IMIX: In-Process Memory Isolation EXtension

Tommaso Frassetto Patrick Jauernig Christopher Liebchen Ahmad-Reza Sadeghi
Technische Universität Darmstadt, Germany
{tommaso.frassetto, patrick.jauernig, christopher.liebchen, ahmad.sadeghi}@trust.tu-darmstadt.de

Abstract

Memory-corruption attacks have been subject to extensive research in the latest decades. Researchers demonstrated sophisticated attack techniques, such as (just-in-time/blind) return-oriented programming and counterfeit object-oriented programming, which enable the attacker to execute arbitrary code and data-oriented attacks that are commonly used for privilege escalation. At the same time, the research community proposed a number of effective defense techniques. In particular, control-flow integrity (CFI), code-pointer integrity (CPI), and fine-grained code randomization are effective mitigation techniques against code-reuse attacks. All of these techniques require strong memory isolation. For example, CFI's shadow stack, CPI's safe-region, and the randomization secret must be protected from adversaries able to perform arbitrary read-write accesses.

In this paper we propose IMIX, a lightweight, in-process memory isolation extension for the Intel-based x86 CPUs. Our solution extends the x86 ISA with a new memory-access permission to mark memory pages as *security sensitive*. These memory pages can then only be accessed with a newly introduced instruction. Unlike previous work, IMIX is not tailored towards a specific defense (technique) but can be leveraged as a primitive to protect the data of a wide variety of memory-corruption defenses. We provide a proof of concept of IMIX using Intel's Simulation and Analysis Engine. We extend Clang/LLVM to include our new instruction, and enhance CPI by protecting CPI's safe region using IMIX.

1 Introduction

Memory-corruption attacks have been a major threat against modern software for multiple decades. Attackers leverage memory-corruption vulnerabilities to perform multiple malicious activities including taking control of systems and exfiltrating information. Memory-corruption attacks can be roughly divided into the categories code-injection [3], code-reuse [50, 52, 54], and data-only attacks [12, 28, 29]. While code-injection attacks introduce new malicious code into the vulnerable program, code-reuse attacks reuse the existing code in an unintended way. Data-only attacks in turn aim to influence the program behavior by modifying crucial data variables, e.g., used in branching conditions.

Defenses against memory-corruption typically reduce the attack surface by preventing the adversary from corrupting part of the application's memory which is essential for a successful attack. Prominent examples include: W$\oplus$X [44, 48] which prevents data from being executed, and hence, code-injection attacks; Control Flow Integrity (CFI) [1] and Code-Pointer Integrity (CPI) [38] which protect code pointers to prevent code-reuse attacks; and Data Flow Integrity (DFI) [2, 10] mitigating data-only attacks by restricting data access.

Some of these defenses can be implemented efficiently using mechanisms that reside entirely outside the underlying application process. For instance, the kernel configures W$\oplus$X and the hardware enforces it. Hence, the adversary cannot tamper with this defense mechanism when exploiting a memory-corruption vulnerability in the application. However, using an external mechanism is not always feasible in practice due to high performance overhead. For instance, CFI requires run-time checks and a shadow stack [1, 9, 18], which is updated every time a function is invoked or returns. CPI requires run-time checks and a *safe region*, which contains metadata about the program's variables. The required code for these defenses can be efficiently protected when marked as read-only, just like the application code. However, as of today no architectural solution exists that protects the data region of these defenses from unintended/malicious accesses. This data cannot be stored outside of the process, e.g., in kernel memory, because accessing it would impose an impractical performance overhead due to the time needed for a context switch. Hence, to pre-

vent the adversary from accessing the data some form of *in-process memory isolation* is needed, i.e., a mechanism ensuring access only by the defense code while denying access by the potentially vulnerable application code. However, devising a memory isolation scheme for current x86 processors is challenging.

Memory Isolation Approaches. A variety of memory isolation solutions have been proposed or deployed both in software and/or hardware. Software solutions use either access instrumentation [8, 61], or data hiding [6, 38]. Instrumentation-based memory isolation inserts run-time checks before every memory access in the untrusted code in order to prevent accesses to the protected region. However, it imposes a substantial performance overhead, for instance, code instrumented using Software Fault Isolation (SFI) incurs an overhead up to 43% [51]. Data hiding schemes typically allocate data at secret random addresses. Modern processors have sufficiently large virtual memory space (140 TB) to prevent brute-force attacks. The randomized base address must be kept secret and is usually stored in a CPU register. However, ensuring that this secret is not leaked to the adversary is challenging, especially if the program is very complex. For instance, compilers sometimes save registers to the stack in order to make room for intermediate results from some computation. This is known as *register spilling* and can leak the randomization secret [14]. Moreover, even a large address space can successfully be brute-forced as it was shown on an implementation of CPI [22, 24]. Thus, current in-process memory isolation either compromises performance or offers limited security.
Memory protection based on hardware extensions is another approach to achieve in-process isolation. For instance, Intel has recently announced Control-flow Enforcement Technology [33] and Memory Protection Keys [34] (already available on other architectures, e.g. *memory domains* on ARM32 [4]). However, these technologies either provide hardware support limited to a specific mitigation, or cause unnecessary performance overhead. We will discuss those technologies in a more detailed way in Section 8.

Goals and Contributions. In this paper we present IMIX, which enables lightweight in-process memory isolation for memory-corruption defenses that target the x86 architecture. IMIX enables *isolated pages*. Marked with a special flag, isolated pages can only be accessed using a single new instruction we introduce, called `smov`. Just like defenses like W⊕X protect the code of run-time defenses from unintended modifications, IMIX protects the data of those defenses from unintended access. In contrast to other recently proposed hardware-based approaches we provide an agnostic ISA extension that can be leveraged by a variety of defenses against code-reuse attacks to increase performance and security. To summarize, our main contributions are:

- **Hardware primitive to isolate data memory**. We propose IMIX, a novel instruction set architecture (ISA) extension to provide effective and efficient in-process isolation that is fundamental for the security of memory-corruption defenses (metadata protection). Therefore, IMIX introduces a new memory-access permission to protect the isolated pages, which prevents regular load and store instructions from accessing this memory. Instead, the code part of defense mechanisms needs to use our newly introduced `smov` instruction to access the protected data.
- **Proof-of-concept implementation**. We provide a fully-fledged proof of concept of IMIX. In particular, we leverage Intel's Simulation and Analysis Engine [11] to extend the x86 ISA with our new memory protection, and to add the *smov* instruction. Further, we extend the Linux kernel to support our ISA extension and the LLVM compiler infrastructure to provide primitives for allocation of protected memory, and access to the former. Finally, we demonstrate how defenses against memory-corruption attacks benefit from using IMIX by porting code-pointer integrity (CPI) [38] to leverage IMIX to isolate its safe-region.
- **Thorough evaluation**. We evaluate the performance by comparing our IMIX-enabled port of CPI to the original x86-64 variant. Further, we compare our solution to Intel's Memory Protection Keys and Intel's Memory Protection Extensions [34] overhead for CPI.

