Donky: Domain Keys – Efficient In-Process Isolation for RISC-V and x86

ARTIFACT EVALUATED usenix ASSOCIATION PASSED

David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl,
Michael Schwarz, Stefan Mangard, Daniel Gruss
Graz University of Technology

Abstract

Efficient and secure in-process isolation is in great demand, as evidenced in the shift towards JavaScript and the recent revival of memory protection keys. Yet, state-of-the-art systems do not offer strong security or struggle with frequent domain crossings and oftentimes intrusive kernel modifications.

We propose Donky, an efficient hardware-software co-design for strong in-process isolation based on dynamic memory protection domains. The two components of our design are a secure software framework and a non-intrusive hardware extension. We facilitate domain switches entirely in userspace, thus minimizing switching overhead as well as kernel complexity. We show the versatility of Donky in three realistic use cases, secure V8 sandboxing, software vaults, and untrusted third-party libraries. We provide an open-source implementation on a RISC-V Ariane CPU and an Intel-MPK-based emulation mode for x86. We evaluate the security and performance of our implementation for RISC-V synthesized on an FPGA. We also evaluate the performance on x86 and show why our new design is more secure than Intel MPK. Donky does not impede the runtime of in-domain computation. Cross-domain switches are 16–116x faster than regular process context switches. Fully protecting the mbedTLS cryptographic operations has a 4 % overhead.

1 Introduction

Memory isolation is a fundamental building block for developing secure systems. Hence, concepts of memory isolation can be found on all layers in the software stack, e.g., via process isolation via separate address spaces. However, recent use cases demand more fine-grained isolation, especially within a process, where traditional process isolation would incur too substantial performance costs. Especially cloud providers are in the process of abandoning process isolation in favor of language-level sandboxing, e.g., via V8 Isolates [16].

Isolation through the V8 sandbox has use cases in the cloud [16], desktop applications [61], and browsers [81]. Unfortunately, JavaScript engines have a huge potential for vulnerabilities, such as memory corruption, incorrect compiler optimizations, type confusion, or erroneous code generation [33, 68, 70], and strong hardware-backed sandboxing is needed. Similarly, native applications may load untrusted (and potentially closed-source) third-party libraries [78], or use a library for certain secure operations. The principle of least privilege would require isolation of such libraries from the rest of the program. However, traditional process isolation is oftentimes prohibitive in practice. Hence, prior work studied more lightweight in-process isolation techniques [14, 15, 30, 35, 44, 50, 51, 56, 72, 82, 85, 89, 94, 99].

In-process isolation mechanisms range from control flow schemes [30], over capability designs [58, 85, 89], to protection key mechanisms operating on memory pages [15, 82, 99] for various architectures [4, 19, 22, 37, 63]. These designs follow either a security-focused approach (e.g., privileged key switches) with oftentimes significant performance impact or favor performance (e.g., fast key switches) at the cost of reduced security. For instance, Intel MPK [19, 46] is fast but allows manipulations of the MPK access policy and, thus, cannot directly be used as a secure sandbox. Instead, prior work uses binary scanning and non-writable code pages to prevent manipulations (e.g., ERIM [82]), complicating sandboxing just-in-time-compiled JavaScript code. If an attacker gains arbitrary code execution, all MPK-based approaches lose their protection guarantees. Others guard their memory access policy via the kernel, which, while secure, demands costly or intrusive kernel interaction and modifications [15, 35, 50, 99]. Finally, existing architectures are oftentimes limited to 16 protection domains [4, 19], and software emulation of more domains has a substantial performance cost [64].

Since existing solutions have different security and performance goals or involve heavy kernel interaction, we identify the following research question and challenge:
As the objectives of MPK (high performance) and kernel-based approaches (high security) are seemingly contradictory, can these two approaches be combined? How can protection keys be securely and efficiently managed in userspace?

In this paper, we solve this challenge with Donky, a hardware-software co-design providing strong in-process isolation guarantees based on memory protection keys. Donky offers pure userspace policy management with negligible overhead and full backward-compatibility. Memory pages are dynamically assigned to protection domains, providing strict hardware-backed isolation between domains. Moreover, policy management is entirely decoupled from the kernel and instead delegated to a self-protecting userspace monitor. Donky provides substantially stronger security guarantees than previous designs [82], at a low performance cost.

We demonstrate the versatility of Donky in three realistic use cases: First, we augment the JavaScript V8 engine with isolation guarantees that usually can only be achieved by spawning multiple instances of the V8 engine, *i.e.*, process isolation. Second, we isolate a third-party library from the main program, preventing illegitimate access to the main program's data, e.g., a parsing library without full access to the program's address space. Third, we build a software vault using Donky with security guarantees that can usually only be obtained by running the software vault in a separate process.

Our design consists of two components. The first component is a secure software framework to define and handle memory protection domains in userspace, e.g., for just-in-time compiled code or third-party binary code. Its core, a lightweight protection domain monitor library called DonkyLib, exposes Donky functionality, such as secure in-userspace domain switching and modification, to an application developer. We completely outsource system call filtering to a privileged userspace domain to avoid usage of extended Berkeley Packet Filters (eBPFs), which have been used several times for kernel exploitation [77]. Expensive context switches to the kernel are not necessary for switching or modifying protection domains.

The second component of Donky is a small hardware extension. Our full open-source hardware implementation is based on the RISC-V Ariane CPU and evaluated on a Xilinx Kintex-7 FPGA KC705. We also implement an Intel-MPK-based emulation mode for x86. We show that a full Donky implementation provides higher security guarantees than MPK-based schemes currently can provide: Donky has a special userspace protection key policy register protected via a hardware call gate. Consequently, we do not need binary inspection or rewriting to guarantee that malicious code cannot change it, unlike all isolation techniques building upon Intel's current MPK implementation [82], and Donky can shield against arbitrary code execution. We outline hardware changes to Intel MPK for full Donky support.

We provide a thorough performance analysis for our RISC-V-based implementation and also, despite the lower security guarantees, for our emulation mode on x86. We show that the performance cost in both implementations of Donky is negligible when compared to the cost of process isolation and earlier proposals. Finally, we discuss previous work on in-process isolation in detail and find that previous work focused only on some goals of Donky (e.g., only isolating trusted code [15]) or even entirely orthogonal goals like CFI [30].

In summary, our contributions are as follows:

- We propose Donky, efficient userspace memory protection domains, without requiring control-flow integrity, binary inspection, or binary rewriting.
- We provide an open-source implementation[1] on a RISC-V CPU, with higher security than MPK-based schemes.
- We repurpose the RISC-V *extension for user-level interrupts* for managing access policies entirely in userspace.
- We evaluate Donky on V8 just-in-time-compiled JavaScript code and native code. Donky is 1–2 orders of magnitude faster than process-based isolation and shows a negligible overhead over no isolation on real-world software.

Paper Outline. Section 2 provides background on RISC-V and protection keys. Section 3 overviews Donky's design. Section 4 details the software component. Section 5 details the hardware extension. Section 6 evaluates Donky's performance and security. Section 7 qualitatively evaluates Donky in terms of applicability, performance, and security. Section 8 discusses related work, and Section 9 concludes.

2 Background

In this section, we overview RISC-V, virtual memory, existing protection key architectures, and JavaScript JIT engines.

2.1 RISC-V

RISC-V is a free and open-source instruction set architecture (ISA). It comprises the unprivileged ISA [28], and the privileged ISA [27]. A set of control and status registers (CSRs) allows configuring the CPU behavior, access performance metrics, and provides additional scratch space for exception handling. CSRs are typically prefixed with `m`, for machine mode, `s`, for supervisor mode, or `u`, for user mode. Exceptions occur upon various occasions, e.g., memory violations. To handle the exception, the CPU switches to machine mode and jumps to the address specified in the trap-vector base-address register (`mtvec` CSR). Exceptions can be delegated to supervisor mode in the `medeleg` CSR. The instructions `mret` and `sret` are used to return from the exception handler.

RISC-V specifies the so-called "Standard Extension for User-Level Interrupts", also abbreviated as *N extension* [29].[2] The *N extension* is intended for embedded systems, and user mode exception handling (e.g., for garbage collection or integer overflows) is only briefly discussed as a potential use case for non-embedded systems (e.g., Unix). The *N extension* adds the `utvec` and `sedeleg` CSRs, amongst others, to delegate exceptions and interrupts directly to user mode handlers without invoking higher privileged code. As with higher privilege

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

[2] It is currently in draft status for the RISC-V ISA 1.12.

modes, `utvec` allows for vectorized exceptions, and the `uret` instruction is used to return from the handler.

Ariane [1, 96] is a 64-bit single issue, 6-stage, in-order CPU, optimized for short critical path length. It implements the RV64IMAC RISC-V ISA and features the M, S, and U privilege modes. Ariane implements v1.10 of the privileged and the working draft of the unprivileged RISC-V ISA v2.3. Thus, it can run Unix-like operating systems.

2.2 Address Translation

Modern 64-bit CPUs typically support 48-bit (recently also 57-bit) virtual address spaces, used for process isolation. For virtual-to-physical address translation, address spaces are mapped in blocks of pages, most commonly 4 KiB. Modern CPUs support multiple levels of translation tables, which are stored in memory. Their entries (also called page-table entries) are cached in the so-called translation-lookaside buffer (TLB). Switching between processes, and thus address spaces, means updating a CPU register to point to a different set of translation tables and flushing the TLB unless it is tagged with an address-space identifier. Via the page-table entries (PTEs), access permissions are managed per page, such that the same physical page may be mapped in multiple virtual address spaces (*i.e.*, multiple processes, shared memory), even with different access permissions. Updates to permissions, mappings, or the switching of the address space can only be done by the kernel. Hence, context switches are required for any of these operations to isolate contexts (e.g., processes) from each other.

