ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK)

Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg, ***Max Planck Institute for Software Systems, Saarland Informatics Campus***

https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner

This paper is included in the Proceedings of the 28th USENIX Security Symposium.

August 14–16, 2019 • Santa Clara, CA, USA

978-1-939133-06-9

Open access to the Proceedings of the 28th USENIX Security Symposium is sponsored by USENIX.

ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK)

Anjo Vahldiek-Oberwagner Eslam Elnikety Nuno O. Duarte
Michael Sammler Peter Druschel Deepak Garg
Max Planck Institute for Software Systems (MPI-SWS), Saarland Informatics Campus

Abstract

Isolating sensitive state and data can increase the security and robustness of many applications. Examples include protecting cryptographic keys against exploits like OpenSSL's Heartbleed bug or protecting a language runtime from native libraries written in unsafe languages. When runtime references across isolation boundaries occur relatively infrequently, then conventional page-based hardware isolation can be used, because the cost of kernel- or hypervisor-mediated domain switching is tolerable. However, some applications, such as the isolation of cryptographic session keys in network-facing services, require very frequent domain switching. In such applications, the overhead of kernel- or hypervisor-mediated domain switching is prohibitive.

In this paper, we present ERIM, a novel technique that provides hardware-enforced isolation with low overhead on x86 CPUs, even at high switching rates (ERIM's measured overhead is less than 1% for 100,000 switches per second). The key idea is to combine protection keys (MPKs), a feature recently added to x86 that allows protection domain switches in userspace, with binary inspection to prevent circumvention. We show that ERIM can be applied with little effort to new and existing applications, doesn't require compiler changes, can run on a stock Linux kernel, and has low runtime overhead even at high domain switching rates.

1 Introduction

It is good software security practice to partition sensitive data and code into isolated components, thereby limiting the effects of bugs and vulnerabilities in a component to the confidentiality and integrity of that component's data. For instance, isolating cryptographic keys in a network-facing service can thwart vulnerabilities like the OpenSSL Heartbleed bug [37]; isolating a managed language's runtime can protect its security invariants from bugs and vulnerabilities in co-linked native libraries; and, isolating jump tables can prevent attacks on an application's control flow.

Isolation prevents an untrusted component from directly accessing the private memory of other components. Broadly speaking, isolation can be enforced using one of two approaches. First, in software fault isolation (SFI) [47], one instruments the code of untrusted components with bounds checks on indirect memory accesses, to prevent access to other components' memory. The bounds checks can be added by the compiler or through binary rewriting. Bounds checks impose overhead on the execution of all untrusted components; additional overhead is required to prevent control-flow hijacks [30], which could circumvent the bounds checks. On x86-64, pointer masking-based SFI techniques like Native Client [42] incur overheads of up to 42% on the execution of untrusted code [30]. Even with hardware-supported bounds checks, like those supported by the Intel MPX ISA extension [26], the overhead is up to 30%, as shown in by Koning *et al.* [30] and later in Section 6.5.

Another approach is to use hardware page protection for memory isolation [9, 10, 13, 32, 33, 34]. Here, access checks are performed in hardware as part of the address translation with no additional overhead on execution *within* a component. However, transferring control *between* components requires a switch to kernel or hypervisor mode in order to change the (extended) page table base. Recent work such as Wedge, Shreds, SeCage, SMVs, and light-weight contexts (lwCs) [10, 13, 24, 33, 34] have reduced the overhead of such switching, but the cost is still substantial. For instance, Litton *et al.* [33] report a switching cost of about 1us per switch for lwCs, which use kernel-managed page tables for in-process isolation. This amounts to an overhead of nearly 10% for an application that switches 100,000 times a second and, in our experiments, an overhead of up to 65% on the throughput of the web server NGINX when lwCs are used to isolate session keys (Section 6.5). Techniques based on Intel VT-x extended page tables with VMFUNC [34] have less overhead, but the overhead is still high—up to 14.4% on NGINX's throughput in our experiments (Section 6.5).

In this paper, we present ERIM, the first isolation technique for x86 that combines near-zero overhead on in-component execution with very low cost switching among components. ERIM relies on a recent x86 ISA extension called protection keys (MPK) [28]. With MPK, each virtual page can be tagged with a 4-bit domain id, thus partitioning a process's address space into up to 16 disjoint domains. A special register, PKRU, that is local to each logical core determines which domains the core can read or write. Switching domain permissions requires writing the PKRU register in userspace, which takes only 11–260 cycles on current Intel CPUs, corresponding to an overhead of 0.07% to 1.0% per 100,000 switches/s on a 2.6 GHz CPU. This amounts to an overhead of at most 4.8% on the throughput of NGINX when isolating all session keys, which is up to 6.3x, 13.5x and 3x lower than the overhead of similar protection using SFI (with Intel MPX), lwCs and Intel VT-x, respectively.

However, MPK by itself does not provide strong security because a compromised or malicious component can sim-

ply write to the PKRU register and grant itself permission to access any component. ERIM relies on *binary inspection* to ensure that all occurrences of instructions that update the PKRU in the binary are *safe*, i.e., they cannot be exploited to gain unauthorized access. With this, ERIM provides isolation without requiring control-flow integrity in untrusted code, and therefore avoids the runtime overhead of ensuring control-flow integrity in unsafe languages.

While ERIM's binary inspection enforces the safety of its MPK-based isolation, it creates a potential usability issue: What to do if a binary has *unintentional* occurrences of PKRU-updating instructions? Since x86 does not require instruction alignment, such occurrences could arise within a longer instruction, or spanning the bytes of two or more adjacent instructions. Any such sequence could be exploited by a control-flow hijack attack and must be rejected by the binary inspection mechanism. To handle such cases, we describe a novel procedure to *rewrite* any instruction sequence containing an unaligned PKRU-updating instruction to a functionally equivalent sequence without the instruction. This rewriting procedure can be integrated with a compiler or our binary inspection.

ERIM is the first technique that enables efficient isolation in applications that require very high domain switching rates (~10^5/s or more) and also spend significant time executing inside untrusted components. We evaluate our ERIM prototype on three such applications: 1) Isolating the frequently accessed session keys in a web server (NGINX), 2) isolating a managed language runtime from native libraries written in unsafe languages, and 3) efficiently isolating the safe region in code-pointer integrity [31]. In all cases, we observe switching rates of order 10^5 or more per second per core. ERIM provides strong, hardware-based isolation in all these cases, with overheads that are considerably lower than those of existing techniques. Moreover, ERIM does not require compiler support and can run on stock Linux.

In summary, this paper makes the following contributions. 1) We present ERIM, an efficient memory isolation technique that relies on a combination of Intel's MPK ISA extension and binary inspection, but does not require or assume control-flow integrity. 2) We describe a complete rewriting procedure to ensure binaries cannot be exploited to circumvent ERIM. 3) We show that ERIM can protect applications with high inter-component switching rates with low overhead, unlike techniques based on hardware (extended) page tables and SFI (even with hardware support).

2 Background and related work

In this section, we survey background and related work. Enforcing relevant security or correctness invariants while trusting only a small portion of an application's code generally requires *data encapsulation*. Encapsulation itself requires *isolating* sensitive data so it cannot be accessed by untrusted code, and facilitating *switches* to trusted code that has access to the isolated state. We survey techniques for isolation and switching provided by operating systems, hypervisors, compilers, language runtimes, and binary rewriting, as well as other work that uses MPK for memory isolation.

OS-based techniques Isolation can be easily achieved by placing application components in separate OS processes. However, this method has high overhead even with a moderate rate of cross-component invocation. Novel kernel abstractions like light-weight contexts (lwCs) [33], secure memory views (SMVs) [24] and nested kernels [14], combined with additional compiler support as in Shreds [13] or runtime analysis tools as in Wedge [10], have reduced the cost of such data encapsulation to the point where isolating *long-term* signing keys in a web server is feasible with little overhead [33]. Settings that require more frequent switches like isolating *session keys* or the safe region in CPI [31], however, remain beyond the reach of OS-based techniques.

Mimosa [20] relies on the Intel TSX hardware transactional memory support to protect private cryptographic keys from software vulnerabilities and cold-boot attacks. Mimosa restricts cleartext keys to exist only within uncommitted transactions, and TSX ensures that an uncommitted transaction's data is never written to the DRAM or other cores. Unlike ERIM, which is a general-purpose isolation technique, Mimosa specifically targets cryptographic keys, and is constrained by hardware capacity limits of TSX.

Virtualization-based techniques In-process data encapsulation can be provided by a hypervisor. Dune [9] enables user-level processes to implement isolated compartments by leveraging the Intel VT-x x86 virtualization ISA extensions [28]. Koning et al. [30] sketch how to use the VT-x VMFUNC instruction to switch extended page tables in order to achieve in-process data isolation. SeCage [34] similarly relies on VMFUNC to switch between isolated compartments. SeCage also provides static and dynamic program analysis based techniques to automatically partition monolithic software into compartments, which is orthogonal to our work. TrustVisor [36] uses a thin hypervisor and nested page tables to support isolation and additionally supports code attestation. SIM [44] relies on VT-x to isolate a security monitor within an untrusted guest VM, where it can access guest memory with native speed. In addition to the overhead of the VMFUNC calls during switching, these techniques incur overheads on TLB misses and syscalls due to the use of extended page tables and hypercalls, respectively. Overall, the overheads of virtualization-based encapsulation are much higher than those of ERIM.

Nexen [45] decomposes the Xen hypervisor into isolated components and a security monitor, using page-based protection within the hypervisor's privilege ring 0. Control of the MMU is restricted to the monitor; compartments are de-privileged by scanning and removing exploitable MMU-modifying instructions. The goal of Nexen is quite different

from ERIM's: Nexen aims to isolate co-hosted VMs and the hypervisor's components from each other, while ERIM isolates components of a user process. Like ERIM Nexen scans for and removes exploitable instructions.

Language and runtime techniques Memory isolation can be provided as part of a memory-safe programming language. This encapsulation is efficient if most of the checks can be done statically. However, such isolation is language-specific, relies on the compiler and runtime, and can be undermined by co-linked libraries written in unsafe languages.

