No Need to Hide: Protecting Safe Regions on Commodity Hardware

Koen Koning
Vrije Universiteit Amsterdam
koen.koning@vu.nl

Xi Chen
Vrije Universiteit Amsterdam
x.chen@vu.nl

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@cs.vu.nl

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

Elias Athanasopoulos
University of Cyprus
eliasathan@cs.ucy.ac.cy

Abstract

As modern 64-bit x86 processors no longer support the segmentation capabilities of their 32-bit predecessors, most research projects assume that strong in-process memory isolation is no longer an affordable option. Instead of strong, deterministic isolation, new defense systems therefore rely on the probabilistic pseudo-isolation provided by randomization to "hide" sensitive (or safe) regions. However, recent attacks have shown that such protection is insufficient; attackers can leak these safe regions in a variety of ways.

In this paper, we revisit isolation for x86-64 and argue that hardware features enabling efficient deterministic isolation *do exist*. We first present a comprehensive study on commodity hardware features that can be repurposed to isolate safe regions in the same address space (e.g., Intel MPX and MPK). We then introduce MemSentry, a framework to harden modern defense systems with commodity hardware features instead of information hiding. Our results show that some hardware features are more effective than others in hardening such defenses in each scenario and that features originally conceived for other purposes (e.g., Intel MPX for bounds checking) are surprisingly efficient at isolating safe regions compared to their software equivalent (i.e., SFI).

CCS Concepts • **Security and privacy** → **Systems security**; **Software and application security**

Keywords hardware features, isolation, information hiding, software fault isolation

EuroSys '17, April 23 - 26, 2017, Belgrade, Serbia

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4938-3/17/04...$15.00

DOI: http://dx.doi.org/10.1145/3064176.3064217

1. Introduction

Operating systems generally provide strong isolation between different processes, but they do not provide any isolation between components *inside* a process. This lack of intra-process isolation allows for fatal memory disclosures in C/C++ binaries, which rank among the most serious vulnerabilities today and have a key role in software exploitation. These memory disclosures allow attackers to reveal the code layout [58] and thus bypass fine-grained diversification [54, 66], leak sensitive data such as cryptographic keys [24], but, most importantly, attackers can bypass state-of-the-art security defenses [40, 42] by deliberately revealing safe regions that host sensitive data of the vulnerable program [26, 29, 32, 52]. In this paper, we advocate that no matter the defense in place, if deterministic isolation is not guaranteed, then the protection can be bypassed. To achieve such guarantees, we consider efficient isolation techniques based on commodity hardware features. We also introduce MemSentry, a framework that implements these techniques and allows for a comprehensive and practical comparison. Finally, we show that, other than for evaluation purposes, MemSentry can be effectively used to harden modern defense systems that are based on information hiding, including those now deployed in production such as SafeStack [40, 61].

Modern security defenses rely on the confidentiality of their metadata for managing their state. No matter how advanced the protection, any leak of the sensitive metadata is sufficient to subvert it completely. For example, code pointer integrity (CPI) [40] stores all sensitive pointers and related metadata in a so-called safe region which, on 64-bit systems, is hidden at a random location in a very large address space. As a result, the protection of the safe region hinges on the entropy of address space layout randomization (ASLR). Unfortunately, there are *several* different strategies an attacker can follow to bypass ASLR. No matter the size of the virtual address space, attackers can infer the location of the hidden object by leveraging allocation oracles [52], thread spraying [32], crash-resistant primitives [29], or other side channels [8, 26, 33]. All the aforementioned strategies are strong evidence that the probabilistic isolation offered by information hiding should no longer be considered an option as a replacement for deterministic isolation. Instead, hardware features should be revisited for delivering deterministic and efficient memory isolation of safe regions. Where 32-bit x86 architectures provide segmentation to efficiently isolate sensitive data, there is no such answer on modern x86-64 processors. While alternatives to probabilistic isolation exist, they are typically tailor-made for a particular setting. In this paper we instead propose a general design for memory isolation, allowing any deterministic isolation technique to be applied to the safe regions of any defense. One example is software fault isolation (SFI), which sandboxes code with instrumentation, for instance to ensure attackers cannot access memory outside of a predefined region [70]. While commonly applied to sandbox native C/C++ code in the browser, some defenses also adopted it to prevent attackers from bypassing the control-flow integrity (CFI) instrumentation [50]. Recent work on optimizing SFI also supports the importance of efficient isolation in modern defenses, but such work often focusses on a very specific domain. For example, LR^2 [9] reduces SFI checks to protect diversified code only.

The traditional POSIX way of achieving intra-process isolation is by using the `mprotect` system call, which can instruct the kernel to add or remove permissions of memory pages on the fly. Unfortunately, using this strategy to protect safe regions results in significant overhead (e.g., 20–50x in our experiments). We therefore look at hardware features of modern 64-bit Intel processors, including memory protection extensions (MPX), encryption instructions (AES-NI), memory protection keys (MPK), software guard extensions (SGX), and virtualization features such as the new VMFUNC instruction. While researchers have used features like AES-NI [41] or leveraged special x86-64 constructs [23], no prior work has evaluated the pros and cons of the different hardware features for isolating safe regions, nor provided a general design. While we focus on the x86-64 architecture, our approach is also applicable to other architectures when similar hardware features are available (e.g., ARM [59]).

In addition to our analysis of isolation techniques, we also present MemSentry, a comprehensive and self-contained deterministic memory isolation framework. MemSentry serves as a testbed for these techniques, allowing existing and future techniques to be implemented and deployed easily. Not only does it provide a platform for benchmarking and comparing these hardware features, but also to harden modern security defenses that are based on information hiding. To demonstrate its applicability, we evaluate MemSentry against SPEC 2006 and in a variety of scenarios, such as shadow stacks, control-flow integrity, and code diversification. Our evaluation reveals a number of interesting findings, which are currently not well established in the community. We show that MPX—a recently added feature in Intel CPUs for checking pointer bounds—can be repurposed as a replacement to SFI for shadow stacks and other control-flow defenses with acceptable overhead (up to 7.5% vs 21.6% for SFI). Our approach is to rely on a new design where only a single bound check is required, making MPX much more efficient than its normal use case of double bounds checking—rapidly dismissed by practitioners for its exorbitant overhead [53]. On the other hand, we show that for sparse instrumentations such as heap protection, isolation techniques based on features such as VMFUNC are more efficient (5.5%).

To summarize, our contributions are:

- An analysis of recent security defenses, demonstrating their vulnerability against information leakage attacks and stressing the fundamental cause of their weakness: *lack of deterministic isolation for safe regions*,
- A survey of a number of recent (Intel) hardware features and a general design that allows for such features to be used for memory isolation.
- MemSentry, the first comprehensive and self-contained deterministic memory isolation framework, which allows users to benchmark these hardware features on any platform and to strengthen existing defenses for additional protection.
- An evaluation of MemSentry and the various hardware features in different scenarios, ranging from control-flow protection to sensitive user data protection. Depending on the underlying defense, we can protect safe regions with a 1.1% overhead in the best case and with a 2.8% (integrity) or 14.7% (integrity and confidentiality) overhead in the worst case. Our results also provide guidance on how to use different hardware-supported techniques on commodity hardware.

2. Memory isolation in review

In this section we give a short introduction of deterministic and probabilistic isolation. Later on, we review some representative defenses that rely on strong isolation for countering exploitation. Finally we discuss the conditions under which probabilistic isolation fails.

2.1 Deterministic vs probabilistic isolation

Modern operating systems allow processes to run isolated from each other. Each process has its own virtual address space, and cannot normally interfere with other processes. However, many systems rely on further isolation inside the running process: certain parts of the process (*safe regions*) should be protected from the rest of it.

Isolating certain parts within a running process is commonly referred to as SFI [65]. Depending on the isolation required, implementations may vary, but the basic concept is simple. The isolated part of the process must be compiled masking read and/or write operations, so that all are confined to a given memory range. Any access outside this range, which defines the boundaries of the isolation, is not permitted.

Instrumenting all these operations can be expensive. In 32-bit x86 architectures, where segmentation is available, isolated parts can be placed in different segments [28, 65, 70]. For example, a defense could create a second segment with a higher limit, and place its sensitive data in the high part. Referencing this isolated part can be done only by correctly setting the segment registers, and not by simply overwriting accessible memory. Enforcing isolation by means of SFI, using full segmentation or by instrumenting read/write operations, is what we refer to as *deterministic isolation*. Deterministic isolation guarantees that the isolated part cannot access anything outside the isolation border. An alternative to deterministic isolation is *probabilistic isolation*, which we can realize using *information hiding*. Rather than truly isolating them, information hiding *hides* the sensitive parts of a running process in a large (64-bit) virtual address space, while removing all references to them from the rest of the process.