2.3 Memory Protection Keys

Memory protection keys are an extension to page-based memory permissions, allowing to change permissions of memory ranges without the slow kernel-level modification of page tables. Instead, page-table entries are tagged with a protection key, but the permissions (which the hardware enforces) for these keys are stored separately. Keys are usually associated with a protection domain (e.g., application, library, module), and each (typically virtual) memory region can have one associated key. Processes can have one or more keys assigned (e.g., one key per application on System/360) via special registers.

Today's implementations differ mainly in the number of loaded keys per thread and process, the types of permissions, if the protection key policy register is privileged or not, as well as memory region granularity. The main differences of protection key implementations of some notable hardware architectures are as follows:

Intel's Memory Protection Keys (MPK) [19] use 4-bit keys stored in the page-table entry, allowing for 15 different domains per process. The corresponding read- and write-disable bits for each key are stored in the `PKRU` (User Page Key Register) and checked by the hardware upon access. As the `PKRU` is non-privileged, allowing fast domain-switching in userspace, MPK itself does not provide secure in-process isolation and, to obtain such, has to be combined with other mechanisms (such as CFI and binary scanning).

ARM Memory Domains [4] are defined in ARMV8 for AArch32 but were dropped in AArch64. They use 4-bit domain IDs (keys) in the translation tables and a kernel-mode Domain Access Control Register (DACR) with a 2-bit field per key. With DACR, access can either be denied, enforced at PTE level, or fully allowed, bypassing PTE permissions. Since only the first-level page-table entries contain domain IDs, domain boundaries must be aligned at 1 MB blocks.

IBM's Power [37] architecture supports 5-bit protection keys, allowing 32 different memory domains. Its privileged (kernel mode) registers (AMR and IAMR) store read, write, and execute permissions for each key.

HP PA-RISC [63] uses 15–18-bit "protection identifiers" with a write-disable bit each stored in privileged control registers. Instead of storing a write-disable bit for each of the keys (which would require a 2^{18} bit register), they have four registers to load one key each.

Itanium (IA-64) [22] is very similar to PA-RISC but provides (at least) 16 registers with 18–24-bit keys each and have additional read- and execute-disable bits as well as a valid bit.

The above hardware designs have various trade-offs. If the protection key policy register can be changed from the userspace using unprivileged operations, domain transitions can be very fast and do not require any kernel interaction. Having a privileged register, however, completely changes the threat model and possible use cases. In this case, the kernel needs to know about the different memory domains, which requires many complex kernel modifications. Existing work based on Intel MPK works around the inherent problem of malicious protection key policy register modification by utilizing additional mechanisms such as compiler-based code rewriting [41], binary inspection [82] and Write-XOR-Execute to ensure there are no unintended writes to the `PKRU`.

2.4 JIT and JavaScript Engines

Just-in-time compilation (JIT) dynamically compiles interpreted programming languages, e.g., JavaScript, into an intermediate representation (byte code) or machine code. A JavaScript engine manages the tasks of compilation and execution of JavaScript, memory management, and optimization. In the case of V8, which is used in Chrome, Chromium, and Node.js [81], the source code is first compiled into a byte code representation, which is then interpreted and executed. While the code is executed, another component of the engine analyses the runtime and further optimizes the byte code directly into machine code. This requires the code region to be both writable and executable.

Typically, browsers use sandboxing to minimize the attack surface for attackers exploiting vulnerabilities via JavaScript.

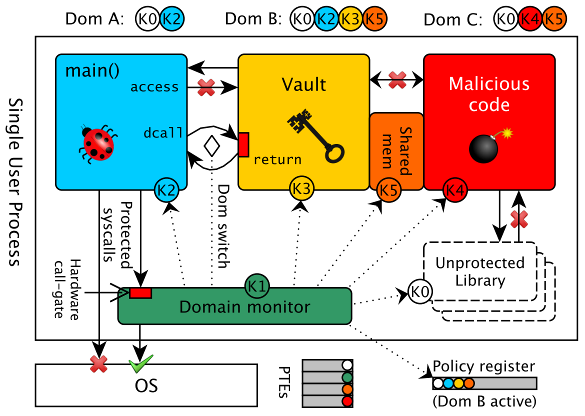

Figure 1: Donky structures a user process into security domains, orchestrating a set of memory regions. Each region is assigned a unique protection key, and access is controlled via a policy register. Keys can be domain-private to implement software vaults (Dom B), or shared across domains. Limiting a domain's keys allows to sandbox malicious code (Dom C). The domain monitor manages protection keys, the policy register, and system call filtering. Call gates prevent control-flow attacks across domains.

E.g., in V8, an *Isolate* is an independent copy of the entire JavaScript runtime environment. Each Isolate has its own code cache, heap, garbage collection, and call stack. Thus, JavaScript code runs in parallel in a separate Isolate within the same process. However, sandbox escapes are still possible by exploiting vulnerabilities in both the JavaScript engine and the sandbox [2,33,70]. An additional security enhancement is to use process isolation, e.g., in the form of site isolation [67].

3 Donky System Design

In this section, we define our threat model and present Donky, a hardware-software co-design for strong and efficient memory isolation within a single user process. Donky provides highly flexible and lightweight domains atop of hardware-backed memory protection keys, as visualized in Figure 1.

Threat model. Donky supports complex user programs with multiple software modules and mixed trust assumptions (cf. Figure 1). Modules can range from small components like individual C++ classes over compounds like plugins or browser tabs to entire binaries and libraries. For the sake of demonstration, we discuss two common scenarios.

First, in a sandbox scenario, an application wants to execute untrusted code modules without specific security assumptions. They may contain vulnerabilities that are actively exploited by an adversary, or even run malicious (e.g., user-provided JavaScript) or arbitrary code, such that it issues adversary-chosen system calls or accesses adversary-chosen memory locations. The adversary may repeatedly inject arbitrary instructions at runtime, including `WRPKRU`. The application encapsulates this untrusted code in a Donky in-process sandbox. Donky shields not only application memory and sandbox transitions but also the system call interface at the discretion of the application. In contrast to ERIM [82], we do not require binary scanning. Also, Donky does not rely on recompiling programs with CFI. Instead, Donky can sandbox unmodified, pre-compiled binaries. Unlike ERIM, we do not assume Write-XOR-Execute and also support self-modifying code. This enables use cases such as JIT compilation, one of the main applications of Donky, *without* modifying the JIT compiler to not emit unsafe `WRPKRU` instructions.

Second, in a vault scenario, an application wants to shield highly sensitive modules such as cryptographic libraries. While not being adversarial, the application wants to enforce the principle of least privilege [69] to reduce the attack surface in case of corruption. For example, the application might be subject to vulnerabilities and exploitation. It might also load other modules (e.g., libc), which themselves are vulnerable or malicious and cannot be securely sandboxed. The application shields sensitive modules in a Donky in-process vault and renounces all access rights to it. Donky enforces memory isolation and call gate protection towards the vault.

We assume that the developer correctly uses Donky. Ill-designed trust relationships, domain interfaces, or system call filter rules [9,31] are out of scope.[3] While DonkyLib carefully validates all untrusted input, we consider confused deputy or corruption attacks [12,36,52,59] out of scope. We assume a trusted code base consisting of DonkyLib, all code that is executed before DonkyLib, and the operating system.

We consider side-channel and fault attacks out of scope, and these types of attacks must be addressed by orthogonal mechanisms [8,17,32,38,57,75,92]. However, Donky can, just as process isolation [67], reduce the attack surface of Spectre attacks [40], as we also show in Section 6.1.

Design Overview. While memory protection keys are a powerful building block for in-process isolation, they do not provide proper abstraction for securely shielding software components. In particular, each memory page has exactly one protection key. However, a software component might require multiple protection keys to share memory with other components. To capture this, we use the term "domain" to denote a set of protection keys (and associated memory), their precise usage rights, and their allowed entry points.

By assigning each domain a different set of protection keys, depicted as circles in Figure 1, a variety of trust models can be enforced, as we demonstrate in our use case studies in Section 7. For example, Donky supports sandboxing of untrusted or even malicious code (see domain C in Figure 1). In particular, strong sandboxing of runtime compilers for scripting languages such as JavaScript is in great demand [16,80]. Also, Donky, by design, supports the inverse trust model in which sensitive data is safeguarded in a vault via privilege separation

[3]Note that this assumption has to be made for any shielding system.

Table 1: Donky API handles protection keys and domains (did), and wraps some standard library calls (🔁).

Donky API function		Description
dk_init(), dk_deinit()		(De)Initialize DonkyLib
dk_domain_create(), dk_domain_free(did)		Create/destroy child domain
dk_mmap([did], [key], addr, len, prot ...)	🔁	Allocate memory
dk_mprotect([did], addr, len, prot)	🔁	Protect memory
dk_munmap([did], addr, length)	🔁	Deallocate memory
dk_pkey_alloc(flags, access)	🔁	Allocate protection key
dk_pkey_mprotect([did], addr, len, prot, key)	🔁	Assign memory a prot. key
dk_pkey_free(key)	🔁	Free an unused prot. key
dk_domain_default_key(did)		Get domain's default key
dk_domain_assign_key(did, key, flags, acc)		Assign prot. key to domain
dk_domain_release_child(did)		Untie child dom. from parent
dk_domain_register_dcall([did], callid, entry)		Register an dcall
dk_domain_allow_caller([did], caller_did)		Allow dcalls among domains
dk_pthread_create(thread, attr, entry, arg)	🔁	Create new thread
dk_pthread_exit(retval)	🔁	Exit thread
dk_signal(sig, handler), dk_sigaction(sig, ...)	🔁	Register signal handler

to, e.g., tackle programming errors and their exploitation [66] (see domain B). The versatility of Donky's design supports a variety of intermediary trust models as well, including shared memory (e.g., key K5 is shared between domain B and C) and unprotected legacy code (key K0).

On the hardware side, Donky extends the concept of protection keys with a userspace call-gate mechanism for secure in-userspace domain transitions. This subtle design change solves the non-trivial challenge of combining userspace protection keys with pure userspace key management. Moreover, the hardware call gate intercepts system calls, allowing for efficient in-userspace system call filtering. On the software side, a thin userspace layer called Donky Monitor leverages the hardware call gate for self-protection. Hence, we can safely entrust Donky Monitor with management of domains and protection keys and the interposition of critical system calls. Moreover, Donky Monitor enables fast and secure domain switches via software-defined call gates *without* kernel interaction (cf. the call into the vault in Figure 1).