2 Background

In this section we provide the necessary technical background which is necessary for understanding the remainder of this paper. We first provide a brief summary of memory corruption attacks and defenses, and then explain memory protection on the x86 architecture.

2.1 Memory Corruption

C and C++ are popular programming languages due to their flexibility and efficiency. However, their requirement for manual memory management places a burden on developers, and mistakes easily result in memory-corruption vulnerabilities which enable attackers to change the behavior of a vulnerable application

during run time. For example, a missing bounds check during the access of a buffer can lead to a buffer overflow, which enables the attacker to manipulate adjacent memory values. With a write primitive in hand the attacker can achieve different levels of control of the target, such as changing data flows within the application, or hijacking the control flow. When conducting a data-flow attack [28, 29], the attacker manipulates data pointers and variables that are used in conditional statements to disclose secrets like cryptographic keys. In contrast, during a control-flow hijacking attack, the attacker overwrites code pointers, which are later used as a target address of an indirect branch, to change control flow to execute injected code [3] or to conduct a code-reuse attack [50, 52, 54].

There exist different approaches to mitigate these attacks, however, they all have in common that they are part of the same execution context as the vulnerable application, and often make a tradeoff between practicality and security.

For example, combining SoftBounds [46] and CETS [47] guarantees memory safety for applications written in C, and hence, prevent the exploitation of memory-corruption vulnerabilities in the first place. Unfortunately, these guarantees come with an impractical performance overhead of more than 100%. To limit the performance impact, other mitigation techniques focus on mitigating certain attack techniques. To mitigate control-flow hijacking attacks, these techniques prevent the corruption of code pointers [38], verify code pointers before they are used [1], or ensure that the values of valid code pointers are different for each execution [16].

Another common aspect of every memory-corruption mitigation technique is that they reduce the attack surface of a potentially vulnerable application to the mitigation itself. In other words, if the attacker is able to manipulate the mitigation or memory on which the mitigation depends, she can undermine the security of the mitigation. The protection mitigation's memory is hard because it is part of the memory which the attacker can potentially access.

Next, we provide a short overview memory protection techniques, which are available on the x86 architecture, that can be leveraged to protect the application's and mitigation's memory.

2.2 Memory Isolation

The x86 architecture offers different mechanisms to enforce memory protection. *Segmentation* and *paging* are the most well-known ones. However, recently, Intel and AMD proposed a number of additional features to protect and isolate memory. As we argue in Section 8, IMIX is most likely to be adapted for Intel-based x86 CPUs, hence, we focus in this section on memory protection features that are implemented or will be implemented for Intel-based x86 CPUs. Note that in most cases AMD provides a similar feature using different naming convention. Finally, we shortly discuss software-based memory isolation.

Traditional Memory Isolation. Segmentation and paging build a layer of indirection for memory accesses that can be configured by the operating system, and the CPU enforces access control while resolving the indirection.

Segmentation is a legacy feature that allows developers to define segments that consists of a start address, size, and an access permission. However, on modern 64-bit systems access permissions are no longer enforced. Nevertheless, many mitigations [6, 18, 38, 41] leverage segmentation to implement information hiding by allocating their data TCB at a random address, and ensure that it is only accessed through segmentation.

On modern systems, paging creates an indirection that maps *virtual memory* to *physical memory*. The mapping is configured by the operating system through a data structure known as *page tables*, which contain the translation information and a variety of access permissions. The paging permission system enables the operating system to assign memory to either itself or to the user mode. To isolate different processes from each other, the operating system ensures that each process uses its own page table. Due to legacy reasons, paging does not differentiate between the read and execute permission, which is why modern systems feature the "non-executable" permission. Further, paging allows to mark memory as (non-)writable.

New Memory Protection Features. Recently introduced or proposed features that enable memory isolation on x86 are Extended Page Tables (EPT), Memory Protection Extensions (MPX), Software Guard Extensions (SGX), Memory Protection Keys (MPK) and Control-flow Enforcement Technology (CET). We provide a comparison in Section 9.

The EPT facilitate memory virtualization and are conceptually the same as regular page tables, except that they are configured by the hypervisor, and allow to set the read/write/execute permission individually. Hence, previous work leveraged the EPT to implement execute-only memory [16, 58, 63]. MPX implements bounds checking in hardware. Therefore, it provides new instructions to configure a lower and upper bound for a pointer to a buffer. Then, before a pointer is dereferenced, the developer can leverage another MPX instruction to quickly check whether this address points into the buffers boundaries. SGX allows to create *enclaves* within a process

that are completely isolated from the rest of the system at the cost of high overhead when switching the execution to the code within an enclave. MPK introduces a new register, which contains a protection key, and enables programmers to tag memory (the tag is stored in the page table) such that it can only be accessed if the protection key register contains a specific key. MPK can be utilized to implement in-process isolation by tagging the security critical data and loading the corresponding key only when executing a benign access, and deleting it after the access succeeded. Intel's hardware support for CFI, CET, provides similar memory isolation the shadow stack as IMIX for security critical data in general. It introduces a new access permission for the shadow stack, and special instructions to access it. Unfortunately, CET is tailored towards CFI and cannot be easily repurposed for other mitigations.

Software-based Approaches. Software Fault Isolation (SFI) [43, 51, 61] instruments every read, write, and branch instruction to enable in-process isolation. However, this approach comes with a significant performance overhead due to the additional instructions.

To summarize, none of the above listed memory protection features provides mitigation-agnostic security and performance benefits at the same time.

3 Adversary Model

Throughout our work, we use the following standard adversary model and assumptions, which are consistent with prior work in this field of research [21, 38, 53, 54].

- **Memory corruption.** We assume the presence of a memory-corruption vulnerability, which the adversary can repeatedly exploit to read and write data according to the memory access permissions.

- **Sandboxed code execution.** The adversary can execute code in an isolated environment. However, the executed code cannot interfere with the target application by any means other than by using the memory corruption vulnerability. In particular, this means that the sandboxed code cannot execute the `smov` instruction with controlled arguments. Arbitrary code execution is prevented by hardening the target application with techniques such as CPI [38], CFI [1], or code randomization [16]. However, the attacker can target those defenses as well using the memory corruption vulnerability. We assume memory-corruption mitigations cannot be bypassed unless the attacker can corrupt the mitigation's metadata.