Software fault isolation (SFI) [47] provides memory isolation in unsafe languages using runtime memory access checks inserted by the compiler or by rewriting binaries. SFI imposes a continuous overhead on the execution of untrusted code. Additionally, SFI by itself does not protect against attacks that hijack control flow (to possibly bypass the memory access checks). To get strong security, SFI must be coupled with an additional technique for control-flow integrity (CFI) [6]. However, existing CFI solutions have nontrivial overhead. For example, code-pointer integrity (CPI), one of the cheapest reasonably strong CFI defenses, has a runtime overhead of at least 15% on the throughput of a moderately performant web server (Apache) [31, Section 5.3]. In contrast, ERIM does not rely on CFI for data encapsulation and has much lower overhead. Concretely, we show in Section 6 that ERIM's overhead on the throughput of a much more performant web server (NGINX) is no more than 5%.

The Intel MPX ISA extension [28] provides architectural support for bounds checking needed by SFI. A compiler can use up to four bounds registers, and each register can store a pair of 64-bit starting and ending addresses. Specialized instructions check a given address and raise an exception if the bounds are violated. However, even with MPX support, the overhead of bounds checks is of the order of tens of percent points in many applications (Section 6.5 and [12, 30, 40]).

Hardware-based trusted execution environments Intel's SGX [27] and ARM's TrustZone [8] ISA extensions allow (components of) applications to execute with hardware-enforced isolation. JITGuard [17], for instance, uses SGX to protect the internal data structures of a just-in-time compiler from untrusted code, thus preventing code-injection attacks. While SGX and TrustZone can isolate data even from the operating system, switching overheads are similar to other hardware-based isolation mechanisms [30].

IMIX [18] and MicroStach [38] propose minimal extensions to the x86 ISA, adding load and store instructions to access secrets in a safe region. The extended ISA can provide data encapsulation. Both systems provide compilers that automatically partition secrets. However, for data encapsulation in the face of control-flow hijack attacks, both systems require CFI. As mentioned, CFI techniques have nontrivial overhead. ERIM, on the other hand, provides strong isolation without relying on CFI and has lower overhead.

ASLR Address space layout randomization (ASLR) is widely used to mitigate code-reuse exploits such as those based on buffer overflow attacks [43, 23]. ASLR has also been used for data encapsulation by randomizing data layout. For example, as one of the isolation techniques used in CPI [31, 46], a region of sensitive data is allocated at a random address within the 48-bit x86-64 address space and its base address is stored in a segment descriptor. All pointers stored in memory are offsets into the region and do not reveal its actual address. However, all forms of ASLR are vulnerable to attacks like thread spraying [43, 25, 16, 19, 39]. Consequently, ASLR is not viable for strong memory isolation, despite proposals such as [35] to harden it.

ARM memory domains ARM memory domains [7] are similar to Intel MPK, the x86 feature that ERIM relies on. However, unlike in MPK, changing domains is a kernel operation in ARM. Therefore, unlike MPK, ARM's memory domains do not support low-cost user-mode switching.

MPK-based techniques Koning et al. [30] present MemSentry, a general framework for data encapsulation, implemented as a pass in the LLVM compiler toolchain. They instantiate the framework with several different memory isolation techniques, including many described above and one based on MPK domains. However, MemSentry's MPK instance is secure only with a separate defense against control-flow hijack/code-reuse attacks to prevent adversarial misuse of PKRU-updating instructions in the binary. Such defenses have significant overhead of their own. As a result, the overall overhead of MemSentry's MPK instance is significantly higher than that of ERIM, which does not rely on a defense against control-flow hijacks.

In concurrent work [22], Hedayati *et al.* describe how to isolate userspace libraries using VMFUNC or Intel MPK. The MPK-based method is similar to ERIM, but does not address the challenge of ensuring that there are no exploitable occurrences of PKRU-modifying instructions. Rewriting binaries in this manner is a key contribution of our work (Section 4). Finally, Hedayati *et al.* rely on kernel changes while ERIM can run safely on a stock Linux kernel.

libmpk [41] virtualizes MPK memory domains beyond the 16 supported in hardware. It also addresses potential security issues in the API of Linux's MPK support. libmpk addresses concerns orthogonal to ERIM because neither limitation is relevant to ERIM's use of MPK. libmpk could be combined with ERIM in applications that require more than 16 components, but the integration remains as future work.

In recent work, Burow *et al.* [11] survey implementation techniques for shadow stacks. In particular, they examine the use of MPK for protecting the integrity of shadow stacks. Burow *et al.*'s measurements of MPK overheads (Fig. 10 in [11]) are consistent with ours. Their use of MPK could be a specific use-case for ERIM, which is a more general framework for memory isolation.

3 Design

Goals ERIM enables efficient data isolation within a user-space process. Like prior work, it enables a (trusted) application component to isolate its sensitive data from untrusted components. Unlike prior work, ERIM supports such isolation with *low overhead* even at *high switching rates* between components *without requiring control-flow integrity*. In the following, we focus on the case of two components that are isolated from each other within a single-threaded process. Later, we describe generalizations to multi-threaded processes, more than two components per process, and read-only sharing among components.

We use the letter T to denote a trusted component and U to denote the remaining, untrusted application component. ERIM's key primitive is memory isolation: it reserves a region of the address space and makes it accessible exclusively from the trusted component T. This reserved region is denoted M_T and can be used by T to store sensitive data. The rest of the address space, denoted M_U, holds the application's regular heap and stack and is accessible from both U and T. ERIM enforces the following invariants:
(1) While control is in U, access to M_T remains disabled.
(2) Access to M_T is enabled atomically with a control transfer to a designated entry point in T and disabled when T transfers control back to U.
The first invariant provides isolation of M_T from U, while the second invariant prevents U from confusing T into accessing M_T improperly by jumping into the middle of M_T's code.

Background: Intel MPK To realize its goals, ERIM uses the recent MPK extension to the x86 ISA [28]. With MPK, each virtual page of a process can be associated with one of 16 protection keys, thus partitioning the address space into up to 16 *domains*. A new register, PKRU, that is local to each logical core, determines the current access permissions (read, write, neither or both) on each domain for the code running on that core. Access checks against the PKRU are implemented in hardware and impose no overhead on program execution.

Changing access privileges requires writing new permissions to the PKRU register with a *user-mode* instruction, WRPKRU. This instruction is relatively fast (11–260 cycles on current Intel CPUs), does not require a syscall, changes to page tables, a TLB flush, or inter-core synchronization.

The PKRU register can also be modified by the XRSTOR instruction by setting a specific bit in the eax register prior to the instruction (XRSTOR is used to restore the CPU's previously-saved extended state during a context switch).

For strong security, ERIM must ensure that untrusted code cannot exploit WRPKRU or XRSTOR instructions in executable pages to elevate privileges. To this end, ERIM combines MPK with binary inspection to ensure that all executable occurrences of WRPKRU or XRSTOR are *safe*, i.e., they cannot be exploited to improperly elevate privilege.

Background: Linux support for MPK As of version 4.6, the mainstream Linux kernel supports MPK. Page-table entries are tagged with MPK domains, there are additional syscall options to associate pages with specific domains, and the PKRU register is saved and restored during context switches. Since hardware PKRU checks are disabled in kernel mode, the kernel checks PKRU permissions explicitly before dereferencing any userspace pointer. To avoid executing a signal handler with inappropriate privileges, the kernel updates the PKRU register to its initial set of privileges (access only to domain 0) before delivering a signal to a process.

3.1 High-level design overview

ERIM can be configured to provide either complete isolation of M_T from U (confidentiality and integrity), or only write protection (only integrity). We describe the design for complete isolation first. Section 3.7 explains a slight design re-configuration that provides only write protection.

ERIM's isolation mechanism is conceptually simple: It maps T's reserved memory, M_T, and the application's general memory, M_U, to two different MPK domains. It manages MPK permissions (the PKRU registers) to ensure that M_U is always accessible, while only M_U is accessible when control is in U. It allows U to securely transfer control to T and back via *call gates*. A call gate enables access to M_T using the WRPKRU instruction and immediately transfers control to a specified entry point of T, which may be an explicit or inlined function. When T is done executing, the call gate disables access to M_T and returns control to U. This enforces ERIM's two invariants (1) and (2) from Section 3. Call gates operate entirely in user-mode (they don't use syscalls) and are described in Section 3.3.

Preventing exploitation A key difficulty in ERIM's design is preventing the untrusted U from exploiting occurrences of the WRPKRU or XRSTOR instruction sequence on executable pages to elevate its privileges. For instance, if the sequence appeared at any byte address on an executable page, it could be exploited using control-flow hijack attacks. To prevent such exploits, ERIM relies on *binary inspection* to enforce the invariant that only *safe* WRPKRU and XRSTOR occurrences appear on executable pages.

A WRPKRU occurrence is safe if it is immediately followed by one of the following: (A) a pre-designated entry point of T, or (B) a specific sequence of instructions that checks that the permissions set by WRPKRU do not include access to M_T and terminates the program otherwise. A safe WRPKRU occurrence cannot be exploited to access M_T inappropriately. If the occurrence satisfies (A), then it does not give control to U at all; instead, it enters T at a designated entry point. If the occurrence satisfies (B), then it would terminate the program immediately when exploited to enable access to M_T.

A XRSTOR is safe if it is immediately followed by a specific sequence of instructions to check that the eax bit that

causes XRSTOR to load the PKRU register is not set. Such a XRSTOR cannot be used to change privilege and continue execution.[1]

ERIM's call gates use only safe WRPKRU occurrences (and do not use XRSTOR at all). So, they pass the binary inspection. Section 3.4 describes ERIM's binary inspection.

Creating safe binaries An important question is how to construct binaries that do not have unsafe WRPKRUs and XRSTORs. On x86, these instructions may arise inadvertently spanning the bytes of adjacent instructions or as a subsequence in a longer instruction. To eliminate such inadvertent occurrences, we describe a binary rewriting mechanism that rewrites any sequence of instructions containing a WRPKRU or XRSTOR to a functionally equivalent sequence without any WRPKRUs and XRSTORs. The mechanism can be deployed as a compiler pass or integrated with our binary inspection, as explained in Section 4.