In Section 5.1, we prototype Donky on RISC-V and implement it on top of the Ariane RISC-V CPU running on an FPGA, and also discuss lightweight adaptations making Intel MPK fully benefit from Donky. In the following, we show how our Donky design meets the goals of secure and efficient in-process isolation and highlight all involved components.

4 Software Design of Donky

In this section, we present the software design of Donky. At its core lies a small handler called Donky Monitor that combines the benefits of a secure hardware call gate with the performance and convenience of pure userspace policy management. Donky Monitor offers a rich software abstraction layer towards application developers via an intuitive Donky API. Also, the monitor safeguards domain transitions via

```
 1 // Allocate domain-private memory
 2 void* pmem = mmap(NULL, 4096, PROT_READ|PROT_WRITE...);
 3 // Allocate (shared) protection key+memory
 4 int key    = pkey_alloc(0, 0);
 5 void* smem = mmap(NULL, 4096, PROT_READ|PROT_WRITE...);
 6 pkey_mprotect(smem, 4096, PROT_READ|PROT_WRITE, key);
 7 // Create child domain & assign shared key
 8 int child = dk_domain_create();
 9 dk_domain_assign_key(child, key, DK_KEY_COPY, 0);
10 // Register a child dcall we can invoke
11 dk_domain_register_dcall(child, 1, child_function);
12 dk_domain_allow_caller(child, current_did);
13 // Decouple child for principle of least priv.
14 dk_domain_release_child(child);
15 // Do dcall
16 child_function(args);
```

Listing 1: The Donky API offers intuitive and secure-by-default management of domains and protection keys.

secure in-userspace software call gates, supports traditional multithreading, and dynamic system call filtering.

Our software design is agnostic to the underlying ISA and works both with our full RISC-V implementation, as well as the x86 emulation mode based on Intel MPK. DonkyLib can sandbox code without recompilation or transformations [15, 86], and be easily integrated into existing projects.

Donky Monitor is our trusted handler in charge of managing in-process access policies in userspace and securing domains from each other. Unlike previous work [15,35,50,99], Donky domains are a pure userspace concept upheld by Donky Monitor without involvement of the kernel.[4]

Donky Monitor is invoked for any operation on domains or protection keys. It also safeguards domain switches via dcalls. To protect itself from tampering, Donky Monitor encapsulates its memory in a separate domain, which has access to all other domains. To achieve security, even in the presence of malicious code, a hardware call-gate mechanism ensures that the monitor can only be entered at its defined entry point. Furthermore, triggering the hardware call gate grants the Donky Monitor permission to update the protection key policy register. Outside the monitor, the register is protected, which obviates the need for binary scanning, CFI, and W$\oplus$X [82].

Software Abstraction Layer. The Donky API is our software abstraction layer, which expands the POSIX interface with Donky API calls. In particular, it allows to manage domains, protection keys and associated memory, and share keys with other domains. The API also manages software call gates to allow for cross-domain calls denoted as dcalls. Table 1 lists our API, of which we discuss the essentials in the following.

Donky API follows a secure-by-default principle, e.g., new domains are isolated by default, and permissions (e.g., to register dcalls to its memory) have to be explicitly granted to other domains. Also, each domain is automatically assigned a unique protection key used to protect its private memory, e.g., stack and `mmap`'ed memory (see Listing 1, line 2). A

[4]Note that Donky reuses Linux MPK support "as is" for allocating and assigning protection keys. The kernel is not aware of domains.

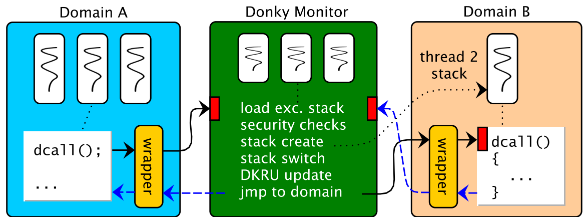

Figure 2: Donky cross-domain dcalls are managed purely in userspace by Donky Monitor, entered via a hardware call gate. Donky Monitor switches domains by switching stacks, updating the policy register (*i.e.*, `DKRU`), and entering the new domain at a software-registered call gate.

protection key is owned by a domain but can be shared with other domains. Starting in the root domain, a program can set up child domains (line 8) with different permissions, also for cross-domain shared memory. A domain can request new protection keys (line 4), tag memory areas with them (line 4), and assign them to other domains for shared memory (line 9). Domain switches require explicit switching permission and well-defined entry points (dcalls) that prevent cross-domain control-flow diversion attacks (lines 11 and 12). Parent domains may drop permissions for child domains (line 14) to reduce attack surface, or to implement a secure software vault (cf. Figure 1). Furthermore, Donky API distinguishes protection key ownership (e.g., for memory mapping) from mere access permission. In line 9, the child domain is only given a copy of the protection key without ownership. E.g., DonkyLib uses this to make its own dynamic string tables read-only visible to others (necessary for the dynamic loader). Finally, DonkyLib ensures that protection keys can only be freed if they are no longer in use, preventing use-after-free [64].

Domain Transitions. Previous work on memory protection keys either requires kernel interaction [15, 99, 99] or Write-XOR-Execute [82] for domain switches. DonkyLib provides fast and secure domain switches without kernel interaction. As shown in Figure 2, dcalls are used to call a function in a different domain and return to the caller again. A dcall invokes the hardware call-gate mechanism to securely trap to Donky Monitor, which handles the domain transition. Automatically generated wrapper code hides interaction with Donky Monitor from the application developer. This is similar to the code generation for SGX's enclave entry points. Moreover, the generated wrapper code has the same type signature as the desired dcall, such that code can transparently invoke dcalls without reordering arguments or return values. DonkyLib also supports nested dcalls, even across an arbitrary number of domains (only constrained by stack size).

DonkyLib registers dcall with unique IDs and their entry addresses to ensure trusted and unforgeable dcalls. At runtime, the monitor is provided with the ID and the information if it is a call or return. It can then decide if the action is allowed and perform the switch to the target domain, which securely switches the protection key policy register and the stack.

As shown in Figure 2, wrappers exist for both the calling and the target domain. They are responsible for interacting with Donky Monitor, saving and restoring non-argument registers before and after a dcall, as well as optionally wiping registers. This ensures integrity and confidentiality of CPU registers across domain transitions. We currently provide macros to auto-generate wrapper code for C functions, and a C++ template class for wrapping C++ member functions in a dcall. The C++ template class furthermore catches uncaught exceptions in the target domain, sanitizes them to avoid information leakage, and re-throws them in the calling domain. Our wrappers support efficient argument passing via CPU registers similar to the system call interface. Large data structures can be passed across domains via shared memory. Tools such as Intel SGX Edger8r [21] could be repurposed for automated copying of such data structures across dcalls.

Multithreading. Donky natively supports POSIX threads. DonkyLib assigns threads to the domain that creates them. Each thread executes in exactly one domain at any point in time. It can switch domains via dcalls. Domains have private user stacks per thread, allocated lazily on first use. For example, in Figure 2, domain A has three threads, of which the second does a dcall. Since domain B was never entered before, Donky Monitor allocates a new stack for this thread.

Each thread gets assigned a separate exception stack, which is protected by Donky Monitor (cf. Figure 2). When invoked, DonkyLib immediately switches to the exception stack in low-level assembler. This ensures that multiple threads can call into DonkyLib. Donky Monitor stores critical thread data in a protected thread-local storage (TLS) area, which we allocate page-aligned in the static TLS and assign it the private protection key of Donky Monitor.

Dynamic System Call Filtering. Controlling system calls is essential for realizing sandboxed environments. Prior work either defines system call protection as an orthogonal problem [35] or demands intrusive changes to the kernel [99].

We filter system calls entirely in userspace using per-domain rules. Compared to kernel filters, our approach offers key advantages: First, we allow fully dynamic filter rules that can be expressed as normal program flow, as opposed to seccomp [47] and eBPF [25]. Appendix A gives an example. Second, we interpose relevant library calls and, thus, can filter at a higher abstraction level.[5] For example, we interpose `pthread_create`, while only blacklisting the underlying `clone` system call. Third, userspace filtering reduces complexity and, thus, also the attack surface of the kernel.

Library interposition is only a convenience, not a security feature. If a malicious domain bypasses it (e.g., by issuing a system call), an exception is raised. We discuss an appropriate hardware and a software mechanism in Section 5.1.

[5]We interpose functions marked with 🔁 in Table 1 via preloading (*i.e.*, `LD_PRELOAD`, `dlsym`) or rewriting symbols with `objcopy`.

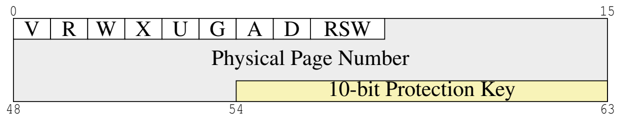

Figure 3: Donky uses reserved top 10 bits of RISC-V page-table entries for protection keys.

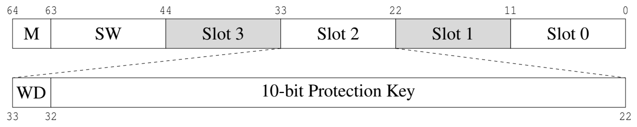

Figure 4: Our RISC-V Donky userspace register (`DKRU`) has four protection key slots with optional write-disable (WD), a monitor bit, and software-defined (SW) space.