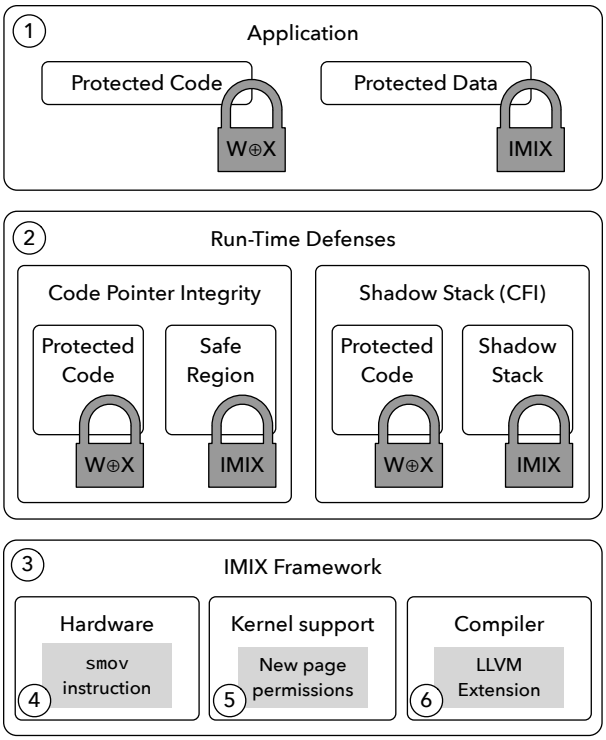

Figure 1: Overview of IMIX.

- **Immutable code.** The adversary cannot inject new code or modify existing code, which would allow her to execute the `smov` instruction with controlled arguments. This is enforced by hardening the target application with the W⊕X memory policy [44, 48].

4 IMIX

As we mentioned in Section 1, application developers protect their applications (① in Figure 1) using run-time defenses ②. Like for applications, the correct functionality of defenses relies on the integrity of their code and data. A number of existing run-time defenses, like CPI and CFI, require to keep their data within the process of the vulnerable application to avoid a high performance overhead. Thus, the attacker may leverage a memory-corruption vulnerability in the application to bypass those defenses [21]. Traditionally, defense developers enforce the integrity of the (static) code using W⊕X or execute-only memory, while the integrity of the data relies on some form in-process memory isolation. However, existing memory isolation techniques, namely instrumentation and data hiding, force the defense developers to choose between high performance overheads and compromised security. IMIX ③ provides an efficient, secure, hardware-enforced in-process memory isolation mechanism. Data belonging to run-time mitiga-

tions is allocated in *isolated pages*, which are marked with a special access permission. We introduce a new dedicated instruction, `smov` ④, to access this data, while normal code belonging to the potentially vulnerable application is denied access to the isolated pages.

In addition to the `smov` instruction and the associated access permissions, IMIX includes a kernel extension ⑤ and compiler support ⑥. The kernel extension enables protected memory allocation by supporting the special access permission. IMIX's compiler integration enables applications as well as run-time defenses to leverage our memory isolation through high-level and low-level constructs for protected memory allocation and access. This makes it easy to adopt IMIX without detailed knowledge of IMIX's implementation.

In the following, we explain the individual building blocks of our IMIX framework in detail.

Hardware. For IMIX, we extend two of the CPU's main responsibilities, instruction processing and memory management. We add our `smov` instruction to the instruction set, reusing the logic of regular memory access instructions, so that the `smov` instruction has the same operand types of regular memory-accessing `mov` instructions, `mov` instructions without a memory operand do not need to be handled. The memory access logic is modified so that it will generate a fault if 1) an instruction other than `smov` is used to access a page protected by IMIX, or if 2) an `smov` instruction is used to access a normal page. Access by normal instructions to normal memory, and by `smov` instructions to protected memory, are permitted. If we allowed `smov` to access normal memory, attacks on metadata would be possible, e.g., the attacker could overwrite a pointer to CPI's metadata with an address pointing to an attacker-controlled buffer in normal memory. Our design ensures instructions intended to operate on secure data cannot receive insecure input.

Kernel. An operating system kernel controls the user-space execution environment and hardware devices. The kernel manages virtual memory using *page tables* that map the address of each page to the physical page frame that contains it. Each page is described by a *page table entry*, which also contains some metadata, including the access permissions for that page. A user-space program can request a change in its access permissions to a page through a system call.

We extend the kernel to support an additional access permission, which identifies all pages protected by IMIX. This enables protected memory allocation not only for statically compiled binaries, but also for code generated at run time, which has been an attractive target for recent attacks [23].

Compiler. A compiler makes platform functionality available as high-level constructs to developers. Its main objective is to transform source code to executables for a particular platform. We extend the compiler on both ends. First, IMIX provides two high-level primitives: one for allocating protected memory and one for accessing it. These memory-protection primitives can either be used to build mitigations, or to protect sensitive data directly. IMIX provides optimized interfaces for both use cases. Mitigations like CPI are implemented as an LLVM optimization pass that works at the intermediate representation (IR) level. IMIX provides IR primitives to use for IR modification. For application developers, IMIX provides source code annotations: variables with our annotation will be allocated in protected memory, and all accesses will be through the `smov` instruction.

5 Implementation

Figure 2 provides an overview of the components of IMIX. Developers can build programs with IMIX, using our extended Clang compiler ①, which supports annotations for variables that should be allocated in protected memory and new IR instructions to access the protected memory. We also modified its back end to support `smov` instructions. Programs protected by IMIX mark isolated pages using the system call `mprotect` with a special flag ②. Therefore, we extended the kernel's existing page-level memory protection functionality to support this flag and mark isolated pages appropriately ④. User-space programs access normal memory using regular instructions, e.g., `mov`, while accesses to protected memory must be performed using the instruction `smov` ③. To support IMIX, the CPU must be modified to support the `smov` instruction ⑤ and must perform the appropriate checks when accessing memory ⑥. In the following we explain each component in detail.

5.1 CPU Extension

As we mentioned in Section 4, every isolated page needs to be marked with a special flag. The CPU already has a data structure to store information about every page, which is called a Page Table Entry (PTE). In addition to the physical address of every virtual page, a PTE stores other metadata about the page, including permissions like writable and executable. Those flags are checked by the Memory Management Unit (MMU) to prevent unintended accesses. To implement our proof of concept, we mapped the IMIX protection flag to an ignored bit in the PTE; specifically, we chose bit 52, as it is the first bit not reserved, and is normally ignored by the MMU [31].