2.2 Defenses that rely on isolation

Many defense systems for countering software exploitation require a part in memory isolated from the vulnerable process itself. If this part is compromised the defense is rendered useless. Since deterministic isolation is considered expensive in 64-bit systems—for instance because of the lack of segmentation—all of the following systems are realized using information hiding. For better presentation, we have divided the systems in four distinct categories:

Code diversification Code reuse attacks rely on creating a sequence of *gadgets*: small pieces of the original code that an attacker jumps over to create the desired code. For countering such attacks, code diversification destroys any prior knowledge of code locations, so that locating gadgets is no longer deterministic. Diversification can be realized via function/basic-block permutation [66], fine-grained ASLR [30], opcode permutations [54], or instruction-layout randomization [35]. Unfortunately, memory-disclosure vulnerabilities render all these mechanisms ineffective [58]. An additional defense against memory disclosure is run-time re-randomization [15, 30, 67], which first diversifies programs at function or basic-block level at compilation or at load time, and then periodically remaps the code to different addresses.

For example, Isomeron [22] and Oxymoron [3] consistently maintain offsets which indicate the distance between the currently mapped code and newly remapped code. By adding these offsets to all branch targets, they properly divert the control flow to the newly remapped code. However, if an attacker is able to leak the offsets before remapping is done, they can predict the address of the newly remapped code and bypass the protection. To protect these offsets Isomeron uses a shadow stack-like structure of *recorded decisions*, and Oxymoron uses an indirection table called the *Rattle table*. Isolation of these components is crucial. In a similar fashion,

Defense	Vuln.		Isolation		Instrumentation points
	r	w	Prob.	Det.	
CCFIR	√		√		Indirect branches
O-CFI	√		√		Indirect branches
Shadow Stack		√	√		`call/ret`
StackArmor	√	√	√		`call/ret`
TASR	√	√	√		System I/O
Isomeron	√	√	√		Indirect branches
Oxymoron	√		√		Code page across edges
CPI		√	√		Memory accesses
CCFI				√	Memory accesses
ASLR-Guard	√	√	√		Memory accesses
DieHard	√	√	√		`malloc/free`
Readactor				√	Indirect branches
LR^2				√	Mem. accesses & ind. branches

Table 1. Defense systems that are based on memory isolation. Shown are what vulnerabilities they protect against: reads (r) and/or writes (w), what type of isolation they provide (probabilistic or deterministic) and where they insert code.

TASR [7] maintains a list of activated code pointers. When remapping occurs, the system remaps these code pointers to point to the newly mapped code. Again, isolation of the list of code pointers is essential, since the attacker could first leak the list of code pointers and then replace them to bypass the remapping entirely.

Control-flow integrity Unlike code diversification, which hides the code space from attackers, control-flow integrity [1] provides more deterministic protection via explicit checking of indirect-branch targets at runtime. CFI instruments every indirect branch to ensure that it can reach only the intended target set as approximated through static analysis. Depending on the number of targets sets, we can classify CFI as either coarse-grained or fine-grained. Coarse-grained CFI [49, 70, 73] supports no more than two or three (large) target sets, making it vulnerable to attacks that construct malicious payloads out of legitimate targets [10, 21, 31]. Fine-grained CFI [20, 25, 63], on the other hand, supports more target sets but also typically introduces more overhead.

Researchers have combined code randomization with coarse-grained CFI to strengthen it, but doing so also introduces an additional component, which must be protected. For example, CCFIR [71] generates code stubs for indirect branches, which it places randomly in its springboard regions, while O-CFI [48] employs a so-called BLT table. In both cases, isolation of these structures is essential.

Code-pointer separation An attacker can hijack the control flow of a program by corrupting a code pointer. One way to prevent such attacks is to store all sensitive code pointers in a well-isolated region of memory. For example, return addresses stored on the stack are high-value targets, and researchers have proposed shadow stacks to protect them from corruption [19, 25, 40]. By storing all return addresses in the isolated "shadow stack", separated from the regular stack, simple stack smashing attacks are no longer possible.

To overcome the problem of leaking the location of the shadow stack, StackArmor [14] allocates individual stack frames at each function call at random addresses to keep the information leakage attacks from expanding to other stack frames. Alternatively, code-pointer integrity (CPI) [40] introduces a safe region to store both the code pointers and their related metadata, critical to its security.

Another approach is to protect pointers using encryption. Following the idea of PointerGuard [17], which applies a `xor` operation on all pointers with secret keys to prevent buffer overflow attacks, CCFI [44] relies on Intel AES-NI instructions to encrypt/decrypt code pointers. The AES keys are stored in dedicated `xmm` registers. Since CCFI introduces a large overhead (3.5x for SPEC 2006), ASLR-Guard [42] proposes a more lightweight encryption scheme which relies on `xor` instructions. A preallocated key table (AG-RandMap) stores different `xor` keys for each entry, making lookups via code pointers efficient. Moreover, by using different `xor` keys, ASLR-Guard provides stronger encryption compared to PointerGuard, with less overhead. Similar to many previous protections, it is essential to isolate the AG-RandMap not just against information disclosures, but also against writes.

Sensitive non-control data Besides the danger of control data attacks, non-control data can also be security sensitive. Examples include sensitive configuration data, user data, and so on [12]. Such data usually resides on the heap and is therefore vulnerable to information leakage and memory corruption attacks. DieHard [6], and follow-ups [51], address this problem by designing a (probabilistically) safe memory allocator, resilient to heap-based memory corruption.

2.3 Threat model

We assume a defense system (see Section 2.2) protecting a vulnerable process against code reuse. The attacker holds an arbitrary read and write primitive, but code reuse is not effective due to the defense system in place. Assuming the safe region is *hidden* and not *isolated*, the attacker can carry out the attack in two phases. First, a safe region, important for the defense (e.g., the safe region of CPI [40]), must be revealed using one of the many available techniques [26, 29, 32, 36, 52]. Once the attacker knows the safe region, the defense can be bypassed, and, at this second phase, code reuse can be effectively launched. MemSentry essentially stops the attack at the first phase, protecting the defense system in place, and consequently the protected program. Note that at this point, code reuse is still not possible and MemSentry cannot be attacked by exploiting vulnerabilities and chaining ROP gadgets: executing any ROP gadget can be done only once the defense is bypassed. Finally, we assume that the code of MemSentry is trusted and implemented correctly.

3. Deterministic memory isolation

From Section 2, it should be clear that memory isolation is critical for many defenses and that their reliance on information hiding is wholly insufficient. In this section, we outline a design for easily-applicable *deterministic* memory isolation. This design can be applied to many existing defenses and abstracts away the details of the underlying features being used. Thanks to our generic design—instead of the ad-hoc solutions of the past—users can now easily swap out different isolation techniques, for instance depending on availability of such features on commodity processors.

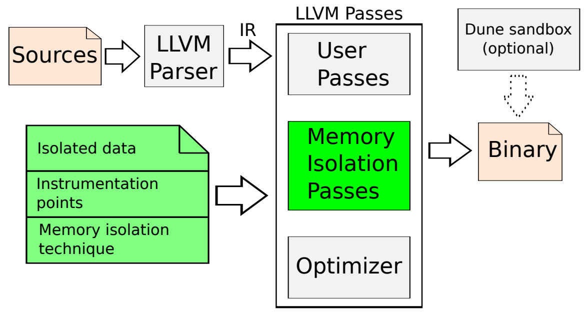

Figure 1. Overview of MemSentry's framework. As an LLVM pass it will transform the IR, depending on the user-provided parameters, resulting in an instrumented binary. The resulting binary can optionally be executed in the Dune sandbox to allow for process-level virtualization (e.g., for VMFUNC).

We categorize deterministic memory isolation solutions in two groups: *address-based* and *domain-based*. In the former, the address space is split into partitions and memory accesses are masked to only access the allowed partition. Domain-based isolation does not explicitly force memory accesses to go to a certain partition. It instead defines areas in the address space that are completely inaccessible and can be toggled on or off, making them only temporarily accessible to any instruction in the program. One can apply either of these memory isolation categories to a defense system given three sources of information: (a) the safe region(s) of sensitive data (*isolated data*), (b) the instructions of the program that are allowed to access sensitive data (*instrumentation points*), and (c) the preferred memory *isolation technique*.