Signals. Donky is compatible with POSIX signals. It installs a self-protected signal handler for all signals, and registers its own protected signal stack (e.g., using `sigaction` and `sigaltstack`). Moreover, Donky Monitor interposes signal-related system calls to protect its own handler and to allow domains to register their own signal handlers. Donky Monitor dispatches arriving signals to the domain that registered the corresponding handler, if any, and prepares the protection key policy register and the signal stack accordingly. Normally, Donky Monitor retrieves the stack pointer from the context information given to its signal handler. If interrupted in a domain different from the one registering the handler, Donky Monitor obtains the stack pointer from its internal bookkeeping data. If no stack exists yet, Donky Monitor allocates a new stack, similarly to dcalls (cf. Section 4). Donky Monitor also pushes signal-specific arguments onto the stack, ensuring correct operation of domain signal handlers.

5 Hardware Design of Donky

In this section, we present our hardware implementation of Donky on RISC-V. We design memory protection keys from the ground up on RISC-V and repurpose the RISC-V *N extension* to implement secure call gates in userspace. Furthermore, we describe minimal hardware changes required for Intel MPK to fully support Donky on x86.

5.1 Donky for RISC-V

To evaluate and fully implement Donky on a hardware level, we use the Ariane RISC-V core, a 6-stage, single issue, in-order CPU supporting the RV64IMAC instruction set.

We design memory protection keys for RISC-V, including our protection key policy register and permission checks in the MMU. Furthermore, we augment the Ariane CPU with the *N extension* and repurpose it to support secure hardware call gates in userspace. As of now, *N extension* has only been used for securing embedded systems [65] (cf. Section 2). To our knowledge, we are the first to implement and utilize it for securing a non-embedded system. Our Donky exception mechanism not only guarantees the security of memory protection keys itself. It additionally enables lazy scheduling of protection keys, system call filtering in userspace, as well as virtualization of Donky and the *N extension*.

Memory Protection Keys. Protection keys are configured in the page-table entries (PTE) of a process. RISC-V currently defines two 64-bit virtual memory systems: Sv39 and Sv48, with 39 and 48-bit address spaces, respectively. As shown in Figure 3, both have the upmost 10 bits of a PTE reserved for possible future extensions and to facilitate research experimentation [27]. For Donky, we use these 10 bits for memory protection keys, allowing 1024 different protection keys.

Policy register. Intel MPK keeps the permissions for their 16 protection keys in a single 32-bit register. However, as Donky supports a much higher number of 1024 keys, this is not possible. Instead, we implement key slots, allowing for four simultaneously loaded protection keys in our 64-bit `DKRU` register (cf. Figure 4). Each key slot holds a 10-bit protection key. Only if a protection key is loaded, its associated memory pages can be read or written. Furthermore, each slot has a write-disable bit in the upmost slot bit to enforce read-only memory. While previous architectures [22, 63] also supported large keys, Donky only uses a single register and allows pure userspace management of the `DKRU` register.

We add the `DKRU` register as a user-mode control and status register (CSR). Thus, `DKRU` can be, in principle, configured with standard CSR instructions from all privilege levels. The upmost bit of the `DKRU` register is the so-called monitor bit. If cleared, any access to `DKRU` is disallowed from user mode (see Figure 4). Thus, by clearing this monitor bit, Donky Monitor can prevent unauthorized alteration of the protection key policy. The monitor bit can only be set again by privileged software or by triggering the hardware call gate into Donky Monitor. Finally, `DKRU` offers 19 software-defined bits (SW), which Donky Monitor can freely use to store metadata, such as the domain ID. To support multicore systems, `DKRU` is core-local, as is `PKRU` for x86.

Donky CPU exception. We define a new CPU exception called Donky exception. It is raised whenever Donky detects a security violation while the monitor bit in `DKRU` is cleared. This includes memory access checks as well as illegal access to `DKRU` or CSR's defined by the *N extension*. We extend the memory management unit (MMU) of the Ariane core to verify that for any data access, the protection key in the corresponding PTE matches at least one key loaded in `DKRU`. For store operations, the MMU also checks the corresponding write-disable bits in `DKRU`. For backward compatibility, we exempt protection key zero, which is the default value of PTEs, from the above checks.

Hardware call gate and the N extension. The *N extension* allows the kernel to delegate interrupts and exceptions to a user mode exception handler via the `sedeleg` CSR. This user handler can be specified via `utvec`. A separate `uscratch` register offers scratch space for setting up an exception stack.

We integrate our Donky hardware call gate into the *N extension* as follows: First, the `utvec` and `uscratch` CSRs cannot be accessed if the monitor bit in the `DKRU` register is cleared. Second, for any delegated user exception, the CPU sets the monitor bit, disabling Donky protection. Third, when returning from the user handler with `uret`, the CPU automatically clears the monitor bit, enforcing protection again. This call gate mechanism ensures the security of Donky Monitor. At initialization, Donky Monitor configures `utvec` to point to its entry point and clears the monitor bit. Since Donky Monitor protects its own memory using protection keys, Donky Monitor can only be invoked at this well-defined entry point by triggering, e.g., a Donky exception. Any other attempt to divert code execution into Donky Monitor will keep the monitor bit cleared and, thus, prevent manipulation of `DKRU` and, consequently, Donky Monitor data.

Scheduling of protection keys. If a domain accesses memory for which no protection key is loaded, a Donky exception is triggered that invokes Donky Monitor. Donky Monitor validates whether the access is allowed, and loads the missing protection key into `DKRU`. This happens completely transparent to the domain. To decide which slot to use for the new key, Donky Monitor currently uses a round-robin based technique on key slots 1-3. Slot 0 is always reserved for the domain's default key. Of course, more sophisticated key scheduling methods can be implemented as well. As our scheduling mechanism purely operates on userspace data structures, it does not need expensive kernel invocations to schedule keys and permissions in the PTEs [64].

Syscall filtering in userspace. Donky supports lightweight system call filtering entirely in userspace. On RISC-V, system calls are triggered via the `ecall` instruction, which throws a dedicated exception. We use the same *N extension* delegation mechanism (`sedeleg`) to delegate these system call exceptions directly to Donky Monitor. If the monitor bit is set, however, the system call is forwarded to the kernel. This allows Donky Monitor to do actual system calls.

Note that, while part of our design, our proof-of-concept prototype does not use system call delegation but instead uses a small kernel module to enforce system call interposition. This simplifies the evaluation of our x86 emulation mode.

Virtualization. Donky supports virtualization of the `DKRU` and the *N extension* CSRs. As long as the monitor bit is cleared, all accesses to the corresponding CSRs are blocked. Instead, they raise a Donky exception that traps to Donky Monitor, allowing it to emulate the desired behavior of both, `DKRU` and the *N extension*. This is in line with RISC-V's trap-and-emulate approach to, e.g., implement missing hardware extensions in software. Hence, other schemes can utilize the *N extension* or protection keys for their own purposes without knowledge of Donky, e.g., to achieve CFI [41].

Linux support. The Linux kernel already supports the RISC-V ISA. However, it does not support its *N extension* yet. We extended the Linux kernel 5.1 with our modified *N extension* and have ported the memory protection key feature, which already existed for other architectures. For this, we added all registers necessary for the *N extension*, as well as `DKRU`, to the relevant per-thread kernel structs used during context-switch. The kernel also delegates Donky exceptions to the userspace by configuring `sedeleg`. In total, 700 LoC were changed to support Donky on RISC-V.

Hardware Utilization. The total utilization of our modified Ariane RISC-V CPU on our evaluation board is 69 321 LUTs ($+1.85\,\%$) and 51 395 FFs ($+0.94\,\%$) to the unmodified CPU. The increase is due to the CSRs of the *N extension* as well as our `DKRU` CSR, and the corresponding control logic.

5.2 Extension to Intel MPK

Intel MPK lacks a mechanism for safeguarding its protection key policy register. The `PKRU` register can be changed by anyone via the unprivileged `WRPKRU` instruction. Thus, MPK does not provide the same security as Donky, and schemes using it impose limitations (CFI, W$\oplus$X, and binary scanning).

We propose the following adaptations to make MPK benefit from Donky. Similar to RISC-V, we propose a secure hardware call gate to a trusted handler (Donky Monitor), which safeguards access to `PKRU`. This can be achieved by having one additional Donky Handler Register (`DKHR`), similar to `utvec`, specifying the handler address. Two new instructions allow entering and exiting the handler. The `DENTER` instruction acts similarly to `SYSENTER`. It enables write access to the `PKRU` and jumps to the address in `DKHR`. The register `rcx` will contain the return address (*i.e.*, the address following `DENTER`). Similar to `SYSRET`, `DRET` returns to the previous code (stored in `rcx`, and disables write access to `PKRU`.

We propose using the top-most bit of `DKHR` as the monitor bit to control write access to `PKRU` as well as `DKHR`. It is set and cleared by `DENTER` and `DRET`, respectively. The monitor bit also decides if MPK access violations should be triggered and delegated to `DKHR`. This is required to permit Donky Monitor to access all application memory. `DKHR` exists per core, and the operating system saves and restores it at context switches. New processes automatically have the top-most bit set, so that they can set up `DKHR` themselves. This also provides backward compatibility for programs unaware of `DKHR`.

While x86 does not have a native system call delegation feature like RISC-V, it could be implemented via a hypervisor. However, for better performance, we envision a lightweight hardware extension similar to our RISC-V design: while the monitor bit is set, syscalls should be delegated to the monitor.

More keys. MPK currently only uses 4 PTE bits, supporting 16 protection keys. Since PTE bits 46-51 are reserved for future use, they could be repurposed to support 1024 keys. The same key slotting, as in Figure 4, could be used for `PKRU`.

6 Security and Performance Evaluation

In this section, we evaluate both the security of Donky, as well as its performance using both micro and macro benchmarks.

6.1 Security Evaluation

The security of Donky is built on several layers. First, the security of its building blocks, *i.e.*, memory isolation, call gates, and kernel interaction via system calls and signals. Second, the security of Donky Monitor, its API, and dcalls. And third, the security of a concrete application leveraging Donky. We defer the latter to our case studies in Section 7.

Hardware Call Gates. We prevent code-reuse attacks on Donky Monitor as it can only be legitimately entered via a hardware call gate. Donky exceptions are delivered to this call gate, and the CPU enables the monitor bit inside `DKRU`.