To enforce hardware protection, the CPU needs to be updated to enforce our access policy: non-`smov` can only

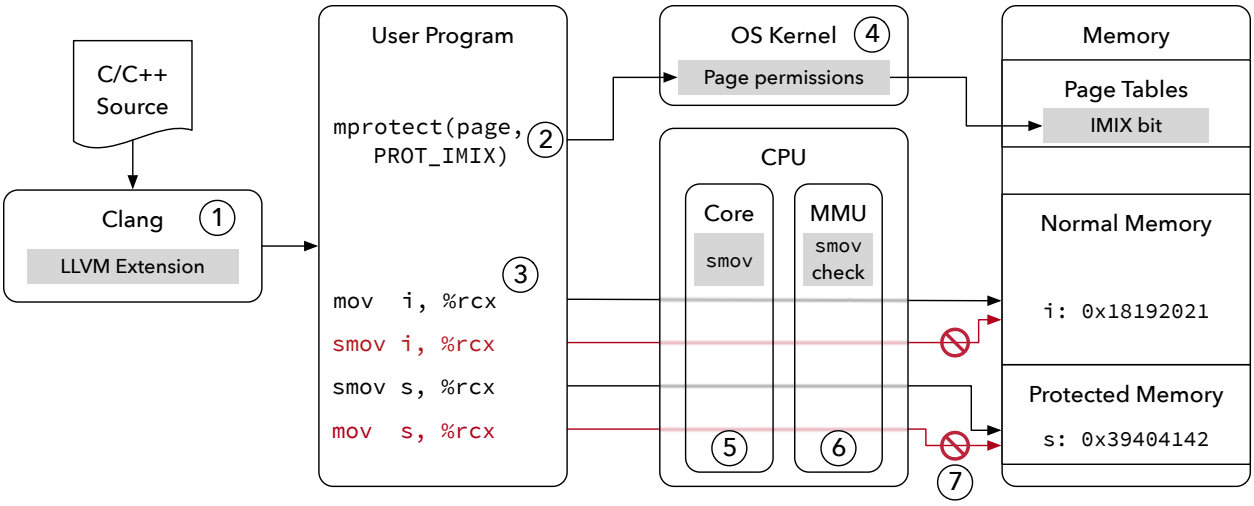

Figure 2: Overview of IMIX.

access regular pages, while `smov` can only access isolated pages. In other cases, the CPU must generate a fault (⑦ in Figure 2). The implementation of this logic requires the modification of the x86-64 ISA, which is challenging without source code access. Thus, we used a hardware simulator to show the feasibility of our design. Next, we describe how we extend x86-64 with the help of Intel's SAE, and then discuss the necessary modification to real hardware.

Simulated Hardware. We use Wind River Simics [64], a full system simulator, in order to simulate a complete computer which supports IMIX. Yet, Simics alone is too slow to boot the Linux kernel and test our kernel extension. Therefore we use the complementary Intel Simulation and Analysis Engine (SAE) add-on by Chachmon et al. [11]. Below we will refer to the system composed by Simics and SAE as simply SAE. SAE supports emulating an x86 system running a full operating system with its processes, while allowing various architectural instrumentations, including the CPU, the memory, and related hardware such as the memory management unit (MMU). This is done using extensions, called *ztools*, that may be loaded and unloaded at any time during emulation. They are implemented as shared libraries written in C/C++.

To instrument a simulated system, ztools registers callbacks for specific hooks either at initialization time or dynamically. First, we make sure that our ztool is initialized by registering a callback for the initialization hook. Then, we register a callback that is executed when an instruction is added to the CPU's instruction cache. If either a `mov` or `smov` instruction that accesses memory is found, we register an instruction replacement callback. Our registered callback handler can replace the instruction (using a provided C function), or execute the original instruction. In this handler, we implement IMIX's access logic. First, we check the protection flag of the memory accessed by the instruction. To identify protected memory, we look up the related PTE by combining the virtual address and the base address of the page table hierarchy linked from the `CR3` register. Our ztool then checks the IMIX page flag we introduced in the PTE.

If a regular instruction attempts to access regular memory, we execute the original instruction to avoid instruction cache changes. For `smov` instructions attempting to access an isolated page, we first remove the instruction from the instruction cache, and then execute our ztool implementation of this instruction. In the remaining cases, namely `smov` attempting to access regular memory, and regular instructions attempting to access isolated pages, we raise a fault.

Real Hardware. Adding IMIX support to a real CPU would require extending the CPU's instruction decoder to make it aware of our `smov` instruction. `smov` requires the same logic as the regular `mov` instruction, so the existing implementation could be reused. Moreover, we need to modify the MMU to perform the necessary checks. Analogously to W⊕X, we check the flag in the page table entry (PTE) belonging to the virtual address, and either permit or deny memory access. Modern MMUs are divided into three major components: logic for memory protection and segmentation, the translation lookaside buffer (TLB) which caches virtual to physical address mappings, and page-walk logic in case of a cache miss [49]. Our extension only modifies the first component to implement the access policy based on the current CPU instruction. Other components do not need to be modified, as we are using an otherwise ignored bit in the

PTEs. In Section 8 we discuss the feasibility of our proposed modification.

5.2 Operating System Extension

Access restrictions to the isolated pages are enforced by the hardware, without any involvement from the kernel. However, the isolated pages need to be marked as such in the PTEs, which are located in kernel memory. To support this, we modified a recent version of the Linux kernel. Specifically, we modified the default kernel for the Ubuntu 16.04 LTS distribution which is 4.10 at the time of writing. Similarly to W⊕X, we use page permissions to represent this information. Processes can request the kernel to mark a page as an isolated page by using the existing `mprotect` system call, which is already used to manage the existing memory access permissions: `PROT_READ`, `PROT_WRITE`, and `PROT_EXEC`. For IMIX, we add a dedicated `PROT_IMIX` boolean flag. The implementation of `mprotect` sets permission bits in the PTE according to the supplied protection modes. Note that once a page is marked as `PROT_IMIX` the only way to remove this flag from a page is by unmapping it first which will also set the memory to zero.