The remainder of this section discusses this design, starting with the three inputs required, followed by an overview of our deterministic isolation techniques based on commodity hardware features. Figure 1 shows an overview of MemSentry, which implements our design using LLVM, allowing it to easily be applied on top of different systems.

Isolated data In general, the sensitive data of a program should be deterministically isolated. Such data may contain cryptographic keys, cookies, access-control metadata, and so on. Additionally, defense systems often contain metadata used internally, which must remain protected. For example, a system may incorporate a safe region containing sensitive code pointers that should not be overwritten or a redirection table which should not be read by other parts of the process.

The effort required to locate the isolated data depends on the particular application. For some defense systems, this is trivial; the memory range occupied by a shadow stack, for example, can be easily determined and isolated. On the other hand, a memory allocator may store sensitive metadata in several places. In that case, modifying the allocator requires additional effort. Systems that are already based on *probabilistic isolation* can always be enhanced to use deterministic isolation, since they already contain a well-defined part that needs to be hidden from the rest of the process.

Instrumentation points We refer to the instructions or code regions that need access to sensitive information as *instrumentation points*, as every access has to be instrumented to be permitted. Depending on the application, locating and annotating the code that operates on sensitive data can be trivial or more challenging. For instance, for a typical shadow stack, only the `call` and `ret` instructions are allowed to read and write the sensitive data—in this case an isolated stack containing the return addresses. As another example, consider secure heap allocators such as DieHard [6]. Here the instrumentation points are all calls to the allocator, such as *malloc* and *free*. In both of these cases, locating and annotating the instrumentation points is trivial. However, more problematic cases exist. For example, CPI [40] instruments all read and write operations to code pointers. Inferring all these operations requires points-to analysis and is subject to issues with the accuracy of the analysis. Even in this case, however, determining the set of instrumentation points is already a requirement for the defense itself.

Isolation techniques Finally, one of the many isolation techniques must be chosen to guarantee full safe region isolation. This can be realized using different hardware features available on x86-64. The best choice for the isolation technique depends on the nature of the application. In the following sections, we first detail each of the supported isolation techniques and then discuss the trade-offs among different selections (Section 6.3).

Usage From the perspective of a defense developer, MemSentry is easy to use. The developer includes the MemSentry pass to run after the defense pass at compilation time, and adds a static library during linking. The developer then allocates the safe regions using `saferegion_alloc(sz)`, which is part of the static library. For the general case where defense passes insert calls to functions at certain points, these functions should be annotated so they can access the safe region. For the common case where these are contained in a static library, we have included a pass to automatically create these annotations. For more general instrumentation, defense passes can use the function `saferegion_access(ins)` for every instruction that needs access to the safe region. Such an instruction can be a memory access, or for example a call to an intrinsic. The MemSentry pass will run afterward and use the annotations, implemented as LLVM metadata, to insert the correct isolation.

3.1 Domain-based isolation

Domain-based solutions split the process operation into multiple *domains*, where a domain can either be *active* (accessible) or *inactive* (triggering faults when accessing data specific to this domain). Domains can contain multiple (smaller) portions of the address space, and are activated using special instructions (which can thus not be triggered by an attacker only equipped with a read/write primitive). For simplicity, we assume a model of two domains: the (default), *nonsensitive domain*, and the *sensitive domain* containing only the sensitive data. Whereas the former is always active, the latter is only enabled when the program accesses isolated data. This model makes it easier to reason about and compare different implementations, but can be extended into multiple and/or disjoint domains, depending on the technique.

For 32-bit x86, segmentation would fall in this category. Sadly, this does not work in x86-64 as segmentation has effectively been removed. Although the `fs` and `gs` selectors still exist, they must point to a location in the accessible address space. We also do not discuss isolation based on traditional paging (optionally sped up using the PCID feature) as this would require intrusive changes to the kernel itself; we aim for a widely deployable and easy to use framework.

Note that, as discussed in Section 2.3, we assume the memory isolation works in conjunction with existing defense solutions. Therefore, when protecting such systems we do not require additional domain-switching logic. However, if arbitrary program data should be protected, more complex domain-switching is required [41, 57].

EPT switching via VM functions ("VMFUNC") Intel CPUs include hardware-accelerated virtualization support, called VT-x (or VMX). With hardware virtualization, the host can run several guests efficiently, without emulating them. Two features that can speed up switching between VMs are the virtual-processor identifier (VPID) and extended page tables (EPT). The former adds tagging information for the TLB, whereas the latter adds a second paging structure. This EPT maps guest-physical to host-physical addresses, that is, physical addresses obtained in the guest via normal page tables are then looked up in the EPT to determine the real physical address. Normally, switching page tables (both normal and extended ones) is a costly operation that only the kernel and hypervisor (respectively) can invoke, making swapping pages in and out impractical. However, with the introduction of VM functions, the guest can invoke several specialized functions (pre-programmed in the CPU) which previously required intervention of the hypervisor. The only VM function currently present is *EPT pointer switching*, which allows the guest to efficiently switch its entire EPT. The hypervisor sets up a list of EPTs, and the guest can then switch between these on its own, selecting the active EPT.

To use these features for memory isolation, we maintain two EPTs, both containing all normal (i.e., non-sensitive) page mappings. However, the mappings for the sensitive data are only present in the second EPT, which is only active when the guest requires access to this data. Thus, we can achieve memory isolation by switching between the normal and sensitive EPTs, representing the *nonsensitive* and *sensitive domains* respectively. By inserting `vmfunc` calls around the instrumentation points, the pages mapped in the secure EPT can be used only by authorized instructions, and thus forms the *sensitive domain*.

Unlike prior `vmfunc`-based solutions, this solution can also function in a self-contained and easily deployable manner by using process-level virtualization [5], as is used in MemSentry. Here, the hypervisor only manages a small VM with the corresponding process running in it, which avoids the overhead and complexity of running the entire operating system and program in a VM. This is also less intrusive than modifying the hypervisor. This aspect is, however, not fundamental to our design; one could also modify an existing hypervisor (such as KVM) to support this.

MPK Memory protection keys will be a feature available in future Intel processors.[1] With MPK, every page belongs to one of 16 domains, determined by 4 bits in every page-table entry (referred to as the *protection key*). For every domain, there are two bits in a special register, named `pkru`. These bits denote whether pages associated with that key can be read or written. While only the kernel can change the key of a page, reading and writing the `pkru` register is possible from user-space using the `rdpkru` and `wrpkru` instructions respectively.

Isolation can be enabled using MPK by placing the sensitive data in pages that have a particular protection key, forming the *sensitive domain*. An appropriate instrumentation enables reads and/or writes to the data by setting the *access-disable* and *write-disable* bits, respectively, using `wrpkru`. As long as these bits are unset, the *sensitive domain* is accessible. By setting the bits back, the sensitive domain is disabled, making only the *nonsensitive domain* available.

AES-NI encryption ("crypt") Where the previously discussed domain-based isolation techniques rely on unmapping the sensitive data when it should not be accessed, an alternative is to instead *encrypt* it in-place until it is needed. In order to speed up AES encryption, Intel has introduced dedicated instructions, called AES-NI. This instruction set accelerates the basic blocks of AES: there are functions for performing a single round of encryption and for generating the round keys.

SGX The recently introduced Intel SGX extension can also provide domain-based isolation, even though it is intended for more far-reaching isolation. SGX allows applications to create *enclaves*, separated compartments of code and data. The processor automatically encrypts these enclave compartments, preventing the application itself and even the operating system and hypervisor from reading the enclave's data. This allows for trusted execution in potentially hostile environments, such as cloud infrastructure; clients can upload an encrypted program to a cloud provider, and be guaranteed that it ran correctly. When enclaves are created, the OS sets up the binary blob that should be loaded and the mappings of the enclave, after which it finalizes the enclave. Once an enclave is finalized the application can perform calls into the code of the enclave (ECALLs) via pre-defined entry points. Similarly, the enclave can perform calls outside to the enclave (OCALL), for instance to perform I/O. It is important to note that currently the mappings of the enclave are fixed: no new memory can be allocated to the enclave.

We can apply our domain-based isolation design using SGX by creating an enclave containing the sensitive data and all the code required to access/modify that data. We can thus switch domains by switching execution to the enclave via an ECALL, assuming all code that touches sensitive information can be extracted and combined into the enclave.