3.2 Threat model

ERIM makes no assumptions about the untrusted component (U) of an application. U may behave arbitrarily and may contain memory corruption and control-flow hijack vulnerabilities that may be exploited during its execution.

However, ERIM assumes that the trusted component T's binary does not have such vulnerabilities and does not compromise sensitive data through explicit information leaks, by calling back into U while access to M_T is enabled, or by mapping executable pages with unsafe/exploitable occurrences of the WRPKRU or XRSTOR instruction.

The hardware, the OS kernel, and a small library added by ERIM to each process that uses ERIM are trusted to be secure. We also assume that the kernel enforces standard DEP—an executable page must not be simultaneously mapped with write permissions. ERIM relies on a list of legitimate entry points into T provided either by the programmer or the compiler, and this list is assumed to be correct (see Section 3.4). The OS's dynamic program loader/linker is trusted to invoke ERIM's initialization function before any other code in a new process.

Side-channel and rowhammer attacks, and microachitectural leaks, although important, are beyond the scope of this work. However, ERIM is compatible with existing defenses. Our current *prototype* of ERIM is incompatible with applications that simultaneously use MPK for other purposes, but this is not fundamental to ERIM's design. Such incompatibilities can be resolved as long as the application does not re-use the MPK domain that ERIM reserves for T.

3.3 Call gates

A call gate transfers control from U to T by enabling access to M_T and executing from a designated entry point of T, and later returns control to U after disabling access to M_T. This requires two WRPKRUs. The primary challenge in designing the call gate is ensuring that both these WRPKRUs are safe in the sense explained in Section 3.1.

```
  xor ecx, ecx                                  1
  xor edx, edx                                  2
  mov PKRU_ALLOW_TRUSTED, eax                   3
  WRPKRU // copies eax to PKRU                  4

// Execute trusted component's code           6

  xor ecx, ecx                                  8
  xor edx, edx                                  9
  mov PKRU_DISALLOW_TRUSTED, eax                10
  WRPKRU // copies eax to PKRU                  11
  cmp PKRU_DISALLOW_TRUSTED, eax                12
  je continue                                   13
  syscall exit // terminate program             14
continue:                                       15
// control returns to the untrusted            16
    application here
```

Listing 1: Call gate in assembly. The code of the trusted component's entry point may be inlined by the compiler on line 6, or there may be an explicit direct call to it.

Listing 1 shows the assembly code of a call gate. WRPKRU expects the new PKRU value in the eax register and requires ecx and edx to be 0. The call gate works as follows. First, it sets PKRU to enable access to M_T (lines 1–4). The macro PKRU_ALLOW_TRUSTED is a constant that allows access to M_T and M_U.[2] Next, the call gate transfers control to the designated entry point of T (line 6). T's code may be invoked either by a direct call, or it may be inlined.

After T has finished, the call gate sets PKRU to disable access to M_T (lines 8–11). The macro PKRU_DISALLOW_TRUSTED is a constant that allows access to M_U but not M_T. Next, the call gate checks that the PKRU was actually loaded with PKRU_DISALLOW_TRUSTED (line 12). If this is not the case, it terminates the program (line 14), else it returns control to U (lines 15–16). The check on line 12 may seem redundant since eax is set to PKRU_DISALLOW_TRUSTED on line 10. However, the check prevents *exploitation* of the WRPKRU on line 11 by a control-flow hijack attack (explained next).

Safety Both occurrences of WRPKRU in the call gate are safe. Neither can be exploited by a control flow hijack to get unauthorized access to M_T. The first occurrence of WRPKRU (line 4) is immediately followed by (a direct control transfer to) a designated entry point of T. This instance can-

[1] We know of only one user-mode Linux application – the dynamic linker, `ld`, that legitimately uses XRSTOR. However, `ld` categorically does *not* restore PKRU through XRSTOR, so this safe check can be added to it.

[2] To grant read (resp. write) access to domain i, bit $2i$ (resp. $2i+1$) must be set in the PKRU. PKRU_ALLOW_TRUSTED sets the 4 least significant bits to grant read and write access to domains 0 (M_U) and 1 (M_T).

not be exploited to transfer control to anywhere else. The second occurrence of WRPKRU (line 11) is followed by a check that terminates the program if the new permissions include access to M_T. If, as part of an attack, the execution jumped directly to line 11 with any value other than PKRU_DISALLOW_TRUSTED in eax, the program would be terminated on line 14.

Efficiency A call gate's overhead on a roundtrip from U to T is two WRPKRUs, a few very fast, standard register operations and one conditional branch instruction. This overhead is very low compared to other hardware isolation techniques that rely on pages tables and syscalls or hypervisor trampolines to change privileges (see also Section 6.5).

Use considerations ERIM's call gate omits features that readers may expect. These features have been omitted to avoid having to pay their overhead when they are not needed. First, the call gate does not include support to pass parameters from U to T or to pass a result from T to U. These can be passed via a designated shared buffer in M_U (both U and T have access to M_U). Second, the call gate does not scrub registers when switching from T to U. So, if T uses confidential data, it should scrub any secrets from registers before returning to U. Further, because T and U share the call stack, T must also scrub secrets from the stack prior to returning. Alternatively, T can allocate a private stack for itself in M_T, and T's entry point can switch to that stack immediately upon entry. This prevents T's secrets from being written to U's stack in the first place. (A private stack is also necessary for multi-threaded applications; see Section 3.7).

3.4 Binary inspection

Next, we describe ERIM's binary inspection. The inspection prevents U from mapping any executable pages with unsafe WRPKRU and XRSTOR occurrences and consists of two parts: (i) an inspection function that verifies that a sequence of pages does not contain unsafe occurrences; and, (ii) an interception mechanism that prevents U from mapping executable pages without inspection.

Inspection function The inspection function *scans* a sequence of pages for instances of WRPKRU and XRSTOR. It also inspects any adjacent executable pages in the address space for instances that cross a page boundary.

For every WRPKRU, it checks that the WRPKRU is safe, i.e., either condition (A) or (B) from Section 3.1 holds. To check for condition (A), ERIM needs a list of designated entry points of T. The source of this list depends on the nature of T and is trusted. If T consists of library functions, then the programmer marks these functions, e.g., by including a unique character sequence in their names. If the functions are not inlined by the compiler, their names will appear in the symbol table. If T's functions are subject to inlining or if they are generated by a compiler pass, then the compiler must be directed to add their entry locations to the symbol table with the unique character sequence. In all cases, ERIM can identify designated entry points by looking at the symbol table and make them available to the inspection function.

Condition (B) is checked easily by verifying that the WRPKRU is immediately followed by *exactly* the instructions on lines 12–15 of Listing 1. These instructions ensure that the WRPKRU cannot be used to enable access to M_T and continue execution.

For every XRSTOR, the inspection function checks that the XRSTOR is followed immediately by the following instructions, which check that the eax bit that causes XRSTOR to load PKRU (bit 9) is not set: `bt eax, 0x9; jnc .safe; EXIT; .safe:...`. Here, `EXIT` is a macro that exits the program. Trivially, such a XRSTOR cannot be used to enable access to M_T and continue execution.

Interception On recent ($\geq$4.6) versions of Linux, interception can be implemented *without kernel changes*. We install a seccomp-bpf filter [29] that catches mmap, mprotect, and pkey_mprotect syscalls which attempt to map a region of memory as executable (mode argument `PROT_EXEC`). Since the bpf filtering language currently has no provisions for reading the PKRU register, we rely on seccomp-bpf's `SECCOMP_RET_TRACE` option to notify a ptrace()-based tracer process. The tracer inspects the tracee and allows the syscall if it was invoked from T and denies it otherwise. The tracer process is configured so that it traces any child of the tracee process as well. While ptrace() interception is expensive, note that it is required only when a program maps pages as executable, which is normally an infrequent operation.

If programs map executable pages frequently, a more efficient interception can be implemented with a simple Linux Security Module (LSM) [50], which allows mmap, mprotect and pkey_mprotect system calls only from T. (Whether such a call is made by U or T is easily determined by examining the PKRU register value at the time of the syscall.) Our prototype uses this implementation of interception. Another approach is to implement a small (8 LoC) change to seccomp-bpf in the Linux kernel, which allows a bpf filter to inspect the value of the PKRU register. With this change in place, we can install a bpf filter that allows certain syscalls only from T, similar to the LSM module.

With either interception approach in place, U must go through T to map executable pages. T maps the pages only after they have passed the inspection function. Regardless of the interception method, pages can be inspected upfront when T attempts to map them as executable, or on demand when they are executed for the first time.

On-demand inspection is preferable when a program maps a large executable segment but eventually executes only a small number of pages. With on-demand inspection, when the process maps a region as executable, T instead maps the region read-only but records that the pages are pending inspection. When control transfers to such a page, a fault occurs. The fault traps to a dedicated signal handler, which

ERIM installs when it initializes (the LSM or the tracer prevents U from overriding this signal handler). This signal handler calls a T function that checks whether the faulting page is pending inspection and, if so, inspects the page. If the inspection passes, then the handler remaps the page with the execute permission and resumes execution of the faulting instruction, which will now succeed. If not, the program is terminated.

The interception and binary inspection has very low overhead in practice because it scans an executable page at most once. It is also fully transparent to U's code if all WRPKRUs and XRSTORs in the binary are already safe.

Security We briefly summarize how ERIM attains security. The binary inspection mechanism prevents U from mapping any executable page with an unsafe WRPKRU or XRSTOR. T does not contain any executable unsafe WRPKRU or XRSTOR by assumption. Consequently, only safe WRPKRUs and XRSTORs are executable in the entire address space at any point. Safe WRPKRUs and XRSTORs preserve ERIM's two security invariants (1) and (2) by design. Thus M_T is accessible only while T executes starting from legitimate T entry points.