5.3 Compiler Extension

To provide C/C++ support for IMIX, we modify the LLVM compiler framework [40]. We chose LLVM over GCC because the majority of memory-corruption defenses leverage LLVM [16, 57, 66]. We modified the most recent version of LLVM (version 5.0) and ported our changes to LLVM 3.3 which is used by CPI [38].
Our modification mainly concerns the intermediate representation (IR) to provide access to the `smov` instruction to mitigations like CPI [38], and the x86 backend to emit the instruction. Further, we introduced an attribute that can be used to protect a single variable by allocating it in an isolated page, e.g., to protect a cryptographic secret. Next, we explain each modification in detail.

IR Extension. Run-time defenses are usually implemented as LLVM optimization passes that interact with and modify LLVM's intermediate representation. In order to allow those defenses to generate `smov` instructions, we extended the IR instructions set. The IR provides two memory accessors, specifically *load* and *store*, which represent respectively a load instruction from the memory to a temporary register, and a store instruction from a temporary register to the memory. Hence, we created two corresponding IMIX instructions: *sload* and *sstore*, which defense developers can use as a drop-in replacement for their regular counterparts.

LLVM IR instructions are implemented as C++ classes and therefore supports inheritance. We implemented our IR instructions to as subclasses of their regular counterparts in order to reuse the existing translation functionality from LLVM IR to machine code, called *lowering* in LLVM parlance.
To allocate memory in the isolated pages, we implemented an LLVM function that can be called from an optimization pass, which allocates memory at page granularity using `malloc` and immediately sets the IMIX permission using `mprotect`. A reference to the allocated memory is returned so that IMIX IR instructions can access the protected memory.

Attribute Support. Data-only attacks are hard to mitigate in practice. To give developers an efficient way to protect sensitive data like cryptographic keys at source code level, we added a IMIX attribute which can be used to annotate C/C++ variables which should be allocated in isolated pages. All instructions accessing those annotated variables will use the IMIX IR instructions instead of the regular ones. LLVM's `annotate` attribute allows arbitrary annotations to be defined, so we only needed to provide the logic needed to process our attribute. We implemented this as an LLVM optimization pass that replaces regular variable allocations with indexed slots in a IMIX protected safe region (one per compilation module), and changes all accessors accordingly.

Modifications to x86 Back End. In the back end, we added code needed to process *sload* and *sstore* instructions. In LLVM, the process of lowering IR instructions to machine code is two-staged. First, the *FastEmit* mechanism is used. It consists of transformation rules explicitly coded in C++ that are too complex to be processed using regular expressions. These are mainly platform-specific optimizations and workarounds. The mechanism can be used to either generate machine code directly, or to assign a rule that should be applied in the next stage. In the second stage, LLVM applies rule-based lowering using pattern matching. The IR instruction and its operands are matched against string patterns in LLVM's TableGen definitions, which define rules to lower the IR to the platform-specific machine code. We modified both stages of the lowering process, similarly to how *load* and *store* are handled.

5.4 Case Study: CPI

To evaluate the impact of our lightweight memory isolation technique to the performance, we ported Code-Pointer Integrity (CPI) by Kuznetsov et al. [38] to use IMIX. CPI uses a safe region in memory to guarantee integrity of code pointers and prevent code-reuse attacks.

All code pointers, pointers to pointers, and so on, are moved to the safe region, so that memory corruption vulnerabilities cannot be exploited to overwrite them. Return addresses are protected using a shadow stack. In contrast to its x86-32 implementation that leverages segmentation, CPI relies on hiding for x86-64 to protect the safe region. CPI places the safe region at a random address and stores this address in a segment, which is selected using the segment register `%gs`. During compilation, CPI's optimization pass moves every code pointer and additional metadata about bounds to the safe region. In order to access the safe region, CPI provides accessors that use `mov` instructions with a `%gs` segment override, which access the safe region using `%gs` as the base address and an offset. These accessors are provided by a compiler runtime extension which is linked late in compilation process. Evans et al. show that this CPI implementation is vulnerable, since the location of the safe region can be brute-forced [22].

We replaced data hiding with IMIX as the memory isolation technique used to prevent unintended accesses to CPI's safe region (including the shadow stack). First, we changed CPI's memory allocation function to not only allocate the safe region, but also set the IMIX protection flag. Second, we modified the compiler runtime, which provides access to the safe region, to make use of our `smov` instruction. Specifically, we changed the safe region functions to access memory directly via `smov` instructions instead of using register-offset addressing. This increases security of CPI dramatically. Since IMIX provides deterministic protection of the safe region, we do not need to prevent spilling of the safe region base address (stored in `%gs`), which IMIX makes CPI leakage resilient. Thus, knowing or brute-forcing the memory location brings no benefit any more, and prevents attacks like "Missing the Point(er)" by Evans et al. [22].

6 Security Analysis

The main objective of IMIX is to provide in-process memory isolation for data in order to make it accessible only by trusted code. Hence, the goal of an attacker is to access the isolated data. As IMIX is a hardware extension, an attacker cannot directly bypass it, i.e., use a regular memory access instruction to access the isolated memory. Thus, the attacker relies on creating or reusing *trusted code*, or manipulating the *data flow* to pass malicious values to the trusted code, or access to the configuration interface of IMIX.

Attacks on Trusted Code. As mentioned in our adversary model, IMIX assumes mitigations preventing the attacker from injecting new code [3], or reusing existing code [7, 50, 52, 54]. This prevents attackers from injecting `smov` instructions that are able to access the isolated data, or reusing trusted code with unchecked arguments, or exploiting unaligned instructions. This assumption is fulfilled by existing mitigations: the strict enforcement of W⊕X [44, 48] prevents the attacker from marking data as code, or changing existing code. Mitigations, such as Control-flow Integrity (CFI) [1, 45, 59] and Code-Pointer Integrity (CPI) [38] prevent the attacker from reusing trusted code.

Attacks on Data Flow. In general, attacks on the data flow [12, 19, 23, 28, 29] are hard to prevent since it would require the ability to distinguish between benign and malicious input data, which generally depends on the context. Therefore, the trusted code must either ensure that its input data originates from isolated pages protected by IMIX, or sanitize the data before using it. The former can be ensured by using the `smov` instruction to access the input data as IMIX's design ensures that the `smov` instruction cannot access unprotected memory. The latter heavily depends on the ability of the defense developer to correctly block inputs that would allow the attacker to manipulate the data within the protected memory in a malicious way: IMIX merely provides a primitive to isolate security critical data. Hence, if the developer fails to sanitize the input data in the trusted code, the code is vulnerable to data-flow attacks independently of whether it leverages IMIX or not. In practice, however, sanitizing inputs correctly requires limited complexity, e.g., in the case of a shadow stack [18] or CPI's safe region [38].