However, this approach is currently markedly inferior to the other isolation solutions which we present in this paper. First and foremost, the overhead is much larger than other solutions, as shown in Table 4. Decomposing the application into the domains (the untrusted code and the enclave) is also far less practical: the actual code touching sensitive data must be present in the enclave, whereas with other solutions we simply trigger the domain switch. All sensitive data must be allocated when the enclave is created, and is limited in size, adding more complications in many cases. Finally, deploying this on-the-fly would be more challenging as an Intel-issued signing key has to be used on the binary for the enclave, and SGX is not yet widely supported by hardware. Processor-level compartmentalization is definitely a promising development, but SGX itself is unsuitable for efficient memory isolation.

3.2 Address-based isolation

Address-based solutions split the address space into two or more partitions, instrumenting the program to allow only certain instructions to access a particular partition. This means that every load and store instruction is instrumented with the partition(s) it is allowed to access. For simplicity, we again assume two partitions (the *sensitive partition* and the *nonsensitive partition*), similar to domain-based isolation.

The difference between *address-based* and *domain-based* isolation is that for address-based techniques all pointers are forced into a partition all the time, whereas for domain-based techniques a domain is activated, giving all loads and stores access to it. Additionally, partitions split the address space at a certain point (e.g., everything above 64 TB is sensitive) whereas domains are more flexible.

Traditional SFI ("SFI") One straightforward way of achieving address-based isolation is by using the classic

[1] At the moment of writing, Intel has published a full specification [37] on MPK but has not yet announced a processor supporting it.

Isolation	Instrumentation points	Application
Address-based	Loads	Code randomization
	Loads	CFI variants
	Stores	ShadowStack
	Stores	CPI
	Both + points-to info	Program data
Domain-based	`call` + `ret`	ShadowStack
	Indirect branches	CFI variants
	Indirect branches	Layout randomization
	System calls	Layout randomization
	Allocator calls	Heap
	Points-to info	Program data

Table 2. MemSentry applies to a range of different defense systems. Some have different variants (e.g., CFI) and the specifics may differ. Nevertheless, the framework is generic and covers various types of applications through different isolation types.

SFI. This can be done at the bit level using software-only instrumentation, requiring no additional hardware support. For example, by storing all sensitive data in the upper half of the address space (forming the *sensitive partition*), the higher bits in a pointer are only set when accessing this sensitive data. By masking the instruction before every non-allowed load and store (using a simple `and` instruction), we can deterministically ensure such instructions cannot access sensitive data. While this approach is widely deployable, it may not be the most efficient solution in many applications of interest (as shown in our evaluation in Section 6). Another concern is that traditional SFI cannot deterministically detect invalid memory accesses, but only prevent them, as the masked pointer might still be a valid (different) pointer in the *nonsensitive partition*.

MPX Intel introduced a new hardware feature that allows for efficient bounds checking, called MPX, in the Skylake CPU series. With MPX, the programmer can create and enforce *bounds*, specified by two 64-bit addresses specifying the beginning and the end of a range. Furthermore, new instructions are introduced to efficiently compare a given value against the bounds, raising an exception when the value does not fall within the permitted range. For efficiency, four bounds can be stored into dedicated registers (`bnd0` to `bnd3`). When more bounds are required, they are stored in memory, and the bound registers serve as a caching mechanism. It should be noted that while the bounds-checking itself is very efficient, the usage of many bounds is not. For instance, GCC's implementation of MPX buffer checking frequently spills bounds registers to memory.

For isolation, the address space is partitioned using MPX bounds. By defining a single bound and adding a single bounds check before every memory access, we effectively verify that every pointer used by the program is in the correct partition. By not adding such checks to instructions which are allowed to touch sensitive data, we can enforce that instructions only access memory in the *nonsensitive partition*.

Isolation technique	Maximum domains	Granularity
SFI	48	–[a]
MPX	4[b]	byte
MPK	16	page
VMFUNC	512	page
AES	Infinite	128 bytes

[a] Depends on least significant bit of mask.
[b] Infinite when also using memory besides bound registers.

Table 3. Limitations of memory isolation techniques.

4. MemSentry applications

In this section we discuss how our design, and in particular MemSentry, offers deterministic isolation to existing systems, such as the ones presented in Section 2, as well as other concerns about the applicability of the framework.

Table 2 summarizes the type of isolation and instrumentation points required for some representative systems. For address-based solutions, instrumented memory accesses cannot operate on protected data, while non-instrumented ones can freely read or write to the process address space. For domain-based solutions, the reverse approach is followed. Instrumentation is inserted for accessing sensitive data. For most defense systems, sensitive data is only accessed by new instructions (part of the instrumentation). It is therefore clear which instructions are allowed to touch sensitive data, and no further instrumentation by MemSentry is required. For in-program data such as private keys this is not the case, and MemSentry has to rely on points-to information.

For instance, in a heap protection system such as DieHard, the metadata is only used by the allocator. Therefore, other parts of the program and libraries should not be able to access it. Similarly, access to a shadow stack occurs during `call` and `ret` instructions. Systems based on a shadow stack instrument these instructions therefore it is trivial for MemSentry to determine where domain switches should take place or what memory instructions should not be instrumented for address-based approaches. In this case, only memory stores have to be protected, since the defense is based on the integrity and not the confidentiality of the shadow stack. On the other hand, when protecting private keys, confidentiality is important. Therefore, both reads and writes should be prevented from accessing the cryptographic keys.

Another concern when employing memory isolation is separation of sensitive data from the rest of the program. For many defense systems, separation is enforced by construction. As an example, the code space (a sensitive region in many solutions) is already separated from normal program data in a traditional address space organization. However, arbitrary program data may need protection. In some cases sensitive data is stored in a global data structure, or embedded in another (non-sensitive) one. Table 3 shows the theoretical minimum granularity supported by each technique implemented in MemSentry for storing sensitive data. For example, the minimum size of isolated data when using VMFUNC is one memory page. Separation of sensitive and non-sensitive data is carried out by the defense system itself, and MemSentry is employed only for enforcing the desired isolation technique.

As an example, we could apply MemSentry to SafeStack [40, 61], a shadow stack implementation used in production, with minimal effort. This only requires instrumenting all memory writes, and ensuring the normal (safe) stack was located separately in the address space.

5. Implementation

In this section we describe some of the details of using the different hardware features in practice, and present our solutions for MemSentry.

5.1 VMFUNC

In order to minimize the impact of deploying EPT switching, we deploy this technique (and thus the VM and hypervisor) per process instead of system-wide. To do this, we use a modified version of Dune [5], which allows a single process to run in a VM by running a stripped down version of the KVM hypervisor per process, and a small library-OS to manage the state of the VM. This requires only the (relatively small) Dune kernel module to be loaded, and the remainder of the system does not experience any performance impact.

For MemSentry, we modified Dune to maintain multiple copies of the EPT, which it normally fills in an on-demand basis when an EPT fault occurs. We added a hypercall so that the hypervisor can mark certain mappings as private to only the active EPT, allowing for secret pages to be isolated to a single EPT. The program itself is instrumented to make hypercalls to inform the hypervisor about these secret mappings, and further inserts `vmfunc` calls to switch domains where necessary. The `rax` and `rcx` registers are used to specify the VM function to invoke and which EPTP index to use (respectively). By using the *sandbox* application that is part of Dune, the program runs transparently in the guest environment, with the sandbox taking care of the kernel-level tasks in the VM (such as interrupts and page table management). This technique inherently requires access to certain hardware features such as EPT, VPID and VMFUNC, which might not be implemented in virtual environments (i.e., nested virtualization support is required). However, this is not fundamental to using VMFUNC for isolation, as one could also implement the EPT management in KVM itself.

5.2 MPK

As MPK is not yet available in any CPU at the time of writing, we have implemented an approximation that, while not offering any protection, gives performance results and allows for comparison with other techniques.

Normally, an application reads the `pkru` register using `rdpkru`, (un)sets some bits in the result and writes it back using `wrpkru`. Afterwards, memory accesses should use the new permissions as set in `pkru`. To simulate this, we copy an `xmm` register into a general purpose register, set bits in the result, and move it back to the `xmm` register. This approximates the cost of reading and writing `pkru`, as both the `xmm` registers and `pkru` are special registers. In particular, `xmm` registers are the slowest available registers, and so our implementation reasonably approximates the performance overhead of accessing `pkru`. Additionally, we perform an `mfence` instruction to simulate the cost of changing permissions and the probable serialization caused by writing to a control register. MPK itself needs to perform this as well, because memory accesses cannot be reordered around the `wrpkru` calls.