3.5 Lifecycle of an ERIM process

As part of a process's initialization, before control is transferred to main(), ERIM creates a second MPK memory domain for M_T in addition to the process's default MPK domain, which is used for M_U. ERIM maps a memory pool for a dynamic memory allocator to be used in M_T and hooks dynamic memory allocation functions so that invocations are transparently redirected to the appropriate pool based on the value of the PKRU register. This redirection provides programmer convenience but is not required for security. If U were to call T's allocator, it would be unable to access M_T's memory pool and generate a page fault. Next, ERIM scans M_U's executable memory for unsafe WRPKRUs and XRSTORs, and installs one of the interception mechanisms described in Section 3.4. Finally, depending on whether main() is in U or T, ERIM initializes the PKRU register appropriately and transfers control to main(). After main() has control, the program executes as usual. It can map, unmap and access data memory in M_U freely. However, to access M_T, it must invoke a call gate.

3.6 Developing ERIM applications

We describe here three methods of developing applications or modifying existing applications to use ERIM.

The *binary-only* approach requires that either U or T consist of a set of functions in a dynamic link library. In this case, the library and the remaining program can be used in unmodified binary form. An additional ERIM dynamic wrapper library is added using LD_PRELOAD, which wraps the entry points with stub functions that implement the call gates and have names that indicate to the ERIM runtime the valid entry points. We have used this approach to isolate SQLite within the Node.js runtime (Section 5).

```
typedef struct secret {                          1
  int number; } secret;                          2
secret* initSecret() {                           3
  ERIM_SWITCH_T;                                 4
  secret * s = malloc(sizeof(secret));           5
  s->number = random();                          6
  ERIM_SWITCH_U;                                 7
  return s;                                      8
}                                                9
int compute(secret* s, int m) {                  10
  int ret = 0;                                   11
  ERIM_SWITCH_T;                                 12
  ret = f(s->number, m);                         13
  ERIM_SWITCH_U;                                 14
  return ret;                                    15
}                                                16
```

Listing 2: C component isolated with ERIM

The *source* approach requires that either U or T consist of a set of functions that are not necessarily in a separate compilation unit or library. In this case, the source code is modified to wrap these functions with stubs that implement the call gates, and choose names that indicate valid entry points. We used this approach to isolate the crypto functions and session keys in OpenSSL (Section 5).

The *compiler* approach requires modifications to the compiler to insert call gates at appropriate points in the executable and generate appropriate symbols that indicate valid entry points. This approach is the most flexible because it allows arbitrary inlining of U and T code. We used this approach to isolate the metadata in CPI (Section 5).

Next, we give a simple example describing the process of developing a new C application using the *source* approach. ERIM provides a C library and header files to insert call gates, initialize ERIM, and support dynamic memory allocation. Listing 2 demonstrates an example C program that isolates a data structure called `secret` (lines 1–2). The structure contains an integer value. Two functions, `initSecret` and `compute`, access secrets and bracket their respective accesses with call gates using the macros `ERIM_SWITCH_T` and `ERIM_SWITCH_U`. ERIM isolates `secret` such that only code that appears between `ERIM_SWITCH_T` and `ERIM_SWITCH_U`, i.e., code in T, may access `secret`. `initSecret` allocates an instance of `secret` while executing inside T by first allocating memory in M_T and then initializing the secret value. `compute` computes a function `f` of the secret inside T.

3.7 Extensions

Next, we discuss extensions to ERIM's basic design.

Multi-threaded processes ERIM's basic design works as-is with multi-threaded applications. Threads are created as usual, e.g. using libpthread. The PKRU register is saved and restored by the kernel during context switches. However, multi-threading imposes an additional requirement on T (not on ERIM): In a multi-threaded application, it is essential that T allocate a private stack in M_T (not M_U) for each thread and execute its code on these stacks. This is easy to implement by switching stacks at T's entry points. Not doing so and executing T on standard stacks in M_U runs the risk that, while a thread is executing in T, another thread executing in U may corrupt or read the first thread's stack frames. This can potentially destroy T's integrity, leak its secrets and hijack control while access to M_T is enabled. By executing T's code on stacks in M_T, such attacks are prevented.

More than two components per process Our description of ERIM so far has been limited to two components (T and U) per process. However, ERIM generalizes easily to support as many components as the number of domains Linux's MPK support can provide (this could be less than 16 because the kernel may reserve a few domains for specific purposes). Components can have arbitrary pairwise trust relations with each other, as long as the trust relations are transitive. A simple setting could have a default domain that trusts all other domains (analogous to U) and any number of additional domains that do not trust any others. ERIM's initialization code creates a private heap for each component, and ERIM's custom allocator allocates from the heap of the currently executing component. Each component can also (in its own code) allocate a per-thread stack, to protect stack-allocated sensitive data when calling into other untrusted domains. Stacks can be mandatorily switched by ERIM's call gates.

ERIM for integrity only Some applications care only about the integrity of protected data, but not its confidentiality. Examples include CPI, which needs to protect only the integrity of code pointers. In such applications, efficiency can be improved by allowing U to *read* M_T directly, thus avoiding the need to invoke a call gate for reading M_T. The ERIM design we have described so far can be easily modified to support this case. Only the definition of the constant PKRU_DISALLOW_TRUSTED in Listing 1 has to change to also allow read-only access to M_T. With this change, read access to M_T is always enabled.

Just-in-time (jit) compilers with ERIM ERIM works with jit compilers that follow standard DEP and do not allow code pages that are writable and executable at the same time. Such jit compilers write new executable code into newly allocated, non-executable pages and change these pages' permissions to non-writable and executable once the compilation finishes. ERIM's mprotect interception defers enabling execute permissions until after a binary inspection, as described in Section 3.4. When a newly compiled page is executed for the first time, ERIM handles the page execute permission fault, scans the new page for unsafe WRPKRUs/XRSTORs and enables the execute permission if no unsafe occurrences exist. This mechanism is safe, but may lead to program crashes if the jit compiler accidentally emits an unsafe WRPKRU or XRSTOR. ERIM-aware jit compilers can emit WRPKRU- and XRSTOR-free binary code by relying on the rewrite strategy described in Section 4, and inserting call gates when necessary.

OS privilege separation The design described so far provides memory isolation. Some applications, however, require privilege separation between T and U with respect to OS resources. For instance, an application might need to restrict the filesystem name space accessible to U or restrict the system calls available to U.

ERIM can be easily extended to support privilege separation with respect to OS resources, using one of the techniques described in Section 3.4 for intercepting systems calls that map executable pages. In fact, intercepting and disallowing these system calls when invoked from U is just a special case of privilege separation. During process initialization, ERIM can instruct the kernel to restrict U's access rights. After this, the kernel refuses to grant access to restricted resources whenever the value of the PKRU is not PKRU_ALLOW_TRUSTED, indicating that the syscall does not originate from T. To access restricted resources, U must invoke T, which can filter syscalls.

4 Rewriting program binaries

The binary inspection described in Section 3.4 guarantees that executable pages do not contain unsafe instances of the WRPKRU and XRSTOR instructions. This is *sufficient* for ERIM's safety. In this section, we show how to generate or modify program binaries to not contain unsafe WRPKRUs and XRSTORs, so that they pass the binary inspection.

Intentional occurrences of WRPKRU that are not immediately followed by a transfer to T and all occurrences of XRSTOR, whether they are generated by a compiler or written manually in assembly, can be made safe by inserting the checks described in Section 3.4 after the instances. Inadvertent occurrences—those that arise unintentionally as part of a longer x86 instruction and operand, or spanning two consecutive x86 instructions/operands—are more interesting. We describe a rewrite strategy to eliminate such occurrences and how the strategy can be applied by a compiler or a binary rewriting tool. The strategy can rewrite any sequence of x86 instructions and operands containing an inadvertent WRPKRU or XRSTOR to a functionally equivalent sequence without either. In the following we describe the strategy, briefly argue why it is complete, and summarize an empirical evaluation of its effectiveness.

Rewrite strategy WRPKRU is a 3 byte instruction, 0x0F01EF. XRSTOR is also always a 3-byte instruction, but it has more variants, fully described by the regular expres-

Overlap with	Cases	Rewrite strategy	ID	Example
Opcode	Opcode = WRPKRU/ XRSTOR	Insert safety check after instruction	1	
Mod R/M	Mod R/M = 0x0F	Change to unused register + move command	2	add ecx, [ebx + 0x01EF0000] → mov eax, ebx; add ecx, [eax + 0x01EF0000];
		Push/Pop used register + move command	3	add ecx, [ebx + 0x01EF0000] → push eax; mov eax, ebx; add ecx, [eax + 0x01EF0000]; pop eax;
Displacement	Full/Partial sequence	Change mode to use register	4	add eax, 0x0F01EF00 → (push ebx;) mov ebx, 0x0F010000; add ebx, 0x0000EA00; add eax, ebx; (pop ebx;)
	Jump-like instruction	Move code segment to alter constant used in address	5	call [rip + 0x0F01EF00] → call [rip + 0x0FA0EEFF]
Immediate	Full/Partial sequence	Change mode to use register	6	add eax, 0x0F01EF → (push ebx;) mov ebx, 0x0F01EE00; add ebx, 0x00000100; add eax, ebx; (pop ebx;)
	Associative opcode	Apply instruction twice with different immediates to get equivalent effect	7	add ebx, 0x0F01EF00 → add ebx, 0x0E01EF00; add ebx, 0x01000000

Table 1: Rewrite strategy for intra-instruction occurrences of WRPKRU and XRSTOR

sion 0x0FAE[2|6|A][8-F]. There are two cases to consider. First, a WRPKRU or XRSTOR sequence can span two or more x86 instructions. Such sequences can be "broken" by inserting a 1-byte nop like 0x90 between the two consecutive instructions. 0x90 does not coincide with any individual byte of WRPKRU or XRSTOR, so this insertion cannot generate a new occurrence.

