and run the given thread routine inside the sandbox on that thread. In this way, all thread creations happen inside the privileged context, and we do not need to expose the `clone` syscall to the unprivileged contexts. For other pthread functions that manipulate threads such as `pthread_join()`, the handling is similar.

However, for multi-threaded programs that use locks or other synchronization mechanisms, trapping to the privileged context constantly can significantly affect the performance. To address this problem, we allow the unprivileged context to freely invoke the `pthread` functions manipulating mutexes and condition-variables, and also allow the sandboxes to invoke the `futex` system call.

Virtualized Memory Management API As μSWITCH depends on memory protection, the system calls for memory management that may break the memory protection need handling. In our system, the privileged context controls the memory management of unprivileged contexts. The unprivileged contexts are not allowed to make any direct system calls for memory management, and can only request the privileged context to allocate memory via privcalls. We modify Musl's implementation of `mmap`, `munmap`, `mremap`, and `mprotect` to achieve this. For security reasons, we block features that can break memory isolation, such as mapping fixed, shared, or executable memory. Upon the invocation of each of these privcalls, the privileged context will first audits the request, before performing the system call on behalf of the unprivileged context. For file-backed mappings, we first duplicate the file from the unprivileged contexts using the `dupFile` method, then map it in the privileged context and finally close it.