Usage of MPK clobbers `rax`, `rcx` and `rdx` (simulated using inline assembly), possibly causing compiler register spilling. Since MemSentry runs as a normal LLVM pass, variables are not yet mapped to registers, as this register allocation happens at a later stage. By marking registers as clobbered, LLVM optimizes register usage to minimize spilling to memory. However, both `rcx` and `rdx` are used as function parameters, and are thus often expensive registers to clobber.

5.3 Encryption

MemSentry implements encryption using Intel AES-NI with 128-bit keys. This requires 11 rounds for both encryption and decryption, and by extension 11 different round-keys (with one being equal to the overall key). Ideally, none of the keys are ever spilled into memory, as an attacker could potentially *sniff* memory and break the encryption.[2] Furthermore, storing keys in memory is suboptimal performance-wise, as additional memory accesses are introduced.

While pinning certain registers for storing these keys is a solution, as used by CCFI [44], we deemed this approach impractical. First of all, it requires recompilation of all system libraries to mark these registers as reserved, potentially affecting system-wide performance. Secondly, MemSentry requires distinct keys for both encryption and decryption (an encryption round-key can be used to calculate the decryption round-key using `aesimc`), and there are not enough `xmm` registers available to hold these keys. Storing only the primary key (which can be used to generate the round-keys using `aeskeygenassist`) requires fewer `xmm` registers but employs costly keygen instructions for every domain switch.

MemSentry stores the keys in the upper part of the `ymm` registers, which are not used by any of the libraries distributed on Debian and Ubuntu installations. This is more efficient and secure than storing the keys in memory. While this was not necessary for our setup, the compiler can also easily be modified not to use this register with minimal impact [44].

5.4 MPX and SFI

Figure 2 shows an example of MemSentry's instrumentation. We can see that the calculation of the pointer is separated from the store (the single `mov` becomes a `lea` and a `mov`). For

[2] We could hide the key in memory using one of the other memory isolation techniques, but this would defeat the purpose of using encryption in the first place.

```
; Store value of rdi in mem
mov     %rdi, 0x8(%rbx)
```

(a) Original

```
; Load pointer into rcx
lea     0x8(%rbx), %rcx

; Faults if rcx is above bnd0
bndcu   %rcx, %bnd0
; Pointer in rcx is now verified
mov     %rdi, (%rcx)
```

(b) MPX

```
; Load pointer into rcx
lea     0x8(%rbx), %rcx
; Mask pointer to be below safe area
movabs  $0x00003fffffffffff, %rax
and     %rax, %rcx
; Pointer in rcx is now verified
mov     %rdi, (%rcx)
```

(c) SFI

Figure 2. Code transformations caused by our address-based instrumentations. In both cases the calculation of the address and the store have been split (`lea` + `mov`). For MPX, the pointer is checked to be below 64 TB, whereas for SFI it is modified to always be below 64 TB (although this split is arbitrary in both cases).

SFI, we load the mask and use the `and` instruction to apply it, using the result for the store. This will force the memory access to always be below 64 TB, although the address space split can be anywhere in the 128 TB address space.

For MPX we insert a single bounds check, which will trigger a fault if the value of `rcx` is above the upper bound of `bnd0`, which we set to 64 TB during program initialization. Because we partition the address space, instead of relying on more fine-grained bounds, only the upper bound has to be checked. The lower bound of the *nonsensitive partition* is 0, and given addresses cannot be negative, a check would be useless. This saves both instrumentation, and slightly increases performance. This implementation does assume the `bnd0` register is not used by the application itself. As MPX is currently only used by compiler passes this is a reasonable assumption, but if an application were to use MPX itself overheads would be larger. Note also that MemSentry enables the `bndpreserve` flag and, therefore, MPX does not reload its bounds (from memory) at any point. Without this flag set, the CPU will load the bounds registers from memory for every branch instruction without a `BND` instruction prefix.

5.5 LLVM & points-to analysis

As our instrumentation is performed by an LLVM pass, we can make use of its optimization passes. For instance, in the IR that we instrument LLVM will have eliminated all register spilling to the stack, thus making sure we only see (and instrument) necessary memory accesses. Afterwards, during register allocation in the backend, LLVM might generate variable spills to the stack. These instructions access a fixed place in memory and thus do not need isolation instrumentation.

As shown in Table 1, many security defenses instrument solely branch instructions or system calls. MemSentry can trivially infer the instrumentation points in these cases. However, protecting arbitrary information requires further analysis, often called *points-to analysis*, for discovering which instructions operate on which data. In LLVM this can be done statically using the *data-structure analysis* (DSA) pass.

While MemSentry supports DSA, we noticed (similarly to other researchers [62]) that DSA is overly conservative, often yielding undesirable results where most memory accesses are classified as being able to touch sensitive data. We thus also looked at a run-time solution for *dynamic* analysis, in order to approximate an ideal (non-conservative) static analysis only for evaluation purposes. First, the program is prepared with initial instrumentation, which allows later analysis passes to map assembly instructions back to IR instructions. Then, the program is run with a PIN pass [43], which records all accesses to objects per instruction. The output of this run can then be fed back into the instrumentation pass as points-to information. The dynamic analysis itself is much slower, and there is a high chance of under-approximating memory accesses, since only accesses related to particular inputs (i.e., execution paths) are recorded.

We stress here that, even though points-to analysis is an open problem, it does not hinder MemSentry in most cases, where defenses already determine the instrumentation points. In cases where arbitrary data should be protected, any points-to analysis pass can easily be incorporated in the framework.

6. Evaluation

In order to evaluate the applicability and trade-offs between the use of different techniques, we evaluate combinations of isolation mechanisms and instrumentation points that are supported by MemSentry on the SPEC CPU2006 benchmark suite. SPEC is very memory and CPU intensive, and thus the overhead for I/O bound applications such as servers will be lower. All benchmarks were performed on a machine with an Intel i7-6700k processor clocked at 4GHz, with 16GB DDR4 RAM. We used Ubuntu 14.04 with Linux 3.19, recompiled to enable MPX support. However, the result for SGX shown in Table 4 was performed on the same system equipped with the similar E3-1240v5 processor at 3.5GHz, as not all i7's include SGX support.

In the rest of the section, we first look at microbenchmarks of all previously discussed techniques, and analyze sources of overhead. Then, we compare the performance overhead of address and domain-based techniques, and show real-world results for SafeStack. Finally, we discuss trade-offs between the two designs, and all the available hardware features.

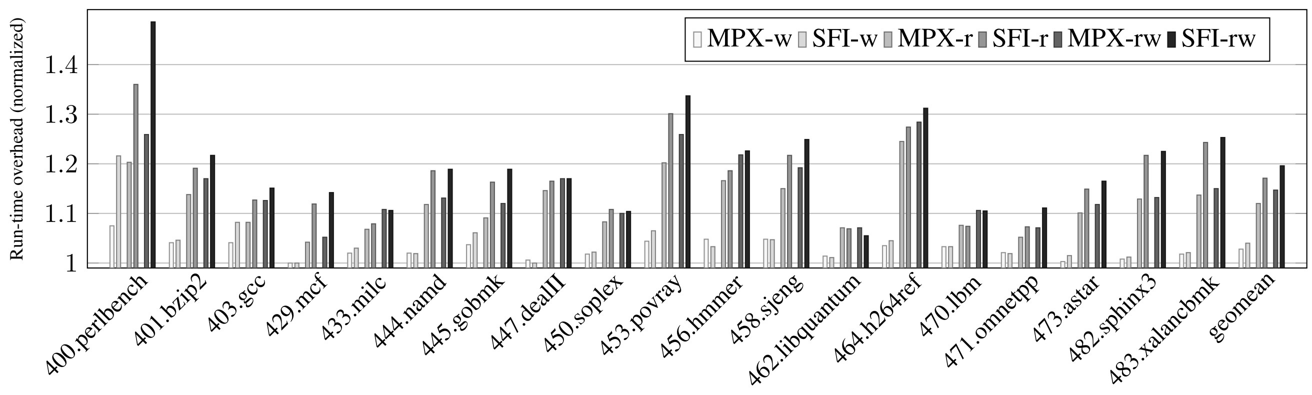

Figure 3. SPEC overhead for instrumentating all stores (*-w*), loads (*-r*) and both (*-rw*) for SFI and MPX, showing MPX introduced less overhead than SFI in most cases.