Second, a WRPKRU or XRSTOR may appear entirely within a longer instruction including any immediate operand. Such cases can be rewritten by replacing them with a semantically equivalent instruction or sequence of instructions. Doing so systematically requires an understanding of x86 instruction coding. An x86 instruction contains: (i) an opcode field possibly with prefix, (ii) a MOD R/M field that determines the addressing mode and includes a register operand, (iii) an optional SIB field that specifies registers for indirect memory addressing, and (iv) optional displacement and/or immediate fields that specify constant offsets for memory operations and other constant operands.

The strategy for rewriting an instruction depends on the fields with which the WRPKRU or XRSTOR subsequence overlaps. Table 1 shows the complete strategy.

An opcode field is at most 3-bytes long. If the WRPKRU (XRSTOR) starts at the first byte, the instruction *is* WRPKRU (XRSTOR). In this case, we make the instruction safe by inserting the corresponding check from Section 3.4 after it. If the WRPKRU or XRSTOR starts after the first byte of the opcode, it must also overlap with a later field. In this case, we rewrite according to the rule for that field below.

If the sequence overlaps with the MOD R/M field, we change the register in the MOD R/M field. This requires a free register. If one does not exist, we rewrite to push an existing register to the stack, use it in the instruction, and pop it back. (See lines 2 and 3 in Table 1.)

If the sequence overlaps with the displacement or the immediate field, we change the mode of the instruction to use a register instead of a constant. The constant is computed in the register before the instruction (lines 4 and 6). If a free register is unavailable, we push and pop one. Two instruction-specific optimizations are possible. First, for jump-like instructions, the jump target can be relocated in the binary; this changes the displacement in the instruction, obviating the need a free register (line 5). Second, associative operations like addition can be performed in two increments without an extra register (line 7). Rewriting the SIB field is never required because any WRPKRU or XRSTOR must overlap with at least one non-SIB field (the SIB field is 1 byte long while these instructions are 3 bytes long).

Compilers and well-written assembly programs normally do not mix data like constants, jump tables, etc. with the instruction stream and instead place such data in a non-executable data segment. If so, WRPKRU or XRSTOR sequences that occur in such data can be ignored.

Compiler support For binaries that can be recompiled from source, rewriting can be added to the codegen phase of the compiler, which converts the intermediate representation (IR) to machine instructions. Whenever codegen outputs an inadvertent WRPKRU or XRSTOR, the surrounding instructions in the IR can be replaced with equivalent instructions as described above, and codegen can be run again.

Runtime binary rewriting For binaries that cannot be recompiled, binary rewriting can be integrated with the interception and inspection mechanism (Section 3.4). When the inspection discovers an unsafe WRPKRU or XRSTOR on an executable page during its scan, it overwrites the page with

1-byte traps, makes it executable, and stores the original page in reserve without enabling it for execution. Later, if there is a jump into the executable page, a trap occurs and the trap handler discovers an entry point into the page.

The rewriter then disassembles the *reserved page* from that entry point on, rewriting any discovered WRPKRU or XRSTOR occurrences, and copies the rewritten instruction sequences back to the executable page. To prevent other threads from executing partially overwritten instruction sequences, we actually rewrite a fresh copy of the executable page with the new sequences, and then swap this rewritten copy for the executable page. This technique is transparent to the application, has an overhead proportional to the number of entry points in offending pages (it disassembles from every entry point only once) and maintains the invariant that only safe pages are executable.

A rewritten instruction sequence is typically longer than the original sequence and therefore cannot be rewritten in-place. In this case, binary rewriting tools place the rewritten sequence on a new page, replace the first instruction in the original sequence with a direct jump to the rewritten sequence, and insert a direct jump back to the instruction following the original sequence after the rewritten sequence. Both pages are then enabled for execution.

Implementation and testing The rewrite strategy is arguably complete. We have implemented the strategy as a library, which can be used either with the inspection mechanism as explained above or with a *static* binary rewrite tool, as described here. To gain confidence in our implementation, we examined all binaries of five large Linux distributions (a total of 204,370 binaries). Across all binaries, we found a total of 1213 WRPKRU/XRSTOR occurrences in code segments. We then used a standard tool, Dyninst [15], to try to disassemble and rewrite these occurrences. Dyninst was able to disassemble 1023 occurrences and, as expected, our rewriter rewrote all instances successfully. Next, we wanted to run these 1023 rewritten instances. However, this was infeasible since we did not know what inputs to the binaries would cause control to reach the rewritten instances. Hence, we constructed two hand-crafted binaries with WRPKRUs/XRSTORs similar to the 1023 occurrences, rewrote those WRPKRUs/XRSTORs with Dyninst and checked that those rewritten instances ran correctly. Based on these experiments, we are confident that our implementation of WRPKRU/XRSTOR rewriting is robust.

5 Use Cases

ERIM goes beyond prior work by providing efficient isolation with very high component switch rates of the order of 10^5 or 10^6 times a second. We describe three such use cases here, and report ERIM's overhead on them in Section 6.

Isolating cryptographic keys in web servers Isolating *long-term* SSL keys to protect from web server vulnerabilities such as the Heartbleed bug [37] is well-studied [33, 34]. However, long-term keys are accessed relatively infrequently, typically only a few times per user session. *Session keys*, on the other hand, are accessed far more frequently—over 10^6 times a second per core in a high throughput web server like NGINX. Isolating sessions keys is relevant because these keys protect the confidentiality of individual users. With its low-cost switching, ERIM can be used to isolate session keys efficiently. To verify this, we partitioned OpenSSL's low-level crypto library (libcrypto) to isolate the session keys and basic crypto routines, which run as T, from the rest of the web server, which runs as U.

Native libraries in managed runtimes Managed runtimes such as a Java or JavaScript VM often rely on third-party native libraries written in unsafe languages for performance. ERIM can isolate the runtime from bugs and vulnerabilities in a native library by mapping the managed runtime to T and the native libraries to U. This use case leverages the "integrity only" version of ERIM (Section 3.7). We isolated Node.js from a native SQLite plugin. Node.js is a state-of-the-art managed runtime for JavaScript and SQLite is a state-of-the-art database library written in C [1, 2]. The approach generalizes to isolating several mutually distrusting libraries from each other by leveraging ERIM's multi-component extension from Section 3.7.

CPI/CPS Code-pointer integrity (CPI) [31] prevents control-flow hijacks by isolating sensitive objects—code pointers and objects that can lead to code pointers—in a *safe region* that cannot be written without bounds checks. CPS is a lighter, less-secure variant of CPI that isolates only code pointers. A key challenge is to isolate the safe region efficiently, as CPI can require switching rates on the order of 10^6 or more switches/s on standard benchmarks. We show that ERIM can provide strong isolation for the safe region at low cost. To do this, we override the CPI/CPS-enabled compiler's intrinsic function for writing the sensitive region to use a call gate around an inlined sequence of T code that performs a bounds check before the write. (MemSentry [30] also proposes using MPK for isolating the safe region, but does not actually implement it.)

6 Evaluation

We have implemented two versions of an ERIM prototype for Linux.[3] One version relies on a 77 line Linux Security Module (LSM) that intercepts all mmap and mprotect calls to prevent U from mapping pages in executable mode, and prevents U from overriding the binary inspection handler. We additionally added 26 LoC for kernel hooks to Linux v4.9.110, which were needed by the LSM. We also implemented ERIM on an *unmodified* Linux kernel using the ptrace-based technique described in Section 3.4. In the

[3] Available online at `https://gitlab.mpi-sws.org/vahldiek/erim`.

following, we show results obtained with the modified kernel. The performance of ERIM on the stock Linux kernel is similar, except that the costs of `mmap`, `mprotect`, and `pkey_mprotect` syscalls that enable execute permissions are about 10x higher. Since the evaluated applications use these operations infrequently, the impact on their overall performance is negligible.

Our implementation also includes the ERIM runtime library, which provides a memory allocator over M_T, call gates, the ERIM initialization code, and binary inspection. These comprise 569 LoC. Separately, we have implemented the rewriting logic to eliminate inadvertent WRPKRU occurrences (about 2250 LoC). While we have not yet integrated the logic into either a compiler or our inspection handler, the binaries used in our performance evaluation experiments do not have any unsafe WRPKRU occurrences and do not load any libraries at runtime. However, the binaries did have two legitimate occurrences of XRSTOR (in the dynamic linker library `ld.so`), which we made safe as described in Section 3.4. Two other inadvertent XRSTOR occurred in data-only pages of executable segments in `libm`, which is used by the SPEC benchmarks. We made these safe by re-mapping the pages read-only. Hence, the results we report are on completely safe binaries.

We evaluate the ERIM prototype on microbenchmarks and on the three applications mentioned in Section 5. Unless otherwise mentioned, we perform our experiments on Dell PowerEdge R640 machines with 16-core MPK-enabled Intel Xeon Gold 6142 2.6GHz CPUs (with the latest firmware; Turbo Boost and SpeedStep were disabled), 384GB memory, 10Gbps Ethernet links, running Debian 8, Linux kernel v4.9.60. For the OpenSSL/webserver experiments in Sections 6.2 and 6.5, we use NGINX v1.12.1, OpenSSL v1.1.1 and the ECDHE-RSA-AES128-GCM-SHA256 cipher. For the managed language runtime experiment (Section 6.3), we use Node.js v9.11.1 and SQLite v3.22.0. For the CPI experiment (Section 6.4), we use the Levee prototype v0.2 available from `http://dslab.epfl.ch/proj/cpi/` and Clang v3.3.1 including its CPI compile pass, runtime library extensions and link-time optimization.

6.1 Microbenchmarks

Switch cost We performed a microbenchmark to measure the overhead of invoking a function with and without a switch to a trusted component. The function adds a constant to an integer argument and returns the result. Table 2 shows the cost of invoking the function, in cycles, as an inlined function (I), as a directly called function (DC), and as a function called via a function pointer (FP). For reference, the table also includes the cost of a simple syscall (getpid), the cost of a switch on lwCs, a recent isolation mechanism based on kernel page table protections [33], and the cost of a VMFUNC (Intel VT-x)-based extended page table switch.

In our microbenchmark, calls with an ERIM switch are between 55 and 80 cycles more expensive than their no-switch counterparts. The most expensive indirect call costs less than the simplest system call (getpid). ERIM switches are up to 3-5x faster than VMFUNC switches and up to 100x faster than lwC switches.