Attacks on Configuration. A common way to *bypass* mitigations is to disable them. For example, to bypass W⊕X, real-world exploits leverage code-reuse attacks to invoke a system call to mark a data buffer as code before executing it.

There are two ways for an attacker to re-configure IMIX: 1) leveraging the interface of the operating system to change memory permissions, or 2) manipulating page table entries.

For the first case, we assume that the attacker is able to manipulate the arguments of a benign system call to change memory permissions (`mprotect()` on Linux). Our design of IMIX's operating system support prevents the attacker from re-mapping protected memory to unprotected memory. Further, before IMIX memory is unmapped, the kernel sets the memory to zero to avoid any form of information disclosure attacks. Similarly, the kernel initializes memory, which is re-mapped as IMIX memory, with zeros to prevent the attacker from initializing memory with malicious values, mapping it as IMIX

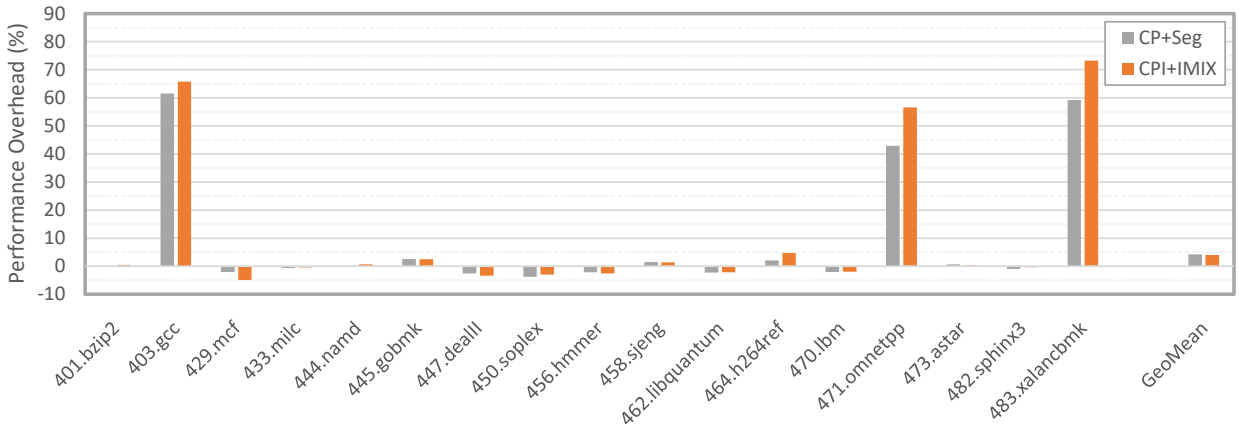

Figure 3: Performance overhead of CPI with segmentation-based memory hiding, and with IMIX.

memory, and then passing it to trusted code. Therefore, the developer must be aware that the attacker is potentially able to pass a pointer into a *zero-filled* page as an input value to trusted code.

For the second case, we assume that the attacker is able to exploit a memory-corruption vulnerability in the kernel. While the focus of this paper is on how user-mode defenses can leverage IMIX, our design allows kernel-based defenses to leverage IMIX as well. Hence, to mitigate data-only attacks against the page table [19] existing defenses [17, 25] can leverage IMIX to ensure that only trusted code can access the page tables.

7 Performance Evaluation

To evaluate the performance of our approach, we ported the original implementation of CPI by Kuznetsov et al. [38] to leverage IMIX to isolate the safe region and applied it to the SPEC CPU2006 benchmark suite. Specifically, we executed all C/C++ benchmarks with the reference workload to measure the performance overheads. The SPEC CPU2006 benchmarking suite is comprised of CPU-intensive benchmarks that frequently access memory, and hence, are well suited to evaluate our instrumentation. We performed our evaluation using Ubuntu 14.04 LTS with Linux Kernel version 3.19.0 on an Intel Core i7-6700 CPU in 64-bit mode running at 3.40 GHz with dynamic voltage and frequency scaling disabled, and 32 GB RAM.

Baseline. First, we measured the performance impact of the original CPI implementation, which we obtained from the project website [39]. Despite efforts, we were unable to execute the CPI-instrumented version of *perlbench* and *povray*. Using the geometric mean of positive overheads, we measured a performance overhead of 4.24% (arithmetic mean of 9.05%, Kuznetsov et al. [38] measured an average performance overhead of 8.4%). We measured a maximum overhead of 61.49% (*gcc*), while a maximum of 44.2% (for *omnetpp*) was reported in the original paper.

CPI with IMIX. Next, we evaluated the performance overhead of IMIX. As hardware emulation turned out to be too slow for executing the SPEC CPU2006 benchmarking tests, we instead evaluated IMIX by replacing `smov` instructions with `mov` instructions that access memory directly. We argue that this reflects the actual costs for `smov` instructions, because the IMIX permission check is part of the paging permission check.

During our performance evaluation we made the interesting observation that our IMIX instrumentation comes with a higher overhead than the baseline. In total, we measured a performance overhead of 14.70% for IMIX, which is an increase of 1.94% in comparison to segmentation-based CPI. In addition, we observed a maximum overhead of 73.27%, compared to a maximum of 61.49% for segmentation-based CPI.

We further investigated this counter-intuitive result. First, we verified with the help of a custom micro-benchmarks that the access time to a memory buffer through a segment register is consistently faster than just dereferencing a general purpose register. Interestingly, it makes no difference whether the base address of the segment is set to 0 or the base address of the buffer. Second, we found that the faster access through segment registers is, at least partially, related to the L2 hardware prefetcher: when we disable it, memory accesses through a general purpose register are faster than segment-based accesses (difference in geometric mean is 0.47% in SPEC CPU2006).

Technique	Policy-based Isolation	Hardware Enforced	Fast Interleaved Access	Fails Safe
SFI	✓	×	✓	×
Segmentation	only for x86-32	✓	✓	✓
Memory Hiding	×	×	✓	×
Paging / EPT	only single-threaded applications	✓	×	✓
Intel MPK	✓	✓	✓	✓
Intel SGX	✓	✓	×	✓
Intel MPX	✓	✓	✓	×
Intel CET	only for Shadow Stack	✓	✓	✓
SMOV	✓	✓	✓	✓

Table 1: Comparison of memory-isolation techniques. Legend: *Policy-based Isolation* means that the memory protection itself cannot be bypassed with an arbitrary memory read-write primitive. *Hardware Enforced* is self-explanatory. *Fast Interleaved Access* refers to the ability to alternately access protected and unprotected memory without additional performance impact. *Fails Safe* means that regular (un-instrumented) memory instructions cannot access the protected memory.