6.1 Microbenchmarks

The results of our microbenchmarks are shown in Table 4. We looked at both the cost of memory operations and of our isolation techniques, including some comparable operations. These results serve as a comparison between different operations, and the overall real-world overhead might be lower (e.g., due to pipelining optimizations) or higher (e.g., due to higher TLB pressure). Results were gathered by timing a tight loop of many iterations with the instruction, disabling interrupts and other external factors wherever possible. Our results are consistent with other sources [27, 38].

Memory operations are obviously highly dependant on caching behavior, with spatially and temporally local accesses being far more efficient. In practice the overall observed latency might be lower, depending on the data dependencies. These results give an idea of the cost of spilling registers, which might add to the overhead of some techniques.

The cost of an `and` operation is highly dependent on what is done with the result. When the result is not used, or used as the destination of a memory write, its overhead is not observable, most likely due to effects of pipelining and out-of-order execution. However, when we use the result of the `and` as a pointer for a memory load, its overhead does become significant as this introduces a data dependency.

For MPX we observed a huge performance difference between using a single bounds check, and performing a full bounds check (i.e., checking both the upper and lower bounds using `bndcu` and `bndcl`). The performance difference is mainly because the second bounds check instruction is delayed until the first one completes, whereas the first one does not delay anything. Further experiments confirm this, e.g., executing three bounds check instructions doubles the overhead to ∼1 cycle, compared to executing only two. Our measurements thus show that in MemSentry (with a single bounds check and 1 domain), MPX should be faster than SFI in basically all cases, as the overhead of SFI might be observed (when used for loads) whereas the MPX overhead will stay consistently low. In cases where more domains would be required (like GCC's bounds checking) this would no longer be true.

Instruction/operation	Cycles
L1 cache access	4[a]
L2 cache access	12[a]
L3 cache access	44[a]
DRAM access	251
SFI (`and`, result used by load)	0.22
SFI (`and`, result used by store)	0
MPX (single `bndcu`)	< 0.1
MPX (both `bndcl` and `bndcu`)	0.50
MPK (simulated)	0.42
`vmfunc` (EPT switch)	147
`vmcall`	613
`syscall`	108
SGX enter + exit enclave[b]	7664
AES encryption and decryption (11 rounds)	41
AES keygen (10 rounds)	121
AES imc (9 rounds)	71
Loading `ymm` into `xmm` (11 times)	10

[a] From Intel [38], in practice these will vary due to access patterns, out-of-order execution and pipelining.
[b] Time of performing an empty ECALL using the Intel SGX SDK for Linux.

Table 4. Microbenchmarks for the latency of hardware protection features and related operations.

Our virtualization experiments, testing `vmfunc` and a hypercall (`vmcall`), show that the use of `vmfunc` is indeed much more efficient than the old way of involving the hypervisor. We show that the cost of a `vmfunc` is similar to that of a traditional `syscall`.

Our AES results show that round-key generation is far more expensive than fetching the round-keys from the `ymm` registers. It also shows that calculating all required keys for decryption (using `aesimc` 9 times) is far more costly than extracting the normal round-keys (used directly for encryption). While the cost of encryption and decryption per chunk is the same, the initialization cost per block will thus be higher for decryption.

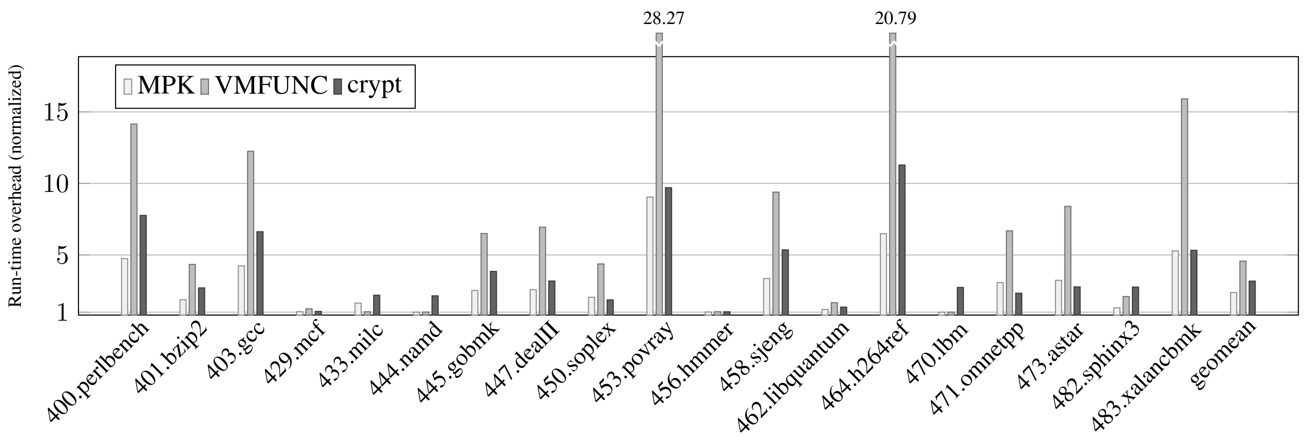

Figure 4. SPEC overhead for MPK, VMFUNC and crypt (on a single 128-bit chunk) when switching domains for every `call` and `ret` instruction, simulating a shadow stack.

6.2 Real-world performance

For a more practical analysis, we run MemSentry on SPEC CPU2006. We present numbers for each benchmark individually, and the geometric mean (geomean) over the set of all C and C++ benchmarks.

As address-based techniques (SFI and MPX) must check the pointers for every memory access, the instrumentation points are different than those of domain-based techniques. In order for address-based techniques to work, *all* memory accesses in an application must be instrumented, with the minor exception of those that are allowed access to sensitive data. In Figure 3 we show the results for instrumenting all read instructions, write instructions, and both reads and writes. As we instrument all memory accesses indiscriminately, these figures represent the worst-case scenario for these techniques. However, as an application mostly deals with non-sensitive data, there is in most cases no difference between instrumenting all instructions, or all minus a few.

As discussed in Section 4, the results for instrumenting only reads (*MPX-r* and *SFI-r*, geomean 12% and 17.1%) represent the overhead for CFI and code randomization defense solutions, whereas those with only writes (*MPX-w* and *SFI-w*, geomean 2.8% and 4%) indicate the overhead for protecting a shadow stack. The results for both reads and writes (*MPX-rw* and *SFI-rw*, geomean 14.7% and 19.6%) present a broader (and worst-case) protection, and are applicable when protecting arbitrary program data from disclosure and tampering, such as private keys.

We can see that in almost all cases, MPX performs better than SFI. Both instrumentations take very little time to execute, as shown in Table 4. However, in the case of SFI the instruction accessing memory has a dependency on the result of the `and`, whereas for MPX the `bndcl` does not modify the address. While for memory writes this overhead is not noticeable, it causes delays for reads.

Figures 4, 5, and 6 present the results for domain-based techniques. For VMFUNC we switch to a secondary, yet identical, EPT. For crypt, we use AES-NI on a single chunk (16 bytes), and retrieve all round keys from the upper parts of `ymm` registers. Figure 4, which shows the results when domain switches occur for every `call` and `ret` instruction, represents the worst-case scenario of all of these, and shows the case of protecting a shadow stack. Figure 5 presents a subset of the previous results, where only indirect branches are instrumented, corresponding to CFI and layout randomization solutions. Finally, Figure 6 shows results for system calls. We observed similar results for calls to the allocator.

Consistent with the microbenchmarks, MPK is the most efficient of the three with a geomean of 130%, 34%, and 1.1% for `call-ret`, indirect calls and system calls respectively. The costs of crypt (with geomeans of 217%, 60%, and 22%) are higher than might be expected, as not only must the data be encrypted (in 128-bit chunks) but the keys must also be copied into `xmm` registers before encryption can happen. The costs for VMFUNC are generally the highest (with geomeans of 357%, 82%, and 5.5%). Part of this overhead comes from the process-level virtualization, where all system calls are converted into hypercalls. This is especially noticeable for `syscall`-heavy benchmarks, and not as much on SPEC [5]. For benchmarks that already heavily rely on the `xmm` registers, crypt incurs a more significant performance overhead. This is especially evident for several floating-point benchmarks in Figure 6, where the encryption itself does not take much time, but clobbering a number of `xmm` registers does.