Call type	Cost (cycles)
Inlined call (no switch)	5
Direct call (no switch)	8
Indirect call (no switch)	19
Inlined call + switch	60
Direct call + switch	69
Indirect call + switch	99
getpid system call	152
Call + VMFUNC EPT switch	332
lwC switch [33] (Skylake CPU)	6050

Table 2: Cycle counts for basic call and return

Because the CPU must not reorder loads and stores with respect to a WRPKRU instruction, the overhead of an ERIM switch depends on the CPU pipeline state at the time the WRPKRUs are executed. In experiments described later in this section, we observed average overheads ranging from 11 to 260 cycles per switch. At a clock rate of 2.6GHz, this corresponds to overheads between 0.04% and 1.0% for 100,000 switches per second, which is significantly lower than the overhead of any kernel- or hypervisor-based isolation.

Binary inspection To determine the cost of ERIM's binary inspection, we measured the cost of scanning the binaries of all 18 applications in the CINT/FLOAT SPEC 2006 CPU benchmark. These range in size from 9 to 3918 4KB pages, contain between 35 and 63765 intentional WRPKRU instructions when compiled with CPI (see Section 6.4), no unintended WRPKRU and no XRSTOR instructions. The overhead is largely independent of the number of WRPKRU instructions and ranges between 3.5 and 6.2 microseconds per page. Even for the largest binary, the scan takes only 17.7ms, a tiny fraction of a typical process' runtime.

6.2 Protecting session keys in NGINX

Next, we use ERIM to isolate SSL session keys in a high performance web server, NGINX. We configured NGINX to use only the ECDHE-RSA-AES128-GCM-SHA256 cipher and AES encryption for sessions. We modified OpenSSL's libcrypto to isolate all session keys and the functions for AES key allocation and encryption/decryption into ERIM's T, and use ERIM call gates to invoke these functions.

To measure ERIM's overhead on the peak throughput, we configure a single NGINX worker pinned to a CPU core, and connect to it remotely over HTTPS with keep-alive from 4 concurrent ApacheBench (`ab`) [3] instances each simulating 75 concurrent clients. The clients all request the same file, whose size we vary from 0 to 128KB across experi-

File size (KB)	1 worker		3 workers		5 workers		10 workers	
	Native (req/s)	ERIM rel. (%)	Native (req/s)	ERIM rel. (%)	Native (req/s)	ERIM rel. (%)	Native (req/s)	ERIM rel. (%)
0	95,761	95.8	276,736	96.1	466,419	95.7	823,471	96.4
1	87,022	95.2	250,565	94.5	421,656	96.1	746,278	95.5
2	82,137	95.4	235,820	95.1	388,926	96.6	497,778	100.0
4	76,562	95.3	217,602	94.9	263,719	100.0		
8	67,855	96.0	142,680	100.0				

Table 3: Nginx throughput with multiple workers. The standard deviation is below 1.5% in all cases.

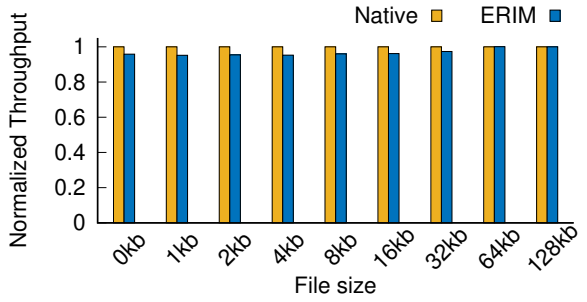


Figure 1: Throughput of NGINX with one worker, normalized to native (no protection), with varying request sizes. Standard deviations were all below 1.1%.

File size (KB)	Throughput		Switches/s	CPU load native (%)
	Native (req/s)	ERIM rel. (%)		
0	95,761	95.8	1,342,605	100.0
1	87,022	95.2	1,220,266	100.0
2	82,137	95.4	1,151,877	100.0
4	76,562	95.3	1,073,843	100.0
8	67,855	96.0	974,780	100.0
16	45,483	97.1	820,534	100.0
32	32,381	97.3	779,141	100.0
64	17,827	100.0	679,371	96.7
128	8,937	100.0	556,152	86.4

Table 4: Nginx throughput with a single worker. The standard deviation is below 1.1% in all cases.

ments.[4] Figure 1 shows the average throughput of 10 runs of an ERIM-protected NGINX relative to native NGINX without any protection for different file sizes, measured after an initial warm-up period.

ERIM-protected NGINX provides a throughput within 95.18% of the unprotected server for all request sizes. To explain the overhead further, we list the number of ERIM switches per second in the NGINX worker and the worker's CPU utilization in Table 4 for request sizes up to 128KB. The overhead shows a general trend up to requests of size 32 KB: The worker's core remains saturated but as the request size increases, the number of ERIM switches per second decrease, and so does ERIM's relative overhead. The observations are consistent with an overhead of about 0.31%–0.44% for 100,000 switches per second. For request sizes 64KB and higher, the 10Gbps network saturates and the worker does not utilize its CPU core completely in the baseline. The free CPU cycles absorb ERIM's CPU overhead, so ERIM's throughput matches that of the baseline.

Note that this is an extreme test case, as the web server does almost nothing and serves the same cached file repeatedly. To get a more realistic assessment, we set up NGINX to serve from main memory static HTML pages from a 571 MB (15,520 pages) Wikipedia snapshot of 2006 [48]. File sizes vary from 417 bytes to 522 KB (average size 37.7 KB). 75 keep-alive clients request random pages (selected based on pageviews on Wikipedia [49]). The average throughput with a single NGINX worker was 22,415 requests/s in the baseline and 21,802 requests/s with ERIM (std. dev. below 0.6% in both cases). On average, there were 615,000 switches a second. This corresponds to a total overhead of 2.7%, or about 0.43% for 100,000 switches a second.

Scaling with multiple workers To verify that ERIM scales with core parallelism, we re-ran the first experiment above with 3, 5 and 10 NGINX workers pinned to separate cores, and sufficient numbers of concurrent clients to saturate all the workers. Table 3 shows the relative overheads with different number of workers. (For requests larger than those shown in the table, the network saturates, and the spare CPU cycles absorb ERIM's overhead completely.) The overheads were independent of the number of workers (cores), indicating that ERIM adds no additional synchronization and scales perfectly with core parallelism. This result is expected as updates to the per-core PKRU do not affect other cores.

6.3 Isolating managed runtimes

Next, we use ERIM to isolate a managed language runtime from an untrusted native library. Specifically, we link the widely-used C database library, SQLite, to Node.js, a state-of-the-art JavaScript runtime and map Node.js's runtime to T and SQLite to U. We modified SQLite's entry points to invoke call gates. To isolate Node.js's stack from SQLite, we run Node.js on a separate stack in M_T, and switch to the

[4] Since NGINX only serves static files in this experiment, its support for Lua and JavaScript is not used. As a result, this experiment does not rely on any support for Jit, which we have not yet implemented.

Test #	Switches/s	ERIM overhead (%)
100	11,183,281	12.73%
110	8,329,914	12.18%
400	8,161,584	15.42%
120	7,190,766	13.81%
142	7,074,553	9.41%
500	6,419,008	12.13%
510	5,868,395	5.60%
410	5,091,212	3.64%
240	2,358,524	3.74%
280	2,303,516	3.22%
170	1,264,366	4.22%
310	1,133,364	2.92%
161	1,019,138	2.81%
160	1,014,829	2.73%
230	670,196	2.04%
270	560,257	2.28%

Table 5: Overhead relative to native execution for SQLite speedtest1 tests with more than 100,000 switches/s. Standard deviations were below 5.6%.

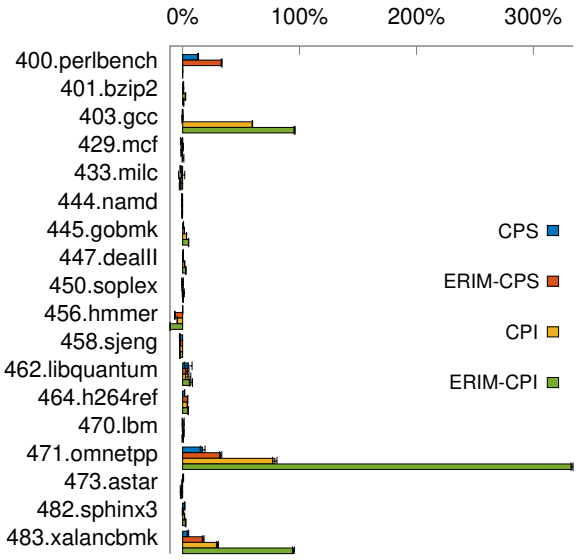


Figure 2: Percentage overhead relative to no protection.

standard stack (in M_U) prior to calling a SQLite function. Finally, SQLite uses the libc function `memmove`, which accesses libc constants that are in M_T, so we implemented a separate `memmove` for SQLite. In total, we added 437 LoC.

We measure overheads on the speedtest1 benchmark that comes with SQLite and emulates a typical database workload [4]. The benchmark performs 32 short tests that stress different database functions like selects, joins, inserts and deletes. We increased the iterations in each test by a factor of four to make the tests longer. Our baseline for comparison is native SQLite linked to Node.js without any protection. We configure the benchmark to store the database in memory and report averages of 20 runs.

The geometric mean of ERIM's runtime overhead across all tests is 4.3%. The overhead is below 6.7% on all tests except those with more than 10^6 switches per second. This suggests that ERIM can be used for isolating native libraries from managed language runtimes with low overheads up to a switching cost of the order of 10^6 per second. Beyond that the overhead is noticeable. Table 5 shows the relative overheads for tests with switching rates of at least 100,000/s. The numbers are consistent with an average overhead between 0.07% and 0.41% for 100,000 switches/s. The actual switch cost measured from direct CPU cycle counts varies from 73 to 260 cycles across all tests. It exceeds 100 cycles only when the switch rate is less than 2,000 times/s. We verified that these are due to i-cache misses—at low switch rates, the call gate instructions are evicted between switches.