CPI with IMIX (Segment-based Addressing). Similarly to a regular `mov` instruction, the IMIX instruction allows to access memory through a segment register. Unsurprisingly, by adjusting our IMIX-based CPI instrumentation to use segment register-based addressing we achieve 0% overhead over CPI. We further compare IMIX to other memory protection approaches, namely Intel MPK and Intel MPX, in Section 9.

8 Discussion

On the Feasibility of Our ISA Extension. One of the main values of any defense in the field of system security is practicality. Therefore, it comes with no surprise that existing research often sacrifices security in favor of performance [45, 53, 67], and retrofit existing hardware features [6, 16, 18, 41, 58, 63] instead of introducing more suitable ones. The reason is that in practice it is unlikely that hardware vendors are going to change their hardware design and risk compatibility issues with legacy software in order to strengthen the security and increase the performance of a specific mitigation.

However, we argue that this does not apply to IMIX for two reasons: 1) IMIX enables strong and efficient in-process isolation of data which is an inevitable requirement of many memory-corruption defenses. 2) IMIX can be implemented by slightly modifying Intel's proposal, Control-flow Enforcement Technology (*CET*) [33].

As we discussed in Section 2, memory-corruption defenses often reduce the attack surface from potentially the whole application's memory to the memory that is used by the defense itself. With IMIX we provide a strong and efficient hardware primitive to enforce the protection of this data which is mitigation-agnostic. By providing a primitive, which is essential to memory-corruption defenses, rather than implementing a specific defense in hardware [33], vendors avoid the risk of a later bypass [50].

We believe that IMIX can be adopted in real world with comparatively low additional effort. With CET [33] Intel provides a specialization of IMIX. Similar to IMIX, CET requires modifications to the TLB, semantic changes to the page table, and the introduction of new instructions. Contrary to IMIX, CET's hardware extension is tailored to isolate the shadow stack of a CFI implementation [45]. As expected, generalizing CET's shadow stack to support arbitrary memory accesses still allows implementation of an isolated shadow stack [18].

9 Related Work

In the following, we discuss techniques that may be used to protect memory against unintended access. Table 1 provides an overview of characteristics of these techniques. We explain each of its aspects in detail, and compare them to IMIX.

Software-based Memory Protection. Software-fault isolation techniques (SFI) [51, 61] allow to create a separate protected memory region. SFI is implemented by instrumenting every memory-access instruction such that the address is masked before the respective instruction is executed. This ensures that the instrumented instruction can only access the designated memory segment, however, this instrumentation also has a significant performance impact. Though SFI instruments every load/store

instruction, invalid memory accesses cannot be detected, but are instead masked to point to unprotected memory [37]. ISboxing [20] leverages instruction prefixes of x86-64 to implicitly mask load and store operations. The instruction prefix determines whether a memory-access instruction uses a 32-bit (default case) or 64-bit address. By ensuring that untrusted code can only use 32-bit addresses to access memory, protected data can be stored in memory that can only be addressed with 64-bit addresses. Yet, this reduces the available address space significantly, and allows linked libraries to access protected memory.

Another way of protecting data against malicious modifications is to enforce data-flow integrity (DFI) [2, 10, 55]. DFI creates a data-flow graph by means of static analysis, which is enforced during run time by instrumenting memory-access instructions. However, the performance overhead of DFI, which e.g. is on average 7% for WIT [2], prevents it from being used to safeguard protection secrets of code-reuse mitigations, since it would further increase the mitigation's performance overhead. IMIX can be used for both protecting sensitive data (like DFI does) and enabling efficient protection of safe regions for control-flow hijacking mitigations.

Retrofitting Existing Memory Protection. Segmentation is a legacy memory-isolation feature on x86-32 that allows to split the memory into isolated segments [61, 65]. For memory accesses, the current privilege level is checked against the segment's required privilege level directly in hardware. On x86-64 segmentation registers still exist but access control is no longer enforced [37]. On the surface, re-enforcing legacy segmentation seems to be an attractive solution, however, IMIX is easier to implement from a hardware perspective: segmentation requires arithmetic operations, IMIX only one check. Moreover, IMIX provides higher flexibility: protected memory does not need to consist of one contiguous memory region. As segmentation registers are rarely used by regular applications any more, they are often used to store base addresses for memory hiding [6, 38, 41]. Indeed, segmentation-based memory hiding comes with no performance overhead, however, unlike IMIX, it does not provide real in-process isolation and is vulnerable to memory-disclosure attacks [22, 26].

Paging can also be used as well to provide in-process isolation by removing read/write permissions from a page when executing untrusted code [5]. However, regularly switching between trusted and untrusted code is expensive because of 1) two added `mprotect()` system calls, and 2) the following invalidation of TLB entries for each of them [60]. Further, this technique is vulnerable to race-condition attacks, i.e., the attacker can access the protected data from a second thread that runs concurrently to the trusted code. IMIX avoids both disadvantages.

A more recent feature introduced with Intel VT-x is Extended Page Tables (EPT) [32] to implement hardware-assisted memory virtualization. EPT provide another layer of indirection for memory accesses that is controlled by the hypervisor but is otherwise conceptually the same as regular paging. Additionally, VT-x introduces an instruction, `vmfunc`, that enables fast switches between EPT mappings. Hence, to isolate memory, the hypervisor maintains two EPT mappings [16] (regular and protected memory) and trusted code invokes the `vmfunc` instruction instead of `mprotect()`). However, this approach suffers from the same disadvantages as the previous approach which relies on regular paging.

Proposed Memory Isolation Mechanisms. There are already several academic proposals for memory isolation. HDFI [56] is a fine-grained data isolation mechanism that uses MMU tagging for RISC-V. However, due to the need of an additional tag table, HDFI needs two accesses per memory operation. Thus, HDFI leverages additional hardware units (like a cache) to lower the performance impact. Still, HDFI relies on complex static analysis for data-flow integrity which does not meet the requirements for modern JIT-compiled code. IMIX supports JIT compilation by building on existing functionality like `mprotect`, furthermore, IMIX does not need any additional static analysis.

CHERI [62] extends a RISC architecture with fine-grained memory isolation using a set of ISA extensions. For this, two compartments are introduced, however, switching costs are comparably high (620 cycles overhead). In addition, CHERI also relies on intensive static analysis unsuitable for JIT code.