Note that for Donky and Intel MPK, code fetches are not subject to protection key checks, as opposed to read and write data accesses. However, this is not a security issue. If a domain jumps into Donky Monitor code, it cannot manipulate `DKRU`, `utvec`, and `uscratch` since the monitor bit in `DKRU` is still cleared. Moreover, it cannot access Donky Monitor data since it uses a different protection key. Exempting code fetches from protection key checks simplifies code sharing across domains and also allows implementing execute-only memory [97]. As our threat model already considers arbitrary code execution, access to more code does not weaken our security guarantees.

System Calls and Signals. A third building block is to safeguard kernel functionality, *i.e.*, system calls and signals that allow bypassing Donky. Donky interposes system calls by redirecting them to Donky Monitor such that a malicious domain cannot bypass it. For our prototype, we implement a traditional approach, blacklisting dangerous system calls directly in the kernel unless issued by Donky Monitor. For RISC-V, we describe a hardware mechanism to interpose system calls without kernel involvement. Donky Monitor filters system calls based on two criteria. First, it constrains syscalls to uphold domain isolation. Second, an application can install arbitrary domain-specific system call filters, similar to seccomp. Definition of appropriate filter rules is crucial for any domain isolation scheme, yet an orthogonal problem to study (e.g., boomerang attacks [52]). To demonstrate feasibility, our prototype filters memory-related system calls (e.g., mmap, mprotect) to only operate on memory of the current domain.

Our prototype does not yet implement signal handling, as this is merely an engineering effort. Since our use case studies do not strictly demand signals, this has no effect on performance. Nevertheless, we argue why signal handling with Donky can be implemented securely. First, Donky Monitor can protect the *signal origin* by only accepting signals from the kernel, discarding fake ones (*i.e.*, induced by malicious code jumping into the monitor's signal handler). Since Linux drops `PKRU` privileges to protection key zero during signal dispatch, which malicious domains cannot achieve, this boils down to a simple `PKRU` check. Second, *signal delivery* is safeguarded by interposing the registration of signal handlers and loading the correct stack and protection key policy register. Third, interruption of Donky Monitor itself (e.g., via asynchronous signals) is not a security issue when using its own protected signal stack and blocking normal Donky API calls and dcalls for the interrupted thread until signal handling is finished.

Donky Monitor. The above building blocks guarantee the security of Donky Monitor, which is the base for all security services offered by the Donky API. For domains, Donky Monitor stores critical domain metadata in its internal protected data structures, and per-thread information is kept in protected thread-local storage. Donky Monitor carefully validates all untrusted input given to Donky API to avoid confused deputy or corruption attacks [12, 36]. Furthermore, we ensure that stack pointers are within a domain's memory before accessing it inside Donky Monitor.

Donky API. The expressiveness of Donky API allows to represent a variety of protection models, e.g., hierarchical sandboxing, vaults, shared memory, and mutual distrust. To study the concrete security guarantees of a program using Donky is a research field on its own, and a general statement cannot be made. One could, for example, analyze concrete security properties as a sequence of graphs via the take-grant model [49]. Since this is orthogonal to our work, we will focus on the security of our use case scenarios from a programmer's perspective instead, which we defer to Section 7.

We informally describe Donky API rules in terms of the take-grant model. Donky API is designed such that domains can only handle their own resources. These resources include a domain's memory, protection keys, call gates as well as its child domains. A domain can request new resources (*create rule*), constrain their usage (*remove rule*), grant permission to other domains (*grant rule*), but not access foreign resources (limited *take rule*). The grant rule allows domains to open up its call gates to other domains, or share their protection keys. The remove rule fosters the concept of least privilege by dropping ownership of protection keys, reducing their usage rights, or releasing a parent-child relationship. Unless released, a parent domain can always act on behalf of its child domains. The limited take rule only allows elevating privileges on resources for which a domain already has ownership. For example, if a domain *owns* a protection key, it is eligible to reprotect the associated memory, e.g., from read-only to read-write (`mprotect` system call). For granting another domain read-only access to its memory, a domain would create a copy of the associated protection key without ownership.

Secure dcalls. Domain transitions via dcalls demand proper stack management and handling of CPU registers. On the one hand, DonkyLib maintains the call stack abstraction to prevent domains from returning from a dcall that has not been called [12]. We do so by pushing metadata on the caller stack inaccessible to the target domain upon each dcall. Thus, Donky Monitor can verify its validity when the target domain attempts to return. On the other hand, a target domain might violate the calling convention defined by the application binary interface (ABI) and corrupt callee-saved registers. Our call wrapper ensures that these registers are restored. Furthermore, the call wrapper optionally erases non-argument registers upon a dcall to avoid information leakage towards the target domain. Similarly, to prevent information leakage to the calling domain, the target wrapper optionally erases the non-return-argument caller-saved registers before returning.

Spectre attacks. Although Spectre attacks [40] are outside our threat model, Donky can also reduce the attack surface by means of protection keys on Meltdown-resilient systems [13,48]. Kiriansky et al. [39] proposed to use Intel MPK to mitigate Spectre attacks by shielding sensitive data with a separate protection key. We reproduced this result with DonkyLib by constructing a Spectre V1 gadget that leaks a secret but is blocked as soon as protection keys are enforced. Therefore, Donky reduces the attack surface of Spectre attacks significantly, just as process-based isolation (e.g., site isolation [67]) at significantly lower domain switch costs.

6.2 Performance Evaluation

Donky's performance is characterized by the domain switch latency and the execution speed of isolated code and system call interposition. We used microbenchmarks to measure the domain switch latency and macro benchmarks to measure the performance impact of isolated code. The performance of real-world applications is evaluated in Section 7.

Setup. We evaluated the performance on three different machines (1) an Intel Xeon 4208 running at 2.1 GHz and with 16 GB RAM, (2) an Amazon AWS c5.2xlarge instance with an Intel Xeon 8275CL running at 3.6 GHz and 16 GB RAM, and (3) our modified Ariane RISC-V CPU running on Xilinx Kintex-7 FPGA KC705 at 50 MHz. We use the Linux kernel version 5.0.0 for (1), 5.3.0 for (2), and 5.1.0 for (3) in its default configuration. Our microbenchmarks measure the latency in CPU cycles and compare it to the system call latency measured using LMbench [54].

Code size. DonkyLib consists of 2693 lines of C code and 34 lines of generic assembly macros, as measured by sloccount. RISC-V adds 605+272, and our x86 implementation 516+226 lines of C and assembly code, respectively. This includes extensive error checks and debugging code.

Latency. Figure 5 shows Donky latencies relative to a null system call, as this represents the lowest possible time a kernel-based protection mechanism would need to switch domains. We ran each test 1000 times and plotted the mean runtime as well as the standard deviation. Simple Donky API calls to DonkyLib take 160 cycles ($\sigma = 1.4\,\%$) on RISC-V, as opposed to the `getpid` system call taking 724 cycles ($\sigma = 1.9\,\%$), as DonkyLib only needs to prepare its stack and

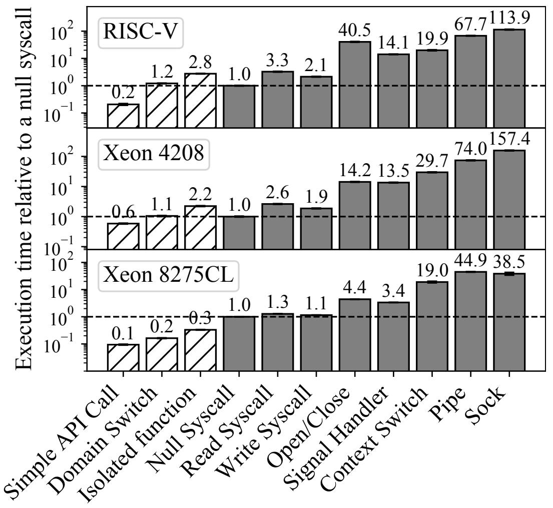

Figure 5: Donky latency for domain switches ▨, compared to system call latency (LMbench) ■.

Table 2: Hardware-based In-process Isolation Systems

Scheme			dcall/syscall (dcall cycles)	CPU	(Linux) kernel
lwC [50]		☢	n.a. (5350*)	Xeon X5650	FreeBSD11
x86-Rings [44]		☢	n.a. (~1400/1200)	i7-4770/AMD1800X	4.13
vmfunc [51]		!	>2x (n.a.)	Xeon 3.4GHz	3.13.7
CHERI [88]	C	☢	n.a. (500)	CHERI 64-bit MIPS	CheriBSD
CODOMs [85]	C		0.1x (30)	gem5-Nehalem	2.6.27
SGX [41]	🛡		71x (7664)	E3-1240v5	3.19
ARMLock [99]	🔑	☢	2.6x (385*)	Raspberry Pi	3.6.11
Shreds [15]	🔑	☢♫	41.7x (n.a.)	Raspberry Pi 2 B	4.1.15
ERIM [82]	🔑	!♫	0.65x (99)	Xeon 6142	4.9.60
Donky	🔑		2.8x (2136)	RISC-V Ariane	5.1.0
Donky	🔑		2.2x (455)	Xeon 4208	5.0.0
Donky	🔑		0.3x (428)	Xeon 8275CL	5.3.0

C Capabilities 🛡 Enclave 🔑 Protection keys * Computed from CPU freq.
☢ Domain switch via kernel ! No full context switch ♫ Instrumentation/CFI

save a few registers. Due to the low latency, performance numbers vary across CPUs and Linux kernel versions. On Xeon 8275CL, simple API calls are even eleven times faster than a system call. To measure a single domain switch, we tested the latency of returning from a dcall to its caller (*i.e.*, the dashed lines in Figure 2). To measure an isolated function call, we tested a full dcall that returns a static value (*i.e.*, the solid and dashed lines in Figure 2). Their runtime is dominated by the domain switches, which include register saving and stack switching, alongside several security checks. Still, dcalls can compete with the fastest possible system calls. On RISC-V, it takes 2.8x the time of a null system call. For our Xeon 4208, it is 2.2x, while on a Xeon 8275CL CPU used in Amazon Web Services, it is even 66.9 % faster than a null system call. When compared to a full process context switch, as reported by LMbench, Donky is even 16–116x faster, making it a viable alternative for process-based isolation mechanisms.