6.4 Protecting sensitive data in CPI/CPS

Next, we use ERIM to isolate the safe region of CPI and CPS [31] in a separate domain. We modified CPI/CPS's LLVM compiler pass to emit additional ERIM switches, which bracket any code that modifies the safe region. The switch code, as well as the instructions modifying the safe region, are inlined with the application code. In addition, we implemented simple optimizations to safely reduce the frequency of ERIM domain switches. For instance, the original implementation sets sensitive code pointers to zero during initialization. Rather than generate a domain switch for each pointer initialization, we generate loops of pointer set operations that are bracketed by a single pair of ERIM domain switches. This is safe because the loop relies on direct jumps and the code to set a pointer is inlined in the loop's body. In all, we modified 300 LoC in LLVM's CPI/CPS pass.

Like the original CPI/CPS paper [31], we compare the overhead of the original and our ERIM-protected CPI/CPS system on the SPEC CPU 2006 CINT/FLOAT benchmarks, relative to a baseline compiled with Clang without any protection. The original CPI/CPS system is configured to use ASLR for isolation, the default technique used on x86-64 in the original paper. ASLR imposes almost no switching overhead, but also provides no security [43, 25, 16, 19, 39].

Figure 2 shows the average runtime overhead of 10 runs of the original CPI/CPS (lines "CPI/CPS") and CPI/CPS over ERIM (lines "ERIM-CPI/CPS"). All overheads are normalized to the unprotected SPEC benchmark. We could not obtain results for 400.perlbench for CPI and 453.povray for both CPS and CPI. 400.perlbench does not halt when compiled with CPI and SPEC's result verification for 453.povray fails due to unexpected output. These problems exist in the code generated by the Levee CPI/CPS prototype with CPI/CPS enabled (-fcps/-fcpi), not our modifications.

Benchmark	Switches/sec	ERIM-CPI overhead relative to orig. CPI in %
403.gcc	16,454,595	22.30%
445.gobmk	1,074,716	1.77%
447.dealII	1,277,645	0.56%
450.soplex	410,649	0.60%
464.h264ref	1,705,131	1.22%
471.omnetpp	89,260,024	144.02%
482.sphinx3	1,158,495	0.84%
483.xalancbmk	32,650,497	52.22%

Table 6: Domain switch rates of selected SPEC CPU benchmarks and overheads for ERIM-CPI without binary inspection, *relative to the original CPI with ASLR*.

CPI: The geometric means of the overheads (relative to no protection) of the original CPI and ERIM-CPI across all benchmarks are 4.7% and 5.3%, respectively. The relative overheads of ERIM-CPI are low on all individual benchmarks except gcc, omnetpp, and xalancbmk.

To understand this better, we examined switching rates across benchmarks. Table 6 shows the switching rates for benchmarks that require more than 100,000 switches/s. From the table, we see that the high overheads on gcc, omnetpp and xalancbmk are due to extremely high switching rates on these three benchmarks (between 1.6×10^7 and 8.9×10^7 per second). Further profiling indicated that the reason for the high switch rate is tight loops with pointer updates (each pointer update incurs a switch). An optimization pass could hoist the domain switches out of the loops safely using only direct control flow instructions and enforcing store instructions to be bound to the application memory, but we have not implemented it yet.

Table 6 also shows the overhead of ERIM-CPI excluding binary inspection, relative to the original CPI over ASLR (not relative to an unprotected baseline as in Figure 2). This relative overhead is exactly the cost of ERIM's switching. Depending on the benchmark, it varies from 0.03% to 0.16% for 100,000 switches per second or, equivalently, 7.8 to 41.6 cycles per switch. These results again indicate that ERIM can support inlined reference monitors with switching rates of up to 10^6 times a second with low overhead. Beyond this rate, the overhead becomes noticeable.

CPS: The results for CPS are similar to those for CPI, but the overheads are generally lower. Relative to the baseline without protection, the geometric means of the overheads of the original CPS and ERIM-CPS are 1.1% and 2.4%, respectively. ERIM-CPS's overhead relative to the original CPS is within 2.5% on all benchmarks, except except perlbench, omnetpp and xalancbmk, where it ranges up to 17.9%.

6.5 Comparison to existing techniques

In this section, we compare ERIM to isolation using SFI (with Intel MPX), extended page tables (with Intel VT-x/VMFUNC), kernel page tables (with lwCs), and instrumentation of untrusted code for full memory safety (with WebAssembly). In each case, our primary goal is a *quantitative* comparison of the technique's overhead to that of ERIM. As we show below, ERIM's overheads are substantially lower than those of the other techniques. But before presenting these results, we provide a brief *qualitative* comparison of the techniques in terms of their threat models.

Qualitative comparison of techniques Isolation using standard kernel page tables affords a threat model similar to ERIM's. In particular, like ERIM, the OS kernel must be trusted. In principle, isolation using a hypervisor's extended page tables (VMFUNC) can afford a stronger threat model, in which the OS kernel need not be trusted [34].

Isolation using SFI, with or without Intel MPX, affords a threat model weaker than ERIM's since one must additionally trust the transform that adds bounds checks to the untrusted code. For full protection, a control-flow integrity (CFI) mechanism is also needed to prevent circumvention of bounds checks. This further increases both the trusted computing base (TCB) and the overheads. In the experiments below, we omit the CFI defense, thus underestimating SFI overheads for protection comparable to ERIM's.

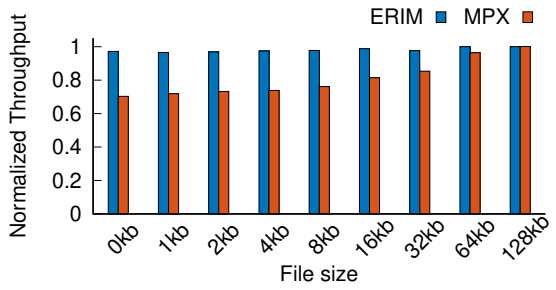
Instrumenting untrusted code for full memory safety, i.e., bounds-checking at the granularity of individual memory allocations, implicitly affords the protection that SFI provides. Additionally, such instrumentation also protects the untrusted code's data from other outside threats, a use case that the other techniques here (including ERIM) do not handle. However, as for SFI, the mechanism used to instrument the untrusted code must be trusted. In our experiments below, we enforce memory safety by compiling untrusted code to WebAssembly, and this compiler must be trusted.

Next, we quantitatively compare the overheads of these techniques to those of ERIM.

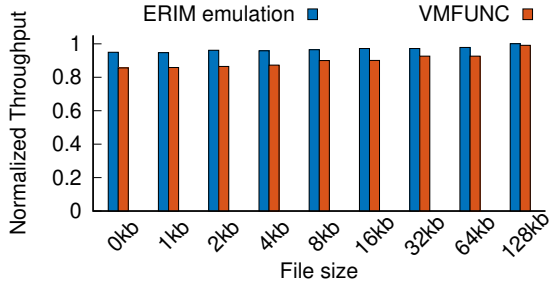
SFI using MPX We start by comparing the cost of ERIM's isolation to that of isolation based on SFI using MPX. For this, we follow the NGINX experiment of Section 6.2. We place OpenSSL (trusted) in a designated memory region, and use MemSentry [30] to compile all of NGINX (untrusted) with MPX-based memory-bounds checks that prevent it from accessing the OpenSSL region directly.[5] To get comparable measurements on the (no protection) baseline and ERIM, we recompile NGINX with Clang version 3.8, which is the version that MemSentry supports. We then re-run the single worker experiments of Section 6.2.

Figure 3a shows the overheads of MPX and ERIM on NGINX's throughput, relative to a no-protection baseline. The MPX-based instrumentation reduces the throughput of NGINX by 15-30% until the experiment is no longer CPU-

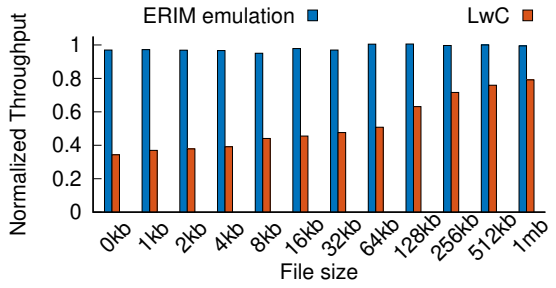
[5] This setup *reduces* the overheads of MPX as compared to the setup of Section 6.2, which isolates only small parts of OpenSSL. It is also less secure. Hence, the MPX overheads reported here are conservative.



(a) ERIM vs. SFI using MPX (averages of 3 runs, std. devs. below 1.9%)



(b) Emulated ERIM vs. VMFUNC (averages of 3 runs, std. devs. below 0.9%)



(c) Emulated ERIM vs. LwC (averages of 5 runs, std. devs. below 1.1%)

Figure 3: Comparison of NGINX throughput with ERIM and alternative isolation techniques

bound (file sizes $\geq$ 64kb). In contrast, ERIM reduces overheads by no more than 3.5%. Across all file sizes, MPX overheads are 4.2-8.5x those of ERIM.

MPX (more generally, SFI) and ERIM impose overhead in different ways. MPX imposes an overhead during the execution of NGINX (the untrusted component), while ERIM imposes an overhead on component switches. Consequently, one could argue that, as the switch rate increases, ERIM *must* eventually become more expensive than MPX. While this is theoretically true, in this experiment, we already observe extremely high switch rates of 1.2M/s (for file size 0kb) and, even then, MPX's overhead is 8.4x that of ERIM's overhead.

Further, as explained earlier, for strong security, SFI must be supported by control-flow integrity, which would induce additional overheads that are not included here.

Extended page tables (VMFUNC) Next, we compare ERIM to isolation based on extended page tables (EPTs) using Intel VT-x and VMFUNC. To get access to EPTs, we use Dune [9] and a patch from MemSentry. We create two page tables—one maps the trusted region that contains session keys, and the other maps the untrusted region that contains all the remaining state of NGINX and OpenSSL. Access to the first table is efficiently switched on or off using the VMFUNC EPT switch call provided by the MemSentry patch. This call is faster than an OS process switch since it does not switch the process context or registers. Since we use Dune, the OS kernel runs in hypervisor mode. It has the switch overheads of hypervisor-based isolation using VMFUNC but includes the OS kernel in the TCB.