While applying memory isolation to SafeStack [40], a shadow stack implementation used in production and present in Clang [61], we opted for address-based isolation based on these results. SafeStack introduces no additional overhead on its own, as it simply replaces all stack loads and stores with accesses to the unsafe stack. We found that when ap-

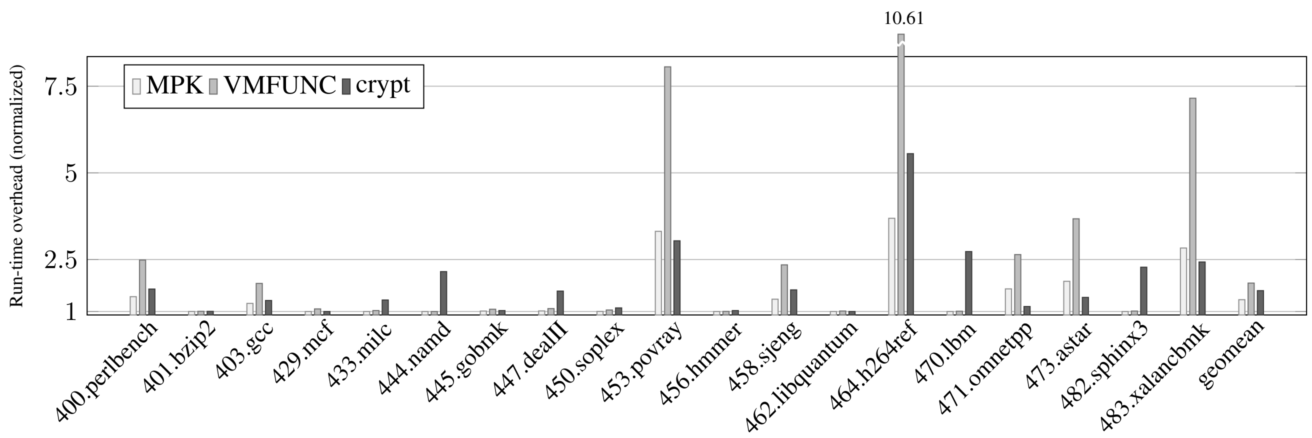

Figure 5. SPEC overhead for MPK, VMFUNC and crypt (on a single 128-bit chunk) when switching domains for every indirect branch.

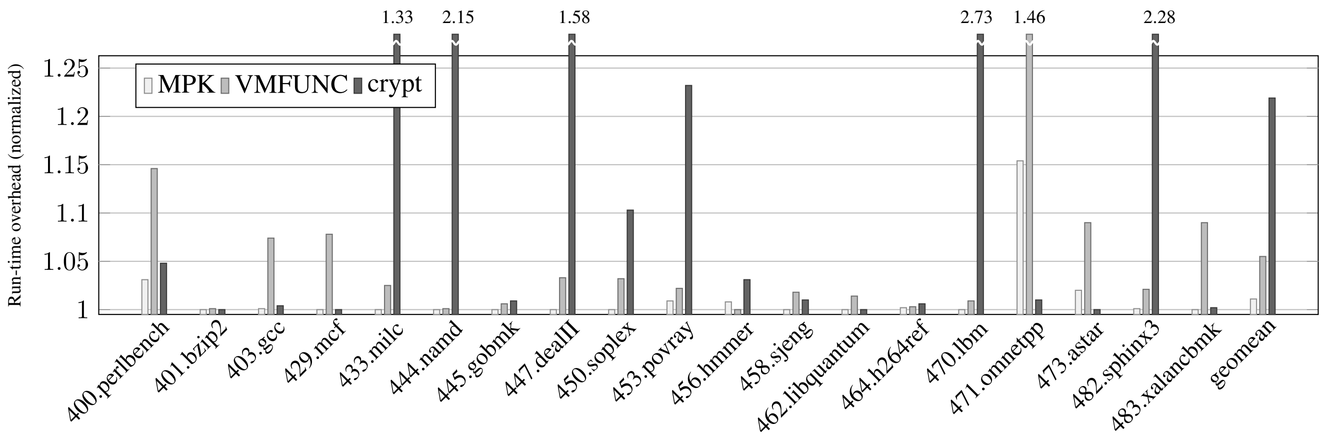

Figure 6. SPEC overhead for MPK, VMFUNC and crypt (on a single 128-bit chunk) when switching domains for every system call.

plying MemSentry, SafeStack still introduced no additional overhead, and the results were identical to Figure 3.

All previous results assume a very small safe region in the case of crypt: a single native 128-bit (16 byte) value. Further experiments showed that in this case, most of its overhead is due to the initialization of round keys. Encryption of larger sizes increases linearly on top of this initial cost. However, we still observed an approximately 15x overhead when protecting a region of 1024 bytes.

6.3 Discussion

In this section, we look at further trade-offs for the different memory isolation techniques, evaluate the merits of each approach and, combined with the performance figures, give an overview of these techniques in practice. We hope this will help future researchers with deciding between such techniques, but we should note that all techniques are practical, depending on the situation (e.g., an older processor).

For address-based isolation techniques, MPX has many advantages over SFI. First of all, the performance of MPX is higher in almost all cases. It also provides a more flexible way of partitioning the address space, as bounds can be placed at any point in memory. Additionally, MPX deterministically informs the system of an invalid attempt, whereas with SFI the masked pointer may still be valid (in the nonsensitive partition). On the other hand, the biggest downside of MPX is that it requires a relatively new Intel CPU (Skylake architecture, released 2015). MPX also becomes much less favorable when many different domains are required, and because bounds must continuously be spilled to memory. Furthermore, in a situation with arbitrary bounds, where both upper and lower bounds must be checked, the overhead also becomes worse: our experiments showed it to be slightly worse than our SFI results. However, SFI cannot support such a situation at all without the need for far more expensive checks.

For domain-based solutions, we would expect MPK to have much higher performance than alternatives, based on our approximated benchmarks. However, the biggest issue with MPK is that it has not yet been released, nor have processors supporting it been announced. Until that time, the choice remains between VMFUNC and crypt. The cost of crypt increases linearly with the size of the isolated domain, whereas the cost remains constant for VMFUNC. This smaller granularity can also be an advantage: for smaller data (1–2 128-bit chunks) crypto is generally faster, and it does not require the data to be placed in separate pages. On the other hand, VMFUNC requires both fairly recent hardware (Intel Haswell or newer, available since 2013) and a (relatively small) piece of privileged code on the host: a modified hypervisor. AES-NI has been present in CPUs for far longer, beginning with the Intel Westmere architecture in 2010, and might thus be more widely and easier deployable. Finally, SGX, while an interesting technique useful for cloud environments, is not practical for the relatively lightweight isolation as discussed in this paper.

When choosing between addressing-based and domain-based techniques, for MPX versus MPK, the optimal choice primarily depends on how often domain switches occur in practice (in other words, what portion of the instructions executed access isolated data). When this happens frequently, such as for every `call` and `ret` instruction, addressing-based approaches are more favorable.

7. Related work

Software-fault isolation has been actively researched over the past 20 years [25, 45, 65]. For instance, Native Client [56, 70] compiles programs written in C/C++ to verifiable sandboxed code, that run natively yet cannot execute arbitrary branches or memory accesses. LR^2 reduces SFI checks and achieves execute-only memory (XoM) by instrumenting instructions so that all pointers are masked [9]. Although LR^2 targets mobile devices and a particular application (resilient to leakage code randomization), the SFI optimization can be incorporated easily to MemSentry and implemented using hardware features of x86-64 such as MPX.

In terms of hardware-based isolation techniques, a lot of options exist across different architectures, many of which could be implemented as part of MemSentry. For instance, *segmentation* present in x86 (and dropped for x86-64) was used as an easy way of isolating memory [28, 65, 70]. ISBoxing [23] uses instruction prefixes to bound load/stores to a memory range, however, this significantly reduces the available address space. Similar features are present in other architectures, such as *memory domains* in ARM32 [74] (but removed from modern AArch64), which work almost identically to Intel's upcoming *MPK* feature for x86-64, except that the domain permissions can only be modified in supervisor mode. ARM also supports *TrustZone*, offering both a normal and secure world which are completely isolated, leveraged by TZ-RPK and Kenali [2, 59]. Usage of a *trusted platform module* allows pieces of application logic to securely run, and can leverage either hardware based solutions such as Intel TXT [46] or software-based solutions [47]. Sanctum [16] provides SGX-like features in hardware for memory isolation.