Comparing against related work. Table 2 compares isolated function calls (dcalls) to other in-process schemes, according to their reported numbers. We collect the dcall/syscall

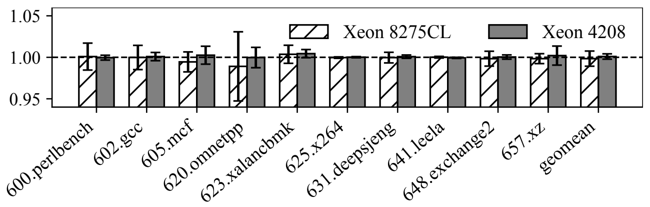

Figure 6: Normalized SPECint 2017 score, isolated with Donky. (Higher is better.)

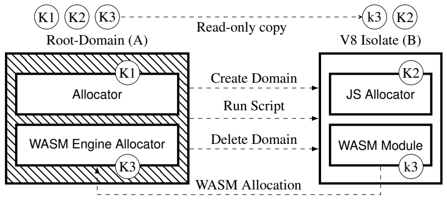

Figure 7: Interactions between root domain and V8 Isolates. Each Isolate and the WASM-Engine share a key. A separate allocator is created in the root domain.

ratio and raw dcall cycles to highlight architectural differences. Donky easily outperforms OS-based schemes [44, 50]. While virtualization seems to achieve good performance [51], the numbers only report overhead for switching translation tables, *i.e.*, extended page tables, but do not prepare stacks or CPU registers necessary for a full dcall. Although the performance of capability-based systems is compelling [85, 88], they require significant changes to both hardware and software. SGX has a different threat model, protecting enclaves from malicious operating systems [41]. Other protection key systems either require significant kernel support for domain switches, instrumentation+CFI+W⊕X, or both [15, 82, 99]. Especially CFI enforcement adds significant runtime overhead [82] not shown here, as opposed to Donky. ARM discontinued protection key support, whose domain switch overhead could compete with Donky [99] at the expense of kernel changes.

Syscalls. To benchmark system call interposition on x86, we run LMbench once with and without our system call blacklisting kernel module. We could not observe measurable overhead even for the fastest Null system call, *i.e.*, the overhead is below the variance. Triggering a blocked system call outside Donky Monitor terminates the application. To evaluate the performance overhead of our proposed RISC-V system call delegation, we benchmark the most restrictive sandboxing filter rule that denies all system calls for the sandboxed domain while allowing them for the root domain. As Donky Monitor can check the domain ID in optimized assembly, the overhead is only 30 cycles (13 instructions), compared to an unfiltered syscall. Thus, on RISC-V, the fastest system call (null system call) is slowed down by only 3.7 %.

Computation. To test the impact of Donky on computation intense workloads without domain switches, we ran the SPEC CPU 2017 intspeed [73] benchmark suite. Since SPEC is long-running, it recommends three runs. To increase significance, we used ten runs. We preloaded DonkyLib with `LD_PRELOAD` and `LD_BIND_NOW`, which initializes itself upon process start and wraps the entire benchmark in a single domain. For comparison, we ran SPEC natively with `LD_BIND_NOW` to avoid bias. As expected, Figure 6 shows that the isolated code runs de-facto at the same speed as native code. The geometric mean runtime overhead for the Xeon 8275CL is -0.16% ($\sigma = 0.91\%$) and 0.10% ($\sigma = 0.32\%$) for the Xeon 4208. Due to its high memory requirements, we could not run SPEC on our RISC-V platform.

Memory overhead. DonkyLib uses metadata for managing domains, which mainly consist of an exception stack for each thread (*i.e.*, 64 KiB), a stack for each actively used thread-domain combination (*i.e.*, with the system's default stack size), and static domain data. This static data includes a list of memory regions along with their permissions and owners and a list of domains with their protection keys and trust-relationships. For 256 domains, each with at most 4096 memory regions, 1024 keys, and 256 threads, this amounts to 2 MiB of static data. Of course, these numbers could be optimized, e.g., by dynamically allocating only as much as is needed.

7 Case Studies

In this section, we evaluate three different real-world use cases. First, we modify the JavaScript engine V8 to provides strong Donky isolation, similar to process isolation (e.g., site isolation). Second, we sandbox the XML-parsing library TinyXML-2 [45], without changing the library. Third, we isolate the cryptographic library Mbed TLS without changing the library.

7.1 Case Study 1: Strong JavaScript Isolation

JavaScript engines have a huge potential for vulnerabilities, such as memory corruption, incorrect compiler optimizations, type confusion, or erroneous code generation [33]. The popular V8 JavaScript engine already uses so-called Isolates for separation, where an Isolate is one instance of a JavaScript runtime environment. While V8 Isolates already encapsulate all the required data, there is no hardware-enforced isolation. Hence, typical exploits escape V8 Isolates by injecting shellcode in their writable code cache [70], and previous work enforced a W⊕X policy [64]. However, advanced sandbox escapes are still possible [33, 68].

In V8, WASM memory is writable and executable by default [79], allowing for the same injection attacks as on the code cache. As a first layer of defense, we use Donky to enforce a W⊕X policy on WASM memory. Furthermore, we add in-process isolation to V8 by encapsulating each Isolate in a separate domain. That is, each Isolate is assigned one domain key. Thus, even if an Isolate gains arbitrary code execution, it is sandboxed in its domain.

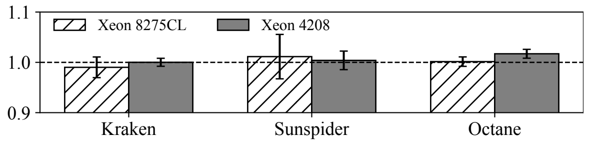

Figure 8: V8 benchmark score with standard deviation running in Donky-protected V8 Isolates, compared to unprotected V8 (dotted line). Higher is better.

We modify V8 (version 8.1.99) to use one allocator per Isolate instead of a global allocator. These per-Isolate allocators leverage DonkyLib to allocate memory with the domain key of the Isolate. The root domain (A) creates Isolates and sets up protection keys and call gates. If a script is executed, the root domain dispatches the script execution to an Isolate, and we switch execution into its domain (B) (see Figure 7). In V8, the WebAssembly (WASM) engine is shared between Isolates. Thus, we create a separate WASM allocator with an additional protection key (K3). Since WASM compilation happens in the root domain, we give the Isolate a read-only copy of its key (k3). Hence, a compromised Isolate cannot use WASM memory to inject custom shellcode. Even if it gains arbitrary code execution, the Isolate cannot access the root domain, since it does not have access to the root key (K1). Only a total of 358 LoC were changed in the V8 engine.

Evaluation. To evaluate sandboxing of V8, we run three JavaScript benchmarks, namely Octane, Kraken, and SunSpider 500 times each. Note that the recommended number of repetitions is 10 for Octane, 100 for SunSpider, and 80 for Kraken [81]. Figure 8 shows the overall scores. In total, there is a performance overhead of 0 to 2 %.

WASM memory corruption is prevented by making its memory writable only by the root domain. To evaluate it, we ported a standard C benchmark program [76] to WASM and measured the overhead between DonkyLib and the original unprotected code. We looped the setup of the WASM program and the calculations 100 times internally to produce WASM memory allocations, with 100 test repetitions, thus giving 10 000 repetitions of the experiment. In total, we observe a runtime overhead of about 2.96 % ($\sigma = 1.02\%$).

To evaluate the security of our Donky V8 sandboxing, we model a strong attacker by providing an arbitrary read and write primitive accessible as global JavaScript functions. We simulate an exploit by performing reads and writes on memory that is not owned by the Isolate's domain. As expected, all memory corruption attempts on memory that is not explicitly assigned to the Isolate domain fail. Since unprotected memory (key zero) might still be vulnerable, one would also protect memory outside V8 from corruption by means of Donky.

7.2 Case Study 2: Third-Party Library

In the second case study, we consider an untrusted third-party library. In the threat model, we assume that the third-party

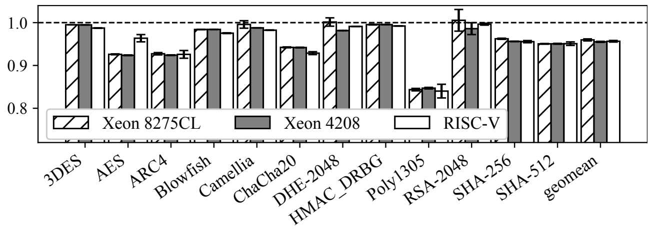

Figure 9: Relative performance of Mbed TLS [6] benchmarks [5], when protected with Donky (higher is better). Similar cryptographic functions are grouped.

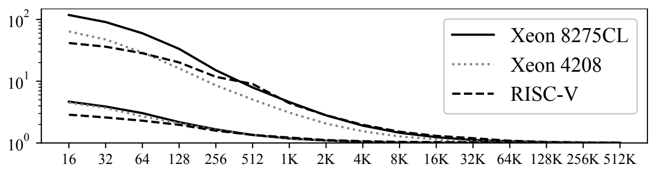

Figure 10: Runtime of different block sizes of Mbed TLS's Poly1305 with process-based isolation (upper three lines) and Donky (lower three lines), normalized using unprotected version.

library contains a vulnerability that can be exploited for arbitrary code execution. As this is often the case for parsing-related activities, we show that Donky can isolate TinyXML-2 [45], an XML-parsing library.

To sandbox the library, we wrap the XMLDocument and XMLElement classes behind Donky dcalls. As these wrappers only call the original methods and handle the domain switch, they can be generated fully automated, similar to SGX Edger8r. Hence, the only difference for an application developer is a different name for the base class. This case study consists of 105 LoC and uses the unmodified TinyXML-2 library. We provide it as part of our open-source code.