Unfortunately, MemSentry's patch works only on old Linux kernels which do *not* have the page table support needed for MPKs and, hence, cannot support ERIM. Consequently, for this comparison, we rely on an emulation of ERIM's switch overhead using standard x86 instructions. This emulation is described later in this section, and we validate that it is accurate to within 2% of ERIM's actual overheads on a variety of programs. So we believe that the comparative results presented here are quite accurate.

Figure 3b shows the throughput of NGINX protected with VMFUNC and emulated ERIM, relative to a baseline with no protection for different file sizes (we use Linux kernel v3.16). Briefly, VMFUNC induces an overhead of 7-15%, while the corresponding overhead of emulated ERIM is 2.1-5.3%. Because both VMFUNC and ERIM incur overhead on switches, overheads of both reduce as the switching rate reduces, which happens as the file size increases. (The use of Dune and extended page tables also induces an overhead on all syscalls and page walks in the VMFUNC isolation.)

To directly compare VMFUNC's overheads to *actual* ERIM's, we calculated VMFUNC's overhead as a function of switch rate. Across different file sizes, this varies from 1.4%-1.87% for 100,000 switches/s. In contrast, actual ERIM's overhead in the similar experiment of Section 6.2 never exceeds 0.44% for 100,000 switches/s. This difference is consistent with the microbenchmark results in Table 2.

Kernel page tables (lwCs) Next, we compare ERIM's overhead to that of lwCs [33], a recent system for in-process isolation based on kernel page-table protections. LwCs map each isolated component to a separate address space in the same process. A switch between components requires kernel mediation to change page tables, but does not require a process context switch. To measure lwC overheads, we rerun the NGINX experiment of Section 6.2, using two lwC contexts, one for the session keys and encryption/decryption functions and the other for NGINX and the rest of OpenSSL. Unfortunately, lwCs were prototyped in FreeBSD, which does not support MPK, so we again use our emulation of

ERIM's switch overhead to compare. All experiments reported here were run on Dell OptiPlex 7040 machines with 4-core Intel Skylake i5-6500 CPUs clocked at 3.2 GHz, 16 GB memory, 10 Gbps Ethernet cards, and FreeBSD 11.

Figure 3c shows the throughput of NGINX running with lwCs and emulated ERIM, relative to a baseline without any protection. With lwCs, the throughput is never above 80% of the baseline, and for small files, where the switch rate is high, the throughput is below 50%. In contrast, the throughput with emulated ERIM is within 95% of the baseline for all file sizes. In terms of switch rates, lwCs incur a cost of 10.5-18.3% for 100,000 switches/s across different file sizes. *Actual* ERIM's switch overhead during the similar experiment of Section 6.2 is no more than 0.44% across all file sizes, which is two orders of magnitude lower than that of lwCs.

Memory safety (WebAssembly) Finally, we compare ERIM's overheads to those of full memory safety on untrusted code. Specifically, we compare to compilation of untrusted code through WebAssembly [21], a memory-safe, low-level language that is now supported natively by all major web browsers and expected to replace existing SFI techniques like Native Client in the Chrome web browser. We compare to ERIM using the experiment of Section 6.3. We re-compile the (untrusted) SQLite library to WebAssembly via emscripten v1.37.37's WebAssembly backend [5], and run the WebAssembly within Node.js, which supports the language. Accross tests of Table 5, the overhead of using WebAssembly varies from 81% to 193%, which is one to two orders of magnitude higher than ERIM's overhead.

Emulating ERIM's switch cost We describe how we emulate ERIM's switch cost when comparing to VMFUNC and lwCs above. Specifically, we need to emulate the cost of a WRPKRU instruction, which isn't natively supported in the environments of those experiments. We do this using xor instructions to consume the appropriate number of CPU cycles, followed by RDTSCP, which causes a pipeline stall and prevents instruction re-ordering. Specifically, we execute a loop five times, with `xor eax,ecx; xor ecx,eax; xor eax,ecx`, followed by a single RDTSCP after the loop.

To validate the emulation we re-ran the SPEC CPU 2006 benchmark with CPI/CPS (Section 6.4) after swapping actual WRPKRU instructions with the emulation sequence shown above and compared the resulting overheads. In each *individual* test, the difference in overhead between actual ERIM and the emulation is below 2%. We note that a perfectly precise emulation is impossible since emulation cannot exactly reproduce the effects of WRPKRU on the execution pipeline. (WRPKRU must prevent the reordering of loads and stores with respect to itself.) Depending on the specific benchmark, our emulation slightly over- or underestimates the actual performance impact of WRPKRU. We also observed that emulations of WRPKRU using LFENCE or MFENCE (the latter was suggested by [30]) in place of RDTSCP incur too little or too much overhead, respectively.

7 Conclusion

Relying on the recent Intel MPK ISA extension and simple binary inspection, ERIM provides hardware-enforced isolation with an overhead of less than 1% for every 100,000 switches/s between components on current Intel CPUs, and almost no overhead on execution within a component. ERIM's switch cost is up to two orders of magnitude lower than that of kernel page-table based isolation, and up to 3-5x lower than that of VMFUNC-based isolation. For VMFUNC, virtualization can cause additional overhead on syscalls and page table walks. ERIM's overall overhead is lower than that of isolation based on memory-bounds checks (with Intel MPX), even at switch rates of the order of 10^6/s. Additionally, such techniques require control-flow integrity to provide strong security, which has its own overhead. ERIM's comparative advantage prominently stands out on applications that switch very rapidly and spend a non-trivial fraction of time in untrusted code.

Acknowledgements We thank the anonymous reviewers, our shepherd Tom Ritter, Bobby Bhattacharjee, and Mathias Payer for their feedback, which helped improve this paper. This work was supported in part by the European Research Council (ERC Synergy imPACT 610150) and the German Science Foundation (DFG CRC 1223).

References

[1] `https://www.sqlite.org`.

[2] `https://nodejs.org`.

[3] `https://httpd.apache.org/docs/2.4/programs/ab.html`.

[4] `https://www.sqlite.org/testing.html`.

[5] `https://github.com/kripken/emscripten`.

[6] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2005.

[7] ARM Limited. Developer guide: ARM memory domains. `http://infocenter.arm.com/help/`, 2001.

[8] ARM Limited. ARM Security Technology. `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf`, 2009.

[9] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mezières, and Christos Kozyrakis. Dune:

Safe user-level access to privileged CPU features. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[10] Andrea Bittau and Petr Marchenko. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of Networked System Design and Implementation (NSDI)*, 2008.

[11] Nathan Burow, Xinping Zhang, and Mathias Payer. SoK: Shining Light On Shadow Stacks. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2019.

[12] Scott A. Carr and Mathias Payer. Datashield: Configurable data confidentiality and integrity. In *Proceedings of ACM ASIA Conference on Computer and Communications Security (AsiaCCS)*, 2017.

[13] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-Grained Execution Units with Private Memory. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2016.

[14] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[15] Dyninst. Dyninst: An application program interface (API) for runtime code generation. `http://www.dyninst.org`.

[16] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the point(er): On the effectiveness of code pointer integrity. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[17] Tommaso Frassetto, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. JITGuard: Hardening just-in-time compilers with SGX. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[18] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. IMIX: In-process memory isolation extension. In *Proceedings of USENIX Security Symposium*, 2018.

[19] Enes Göktas, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. Undermining Information Hiding (and What to Do about It). In *Proceedings of USENIX Security Symposium*, 2016.

[20] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[21] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[22] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2019.

[23] Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. librando: Transparent Code Randomization for Just-in-Time Compilers. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013.

[24] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. Enforcing least privilege memory views for multithreaded applications. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[25] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2013.

[26] Intel Corporation. Memory Protection Extensions (Intel MPX). `https://software.intel.com/en-us/isa-extensions/intel-mpx`.

[27] Intel Corporation. Software Guard Extensions Programming Reference. `https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf`, 2014.

[28] Intel Corporation. Intel(R) 64 and IA-32 Architectures Software Developer's Manual, 2016. `https://software.intel.com/en-us/articles/intel-sdm`.

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

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

Proceedings of ACM European Conference on Computer Systems (EuroSys), 2017.

[31] Volodymyr Kuznetsov, László Szekeres, and Mathias Payer. Code-pointer integrity. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[32] Hojoon Lee, Chihyun Song, and Brent Byunghoon Kang. Lord of the x86 rings: A portable user mode privilege separation architecture on x86. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[33] 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 *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[34] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[35] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[36] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient TCB reduction and attestation. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[37] MITRE. CVE-2014-0160. `https://nvd.nist.gov/vuln/detail/CVE-2014-0160`, 2014.

[38] Lucian Mogosanu, Ashay Rane, and Nathan Dautenhahn. MicroStache: A Lightweight Execution Context for In-Process Safe Region Isolation. In *Proceedings of International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2018.

[39] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. Poking Holes in Information Hiding. In *Proceedings of USENIX Security Symposium*, 2016.

[40] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Analysis of Computing Systems (ACM Sigmetrics)*, 2018.

[41] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2019.

[42] 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 *Proceedings of USENIX Security Symposium*, 2010.

[43] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2004.

[44] Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-VM monitoring using hardware virtualization. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2009.

[45] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, Haibing Guan, and Jinming Li. Deconstructing Xen. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2017.

[46] The Clang Team. Clang 5 documentation: Safestack. `http://clang.llvm.org/docs/SafeStack.html`, 2017.

[47] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 1993.

[48] Wikimedia Foundation. Static HTML dump. `http://dumps.wikimedia.org/`, 2008.

[49] Wikimedia Foundation. Page view statistics April 2012. `http://dumps.wikimedia.org/other/pagecounts-raw/2012/2012-04/`, 2012.

[50] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proceedings of USENIX Security Symposium*, 2002.