Several systems have been proposed to prevent stealing keys from memory [44, 55], including the use of *hardware-transactional memory* [34]. AMD's upcoming secure memory encryption (SME) encrypts all memory transparently, protecting it from outside analysis and cold-boot attacks. Secure encrypted virtualization (SEV) expands this to allow for per-VM memory encryption [39]. However, both SME and SEV do not help against local memory exploits. Isolating complete applications running in untrusted environments has been also studied. For example, Intel SGX can be used to protect applications running in untrusted clouds [4, 72], while other approaches are based on *virtualization* [11, 13, 41, 57, 69]. For instance, SeCage [41] offers domains that can be efficiently switched between using `vmfunc`, similarly to what was discussed in Section 3.1, and includes automatic application decomposition using points-to analysis. Both the decomposition of software and the protection of domain switches has been a major issue for such systems. Systems such as Readactor [18] and Heisenbyte [60] similarly use the EPT feature of VT-x to achieve XoM and destructive code-reads respectively. Finally, entirely new hardware features have been proposed to achieve memory isolation [64, 68].

We takes inspiration from a number of these approaches in how to leverage hardware-features to provide memory isolation. All of the aforementioned systems implement a custom protection scheme, supporting only a single hardware-feature, and often supporting only a single defense-system. In contrast, we provide a more general form of memory isolation on x86-64, and a general-purpose design supporting different isolation techniques for defense systems.

8. Conclusion

In this paper we explored the importance of isolation in realizing software defenses. We reviewed a series of systems which rely on information hiding for securing some vital component for their operation. We argued that such probabilistic isolation is insufficient, and presented a design that can apply hardware features to provide *deterministic* isolation. We introduced MemSentry, a framework that can easily provide strong memory isolation to existing defense systems, and a practical evaluation of such hardware features.

We believe that MemSentry is useful for the community, since it provides a vital feature for many defense systems: the ability to properly isolate sensitive data from the rest of the process. Additionally, it helps further research towards this goal by offering a platform for testing hardware-supported isolation. MemSentry can be found at `http://github.com/vusec/memsentry`.

Acknowledgments

We thank our shepherd, Andrew Baumann, and the anonymous reviewers for their valuable feedback. This work was supported by the European Commission through project H2020 ICT-32-2014 "SHARCS" under Grant Agreement No. 64457 and the Netherlands Organisation for Scientific Research through grant NWO 639.023.309 VICI "Dowsing".

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS*, 2005.

[2] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision across worlds: Real-time kernel protection from the ARM TrustZone secure world. In *CCS*, 2014.

[3] M. Backes and S. Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *USENIX SEC*, 2014.

[4] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with Haven. In *OSDI*, 2014.

[5] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *OSDI*, 2012.

[6] E. D. Berger and B. G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *PLDI*, 2006.

[7] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi. Timely rerandomization for mitigating memory disclosures. In *CCS*, 2015.

[8] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida. Dedup Est Machina: Memory deduplication as an advanced exploitation vector. In *S&P*, 2016.

[9] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A. R. Sadeghi. Leakage-resilient layout randomization for mobile devices. In *NDSS*, 2016.

[10] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *USENIX SEC*, 2014.

[11] H. Chen, J. Chen, W. Mao, and F. Yan. Daonity - grid security from two levels of virtualization. *Inf. Secur. Tech. Rep.*, 12(3), 2007.

[12] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *USENIX SEC*, 2005.

[13] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS*, 2008.

[14] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida. StackArmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *NDSS*, 2015.

[15] X. Chen, H. Bos, and C. Giuffrida. CodeArmor: Virtualizing the code space to counter disclosure attacks. In *EuroS&P*, 2017.

[16] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX SEC*, 2016.

[17] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *USENIX SEC*, 2003.

[18] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A. R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *S&P*, 2015.

[19] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *ASIACCS*, 2015.

[20] L. Davi, R. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A. R. Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *NDSS*, 2012.

[21] L. Davi, A. R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX SEC*, 2014.

[22] L. Davi, C. Liebchen, A. R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *NDSS*, 2015.

[23] L. Deng, Q. Zeng, and Y. Liu. ISboxing: An instruction substitution based data sandboxing for x86 untrusted libraries. In *IFIP SEC*, 2015.

[24] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. The matter of heartbleed. In *IMC*, 2014.

[25] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *OSDI*, 2006.

[26] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In *S&P*, 2015.

[27] A. Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. *Copenhagen University College of Engineering*, 2016.

[28] B. Ford and R. Cox. Vx32: Lightweight user-level sandboxing on the x86. In *USENIX ATC*, 2008.

[29] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz. Enabling client-side crash-resistance to overcome diversification and information hiding. In *NDSS*, 2016.

[30] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX SEC*, 2012.

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

[32] E. Göktaş, R. Gawlik, B. Kollenda, E. Athanasopoulos, , G. Portokalidis, C. Giuffrida, and H. Bos. Undermining entropy-based information hiding (and what to do about it). In *USENIX SEC*, 2016.

[33] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *NDSS*, 2017.

[34] L. Guan, J. Lin, B. Luo, J. Jing, and J. Wang. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *S&P*, 2015.

[35] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd my gadgets go? In *S&P*, 2012.

[36] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space ASLR. In *S&P*, 2013.

[37] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. April 2016.

[38] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. January 2016.

[39] D. Kaplan, J. Powell, and T. Woller. AMD memory encryption. Technical report, AMD, April 2016. URL `http://bit.ly/2gr5hQM`.

[40] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *OSDI*, 2014.

[41] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *CCS*, 2015.

[42] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee. ASLR-Guard: Stopping address space leakage for code reuse attacks. In *CCS*, 2015.

[43] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[44] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. CCFI: Cryptographically enforced control flow integrity. In *CCS*, 2015.

[45] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *USENIX SEC*, 2006.

[46] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *EuroSys*, 2008.

[47] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *S&P*, 2010.

[48] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz. Opaque control-flow integrity. In *NDSS*, 2015.

[49] B. Niu and G. Tan. Monitor integrity protection with space efficiency and separate compilation. In *CCS*, 2013.

[50] B. Niu and G. Tan. Per-input control-flow integrity. In *CCS*, 2015.

[51] G. Novark and E. D. Berger. DieHarder: Securing the heap. In *CCS*, 2010.

[52] A. Oikonomopoulos, C. Giuffrida, E. Athanasopoulos, and H. Bos. Poking holes into information hiding. In *USENIX SEC*, 2016.

[53] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer. Intel MPX explained: An empirical study of Intel MPX and software-based bounds checking approaches. *arXiv*, 2017.

[54] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *S&P*, 2012.

[55] T. P. Parker and S. Xu. A method for safekeeping cryptographic keys from memory disclosure attacks. In *INTRUST*, 2009.

[56] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary CPU architectures. In *USENIX SEC*, 2010.

[57] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-VM monitoring using hardware virtualization. In *CCS*, 2009.

[58] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. R. Sadeghi. Just-In-Time code reuse: On the effectiveness of fine-grained address space layout randomization. In *S&P*, 2013.

[59] C. Song, B. Lee, K. Lu, W. R. Harris, T. Kim, and W. Lee. Enforcing kernel security invariants with data flow integrity. In *NDSS*, 2016.

[60] A. Tang, S. Sethumadhavan, and S. Stolfo. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *CCS*, 2015.

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

[62] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical context-sensitive CFI. In *CCS*, 2015.

[63] V. van der Veen, E. Göktaş, M. Contag, A. Pawloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *S&P*, 2016.

[64] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero. CODOMs: Protecting software with code-centric memory domains. In *ISCA*, 2014.

[65] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP*, 1993.

[66] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *CCS*, 2012.

[67] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello. Shuffler: Fast and deployable continuous code re-randomization. In *OSDI*, 2016.

[68] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *ISCA*, 2014.

[69] J. Yang and K. G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *VEE*, 2008.

[70] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *S&P*, 2009.

[71] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *S&P*, 2013.

[72] F. Zhang and H. Zhang. SoK: A study of using hardware-assisted isolated execution environments for security. In *HASP*, 2016.

[73] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX SEC*, 2013.

[74] Y. Zhou, X. Wang, Y. Chen, and Z. Wang. ARMlock: Hardware-based fault isolation for ARM. In *CCS*, 2014.