Evaluation. To evaluate the security benefits of sandboxing TinyXML-2, we introduce an artificial vulnerability in the library. Donky prevents the library from manipulating any data structures in the host domain, such as the stack. We verified that any such access to host data structures leads to an immediate abortion of the application. Hence, the library cannot mount return-oriented programming attacks on the host, as this can be done from SGX enclaves [71], for example.

7.3 Case Study 3: Library as a Vault

In this case study, we show a different threat model, where DonkyLib protects a library from the rest of the application in a vault. We use Mbed TLS, a cryptographic library, with cryptographic keys as the assets to protect. In the threat model, we assume a vulnerability in the host application, which allows arbitrary memory reads, similar to the Heartbleed bug [93].

We isolate Mbed TLS in its own domain and expose all functions as dcalls. The host application can provide a custom memory allocator to Mbed TLS. By providing the memory management functions from DonkyLib, we ensure that all internal data structures and states of the library are protected with the same domain key. Furthermore, all cryptographic secrets are allocated using DonkyLib to protect them with the same key as the library. Cryptographic secrets are protected from the host application and are only modified through the API, resulting in a strong protection of these assets.

Evaluation. To evaluate the performance impact of the isolation using DonkyLib, we use Mbed TLS's integrated benchmarking suite [5]. We added 95 LoC to the benchmark, which then uses the unmodified Mbed TLS library.

Figure 9 shows the overhead when using the cryptographic functions on a 1 KiB block of input data, which is the default choice. Internally, the benchmark runs for 1000 iterations for each cipher. We ran this experiment 10 times, resulting in a total number of 10 000 repetitions, and plotted their mean values as well as the standard deviations across the 10 runs. As a baseline, we use the performance of the unprotected Mbed TLS library. We group similar cryptographic functions (e.g., same algorithm but different key size) by summing up their respective runtimes. With a throughput of 96 % (geomean) compared to the unprotected version, the performance impact of Donky is minimal. Even the fastest operation (Poly1305), *i.e.*, the function requiring the most domain switches, has only a small throughput reduction of 15 %.

To account for different block sizes, we compared Donky with process-based isolation by isolating Poly1305 using both techniques. We chose Poly1305 as it does most domain switches. Other algorithms would show significantly less overhead. For process isolation, we used a semaphore and shared memory for synchronization and pinned both processes to the same CPU core. As shown in Figure 10, at a block size of 16 Bytes, process-based isolation runs 42–118x slower, while Donky is only 2.9–4.7x slower.

8 Discussion

In this section, we discuss limitations as well as future work and elaborate on related work.

8.1 Limitations and Future Work

Static Limits. Our prototype uses statically allocated arrays to store its metadata, which poses an upper limit on the number of domains, memory regions, and keys. To overcome these limits, one could dynamically allocate Donky Monitor's memory. Moreover, Donky is limited to 16 protection keys for x86 and 1024 for RISC-V. If an application needs more keys, one could schedule protection keys, as done by [64]. Alternatively, one could resort to weaker probabilistic protection by reusing protection keys. We prototyped a virtualization scheme that hands out protection keys marked for virtualization multiple times. One could also increase the number of keys supported by the hardware, as mentioned in Section 5.

Availability. DonkyLib is designed for security and, in-line with related shielding technologies, e.g., Intel SGX, denial-of-service attacks are possible. One could retrofit DonkyLib with safety guarantees, e.g., by limiting the number of protection keys a domain can allocate, or rate-limiting the API calls.

Thread-Local Storage. Previous work largely ignores the security of the TLS across domain switches. While Intel SGX is a notable exception, we believe more research is needed. SGX switches the TLS at enclave entry and exit, and Donky could similarly swap the TLS pointer for dcalls.[6] However, SGX enclaves are built as standalone libraries without external dependencies, and code is never shared across domains. It is unclear whether and how secure code reuse across domains is possible, should this code make use of TLS.

8.2 Related Work

Software-based Approaches. Software Fault Isolation (SFI) schemes [24, 26, 53, 72, 86, 95, 98] use CFI and binary rewriting to confine sandboxes to a restricted memory area. In comparison to SFI, our context-switching overheads are higher, but the overhead within a domain is lower. Furthermore, Donky's threat model is stronger. We can isolate unmodified code without enforcing the control-flow integrity of isolated code. Because CFI usually requires W$\oplus$X, it cannot easily support self-modifying code. This is a clear advantage for Donky. Also, some CFI schemes only offer probabilistic protection [42].

NaClJIT [3] adds SFI to a JIT compiler with a runtime overhead of 50 to 60 % for V8. Other works [7, 10, 35, 50, 74] rely on substantial kernel modifications to provide isolation between domains, such as, e.g., separate address spaces for threads [35, 87].

NaCl [95] and Dune [7] can provide similar software-based system call filtering as Donky. However, in contrast to NaCl, Donky provides a mechanism to enforce these filters even when the application manages to break out of its SFI/CFI sandbox. Compared to Dune, Donky addresses multiple in-process compartments not only on a thread boundary. Also, Donky's syscalls are significantly faster than Dune's.

Hardware Protection Key Approaches. ERIM [82] uses MPK for in-process isolation. Unlike Donky, they demand binary scanning and rewriting, alongside W$\oplus$X. While they defer setting up private stacks to the developer, DonkyLib provides them by default. ERIM's binary rewriting could be integrated into a JIT compiler. However, it may lead to crashes if the compiler accidentally emits unsafe `WRPKRU` instructions. Also, the performance and implementation costs to adapt JIT compilers accordingly is unclear. However, NaClJIT [3] could serve as a starting point for further research. Koning et al. [41]

[6] E.g., Donky Monitor could update the RISC-V `tp` register, which is otherwise protected by the monitor bit in `DKRU`.

survey different hardware isolation mechanisms such as Intel MPK and isolate safe regions (e.g., shadow stacks) atop of them. `libmpk` [64] schedules protection keys for Intel MPK via expensive PTE updates if more than 16 keys are used.

ARMLock [99] implements an in-process isolation framework using ARM's Memory Domains [4]. Binary scanning is not required on ARM, as their protection key policy register cannot be written in userspace. ARMLock implements domains in the kernel, which increases the attack surfaces and likely impedes wide adoption. Also, ARM removed Memory Domains on 64-bit architectures. In contrast, Donky manages domain metadata and domain transitions entirely in userspace, which allows for faster inter-domain calls.

Shreds [15] uses ARM's Memory Domains to isolate so-called shreds from the rest of an application. They do not support the sandboxing scenario, demand recompilation of in-shred code, and a coarse-grained CFI policy. Different shreds cannot easily share data. Protection keys are lazily switched during context switches using an expensive page-table walk.

Apart from [41, 64, 82], others did not open-source their code, hindering further research. We open-source both DonkyLib and our RISC-V hardware.

Trusted Execution Environments. Intel SGX [20], ARM TrustZone [60], Sancus [62], and proposed RISC-V extensions [23, 43] protect against a malicious operating system. However, they require extensive hardware modifications, and communication between domains is typically slow.

Intel SGX [20] runs code in so-called enclaves, which only allow an asymmetric trust model [90], in which an enclave has access to the entire process. Furthermore, they have a higher performance overhead [91]. Recent work used MPK to also protect the host application from the enclave [90] or to provide additional privilege separation within an enclave [55].

Compartmentalization. Decomposing software to run in isolated compartments is an orthogonal problem. Previous work aids in finding suitable isolation boundaries, but splitting up existing software is still a hard problem [11, 34, 51, 83, 84]. Choosing an isolation boundary is always a trade-off between fine isolation granularity and minimizing switching overhead and, hence, it often cannot be fully automated. RLBox [59] identifies such compartmentalization boundaries in Firefox and designs secure interfaces. Furthermore, they automatically sanitize pointers across compartments to prevent confused deputy attacks. In contrast, Donky provides a strong, generic isolation framework RLBox could use to enforce their compartmentalization.

9 Conclusion

In this paper, we proposed Donky, a hardware-software co-design solution for secure and efficient in-process isolation. It provides strong isolation guarantees with a negligible performance impact. It is fully backward compatible with existing software libraries and dynamically generated code (e.g., JIT). Donky relies on a small hardware extension of memory protection keys to back the security guarantees of our software framework called DonkyLib. We presented a fully working implementation on a RISC-V processor and showed that Donky can be implemented on top of commodity x86 processors with a minimal hardware extension. Our trusted monitor runs entirely in userspace, thus minimizing switching overhead as well as kernel complexity. DonkyLib works on both x86 and RISC-V CPUs and provides pure userspace domains atop protection keys through an intuitive API.

Donky combines the high performance of MPK with the security of kernel-based schemes. Donky cross-domain switches are 16–116x faster than process context switches and have only 4 % overhead compared to fully unprotected mbedTLS cryptographic operations. We support self-modifying code, just-in-time compilation, and in-process third-party binary sandboxing without scanning or rewriting instructions. This addresses recent challenges in JavaScript sandboxing, ranging from browsers and desktop applications to the cloud.

Acknowledgments

We thank the anonymous reviewers, the artifact evaluators, and especially our shepherd, Nathan Dautenhahn, for their valuable suggestions and comments, which helped in improving the paper. This work has been supported by the Austrian Research Promotion Agency (FFG) via the competence center Know-Center (grant number 844595), which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, and Styria, and via the project ESPRESSO, which is funded by the province of Styria and the Business Promotion Agencies of Styria and Carinthia. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). Additional funding was provided by generous gifts from Intel and from Cloudflare. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Ariane RISC-V CPU. https://github.com/pulp-platform/ariane, 2019.

[2] A Collection of Chrome Sandbox Escape POCs/Exploits for learning. https://github.com/allpaca/chrome-sbx-db, 2019.

[3] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff Biffle, and Bennet Yee. Language-independent sandbox-

ing of just-in-time compilation and self-modifying code. In *PLDI*, pages 355–366, 2011.

[4] ARM. ARM Developer Suite Developer Guide. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0056d/BABBJAED.html, 2001.

[5] ARM. Mbed TLS Benchmark. https://github.com/ARMmbed/mbedtls/blob/master/programs/test/benchmark.c, 2019.

[6] ARM. SSL Library Mbed TLS / PolarSSL. https://tls.mbed.org/, 2019.

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

[8] Daniel J. Bernstein. Cache-Timing Attacks on AES, 2004.

[9] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX. In *USENIX Security Symposium*, 2018.

[10] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *NSDI*, 2008.

[11] David Brumley and Dawn Xiaodong Song. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *USENIX Security Symposium*, 2004.

[12] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In *CCS*, 2019.

[13] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium*, 2019.

[14] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *SOSP*, 2009.

[15] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-Grained Execution Units with Private Memory. In *S&P*, 2016.

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

[17] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *S&P*, 2009.

[18] Jonathan Corbet. Deferring seccomp decisions to user space. https://lwn.net/Articles/756233/, 2018.

[19] Intel Corporation. Intel 64 and IA-32 Architectures Software Developers Manual, October 2019.

[20] Intel Corporation. Intel Software Guard Extensions (Intel SGX). https://software.intel.com/en-us/sgx.

[21] Intel Corporation. Intel Software Guard Extensions (Intel SGX) SDK. https://software.intel.com/sgx-sdk.

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

[23] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security Symposium*, 2016.

[24] Liang Deng, Qingkai Zeng, and Yao Liu. ISboxing: An Instruction Substitution Based Data Sandboxing for x86 Untrusted Libraries. In *SEC*, volume 455 of *IFIP Advances in Information and Communication Technology*, 2015.

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

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

[27] RISC-V Foundation. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, version 1.10. https://content.riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf, 2017.

[28] RISC-V Foundation. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, document version 20191213. https://riscv.org/specifications/, 2019.

[29] RISC-V Foundation. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, document version 1.12-draft. https://github.com/riscv/riscv-isa-manual/releases/download/draft-20200212-c3d1f07/riscv-privileged.pdf, 2020.

[30] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. IMIX: In-Process Memory Isolation EXtension. In *USENIX Security Symposium*, 2018.

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

[32] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptographic Engineering*, 8, 2018.

[33] Github: Tunz. Case Study of JavaScript Engine Vulnerabilities. https://github.com/tunz/js-vuln-db.

[34] Khilan Gudka, Robert N. M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. Clean Application Compartmentalization with SOAAP. In *CCS*, 2015.

[35] Terry Ching-Hsiang Hsu, Kevin J. Hoffman, Patrick Eugster, and Mathias Payer. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *CCS*, 2016.

[36] Hong Hu, Zheng Leong Chua, Zhenkai Liang, and Prateek Saxena. Identifying Arbitrary Memory Access Vulnerabilities in Privilege-Separated Software. In *ESORICS*, volume 9327 of *LNCS*, pages 312–331, 2015.

[37] IBM Corporation. Power ISA version 3.0b. 2017.

[38] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA*, 2014.

[39] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. *ePrint 2018/418*, May 2018.

[40] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, 2019.

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

[42] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer Integrity. In *OSDI*, 2014.

[43] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Dawn Song, and Krste Asanovic. Keystone: A Framework for Architecting TEEs. *CoRR*, abs/1907.10119, 2019.

[44] Hojoon Lee, Chihyun Song, and Brent ByungHoon Kang. Lord of the x86 Rings: A Portable User Mode Privilege Separation Architecture on x86. In *CCS*, 2018.

[45] Lee Thomason. TinyXML-2. https://github.com/leethomason/tinyxml2, 2019.

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

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

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

[49] Richard J. Lipton and Lawrence Snyder. A Linear Time Algorithm for Deciding Subject Security. *J. ACM*, 24, 1977.

[50] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *OSDI*, 2016.

[51] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *CCS*, 2015.

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

[53] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC Architecture. In *USENIX Security Symposium*, 2006.

[54] Larry W. McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *USENIX ATC*, 1996.

[55] Marcela S. Melara, Michael J. Freedman, and Mic Bowman. EnclaveDom: Privilege Separation for Large-TCB Applications in Trusted Execution Environments. *CoRR*, abs/1907.13245, 2019.

[56] Lucian Mogosanu, Ashay Rane, and Nathan Dautenhahn. MicroStache: A Lightweight Execution Context for In-Process Safe Region Isolation. In *RAID*, volume 11050 of *LNCS*, 2018.

[57] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *Security and Privacy (S&P)*, 2020.

[58] Myoung Jin Nam, Periklis Akritidis, and David J. Greaves. FRAMER: a tagged-pointer capability system with memory safety applications. In *ACSAC*, 2019.

[59] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting Fine Grain Isolation in the Firefox Renderer (Extended Version). *CoRR*, abs/2003.00572, 2020.

[60] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. TrustZone Explained: Architectural Features and Use Cases. In *CIC*, 2016.

[61] Node.js. https://nodejs.org/en/docs/es6/, 2019.

[62] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix C. Freiling. Sancus 2.0: A Low-Cost Security Architecture for IoT Devices. *ACM Trans. Priv. Secur.*, 20, 2017.

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

[64] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *USENIX ATC*, 2019.

[65] Sandro Pinto and Cesare Garlati. User mode interrupts: A must for securing embedded systems. In *Embedded World Conference*, 2019.

[66] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing Privilege Escalation. In *USENIX Security Symposium*, 2003.

[67] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site Isolation: Process Separation for Web Sites within the Browser. In *USENIX Security Symposium*, 2019.

[68] Google Security Research. Google Chrome 72.0.3626.121 / 74.0.3725.0 - 'NewFixedDoubleArray' Integer Overflow. https://github.com/riscv/riscv-isa-manual/releases/download/draft-20200212-c3d1f07/riscv-privileged.pdf, 2020.

[69] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63, 1975.

[70] Samuel Gross. Exploiting Logic Bugs in JavaScript JIT Engines. http://www.phrack.org/papers/jit_exploitation.html.

[71] Michael Schwarz, Samuel Weiser, and Daniel Gruss. Practical Enclave Malware with Intel SGX. In *DIMVA*, volume 11543 of *LNCS*, 2019.

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

[73] Standard Performance Evaluation Corporation. SPEC CPU 2017. https://www.spec.org/cpu2017.

[74] Raoul Strackx, Pieter Agten, Niels Avonds, and Frank Piessens. Salus: Kernel Support for Secure Process Compartments. *ICST Trans. Security Safety*, 2, 2015.

[75] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *USENIX Security Symposium*, 2017.

[76] The Computer Language Benchmarks Game Team. Nbody C Benchmark. https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/nbody.html#nbody.

[77] Peter Teoh. How can eBPF be compromised by vulnerabilities? https://tthtlc.wordpress.com/2019/01/01/how-can-ebpf-be-compromised-by-vulnerabilities/, 2019.

[78] The New York Times. The Loophole That Turns Your Apps Into Spies. https://www.nytimes.com/2019/09/24/opinion/facebook-google-apps-data.html, 2019.

[79] V8. The official mirror of the V8 Git repository. https://github.com/v8/v8/blob/3fbeb93760bcf663dcf84b57597f49d7d3b29c02/src/flags/flag-definitions.h#L665, 2020.

[80] v8 - Untrusted code mitigations. https://v8.dev/docs/untrusted-code-mitigations, 2019.

[81] v8 developer blog. https://v8.dev/docs, 2019.

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

[83] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. Towards Fine-grained, Automated Application Compartmentalization. In *PLOS*, 2017.

[84] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. BreakApp: Automated, Flexible Application Compartmentalization. In *NDSS*, 2018.

[85] Lluís Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. CODOMs: Protecting software with Code-centric memory Domains. In *ISCA*, 2014.

[86] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *SOSP*, 1993.

[87] Jun Wang, Xi Xiong, and Peng Liu. Between Mutual Trust and Mutual Distrust: Practical Fine-grained Privilege Separation in Multithreaded Applications. In *USENIX ATC*, 2015.

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

[89] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert M. Norton, Michael Roe, Stacey D. Son, and Munraj Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *S&P*, 2015.

[90] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. SGXJail: Defeating Enclave Malware via Confinement. In *RAID*, 2019.

[91] Ofir Weisse, Valeria Bertacco, and Todd M. Austin. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. In *ISCA*, 2017.

[92] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *USENIX Security Symposium*, 2019.

[93] David A. Wheeler. Preventing Heartbleed. *IEEE Computer*, 47, 2014.

[94] Emmett Witchel, Josh Cates, and Krste Asanovic. Mondrian memory protection. In *ASPLOS*, 2002.

[95] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *S&P*, 2009.

[96] Florian Zaruba and Luca Benini. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Trans. VLSI Syst.*, 27, 2019.

[97] Mingwei Zhang, Ravi Sahita, and Daiping Liu. executable-only-memory-switch (xom-switch): Hiding your code from advanced code reuse attacks in one shot. *Black Hat Asia*, 2018.

[98] Lu Zhao, Guodong Li, Bjorn De Sutter, and John Regehr. ARMor: fully verified software fault isolation. In *EMSOFT*, 2011.

[99] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. ARMlock: Hardware-based Fault Isolation for ARM. In *CCS*, 2014.

A System Call Filter Example

```
1  int interpose_socket(int dom, int type, int prot) {
2    if (CURRENT_DOMAIN != 0) {
3      errno = EACCES;
4      return -1;
5    }
6    return socket(dom, type, prot);
7  }
8  int interpose_open(const char *path, int flags) {
9    if (!login || strchr(path, '/')) {
10     errno = EACCES;
11     return -1;
12   }
13   return open(path, flags);
14 }
```

Listing 2: DonkyLib user mode filters benefit from the full application context.

Listing 2 shows how an application using Donky can constrain socket creation to the root domain (did=0) only (line 2). Furthermore, opening of files is bound to some login procedure via a global variable `login` and limited to the current directory (line 9).

Recent additions to the Linux kernel similarly allow such filters in userspace [18]. However, unlike Donky, it requires kernel interaction and a separate thread or process.