ILDI [13] is another data isolation approach, but for ARM. It leverages existing ARM features (`Privileged Access Never`, `PAN`) to create a safe region for sensitive kernel memory, isolated from potential kernel exploits. By explicitly granting `Load and Store Unprivileged` (`LSU`) instructions access to sensitive data, regular accesses (possibly attacker controlled) are no longer allowed to access the safe region. However, ILDI imposes a high performance overhead on the kernel (35.3%). IMIX proposes a general approach that can be leveraged by both kernel-space and user-space mitigations.

Recent Hardware Extensions. Recent Intel CPUs implement a variety of new memory-protection features. In particular, Memory Protection Extensions (MPX) and Memory Protection Keys (MPK) can be retrofitted to enable in-process memory isolation. Nevertheless, as we discuss in the following, they are not viable alternatives

Name	CPI+Seg (%)	CPI+IMIX (%)	CPI+MPK (%)	CPI+MPX (%)
400.perlbench	-	-	-	-
401.bzip2	0.13	0.44	0.19	132.36
403.gcc	61.49	65.73	2856.48	-
429.mcf	-2.08	-4.89	-2.41	203.71
433.milc	-0.63	-0.47	-0.45	-6.36
444.namd	-0.10	0.66	-0.09	-8.60
445.gobmk	2.55	2.52	32.41	-
447.dealII	-2.57	-3.37	-	-
450.soplex	-3.83	-2.96	-0.74	2.88
453.povray	-	-	-	-
456.hmmer	-2.17	-2.54	-1.35	15.43
458.sjeng	1.43	1.36	1.39	56.81
462.libquantum	-2.32	-2.16	-2.62	106.41
464.h264ref	2.04	4.67	536.02	46.87
470.lbm	-2.04	-1.99	-1.94	-9.82
471.omnetpp	42.95	56.62	1444.02	-
473.astar	0.67	0.20	0.70	-1.29
482.sphinx3	-0.99	-0.32	5.52	-0.68
483.xalancbmk	59.23	73.27	1385.67	-
GeoMean	4.24	3.99	12.43	36.86

Table 2: Comparison of memory isolation techniques. *CPI+Seg* uses memory hiding to protect the safe region, for the remaining the respective technique is used. Note that entries marked with "-" crashed with CPI applied.

to IMIX as both come with disadvantages that render them impractical.

The main goal of MPX [31] is to provide hardware-assisted bounds checking to avoid buffer overflows. Therefore, the developer specifies bounds using dedicated registers (each contains a lower and an upper bound) that can be checked by newly introduced instructions. MPX can be retrofitted to enforce memory isolation by defining one bound that divides the address space in two segments: a regular, and a protected region. Then, bounds checks are inserted for every memory access instruction that is not allowed to access protected memory [37]. This has two main disadvantages. First, MPX does not fail safe, i.e., not instrumented instructions (by a third-party library, for example) can still access the safe region. Second, instructions that are allowed to access protected memory can still access unprotected memory. Hence, an attacker might be able to redirect memory accesses of trusted code to attacker-controlled memory. To avoid such attacks, additional instrumentation of the trusted code is required, which significantly increases the performance overhead, as depicted in Table 2. Protecting CPI's safe region with MPX using the open-source implementation by Koning et al. [37] results in a total performance overhead of 36.86% with a maximum of 203.71% for *mcf*, which cannot be considered practical, especially since we were not able to execute the benchmarks that show the highest overheads across all techniques. In comparison, IMIX is secure by default, and enforces strict isolation between protected and unprotected memory without additional overhead.

Intel's MPK is a feature to be available in upcoming Intel x86-64 processors [27, 34], already available on other architectures like IA-64 [30], and ARM32 (called *memory domains*) [4]. Since IMIX and MPK implement a similar idea, we also evaluated MPK based on the approximation given by Koning et al. [37] using the setup we describe in Section 7.

As shown in Table 2, using MPK to protect the CPI safe region results in a total performance overhead of 12.43% with a maximum of 2856.48% for *gcc*. We identified the additional instrumentation to switch between trusted and untrusted code to be the root cause of the additional overhead. This emphasizes the conceptual differences of MPK and IMIX. MPK enables many distinct domains to be present. Reducing these to two possible domains allows IMIX to be leveraged by mitigations like CPI or CFI that rely on frequent domain switches. In contrast, MPK is useful if the application changes domains infrequently, i.e., for temporal memory isolation, or to isolate different threads.

Encryption can also be used to protect memory. For instance, Intel Total Memory Encryption [35] (Secure Memory Encryption for AMD [36]) allows to encrypt the whole memory transparently, protecting it from physical analysis like cold-boot attacks, but not local memory corruption attacks [37]. Another encryption feature, AES-NI [35], reduces overhead associated with encryption

dramatically, which can be used to encrypt and decrypt safe regions as needed. Even with hardware encryption support, solutions like CCFI still induce a performance overhead of up to 52% [42], and keeping the encryption key safe requires relying on unused registers and ensuring that this key is never spilled to memory [14, 37]. IMIX is not prone to register spilling, since it does not rely on a secret to protect memory.

Trusted Execution Environments like Intel SGX [15] offer strong security guarantees through hardware support, but require intensive effort to decouple code to be run in the enclave. SGX can also be used for memory protection, but only at high performance costs due to overheads for entering and exiting the enclave.

10 Conclusion

Mitigations against memory-corruption attacks for modern x86-based computer systems rely on in-process protection of their code and data. Unfortunately, neither current nor planned memory-isolation features of the x86 architecture meet these requirements. As a consequence, many mitigations rely on information hiding via segmentation, on expensive software-based isolation, or on retrofitting memory-isolation features that require compromises in the design of the mitigation.

With IMIX we design a mitigation-agnostic in-process memory-isolation feature for data that targets the x86 architecture. It provides memory-corruption defenses with a well-suited isolation primitive to protect their data. IMIX extends the x86 ISA with an additional memory permission that can be configured through the page table, and a new instruction that can only access memory pages which are isolated through IMIX. We implement a fully-fledged proof of concept of IMIX that leverages Intel's Simulation and Analysis Engine to extend the x86 ISA, and we extend the Linux kernel and the LLVM compiler framework to provide interfaces to IMIX. Further, we enhance Code-pointer Integrity (CPI), an effective defense against code-reuse attacks, using IMIX to protect CPI's safe region.

Our evaluation shows that defenses, like CPI, greatly benefit from IMIX in terms of security without additional performance overhead. We argue that the adoption of IMIX is possible by adjusting the design of Intel's Control-flow Enforcement Technology (CET). Finally, IMIX provides a solution that can serve as a building block for forthcoming defenses to tackle challenging problems, such as data-oriented attacks.

Acknowledgments. This work was supported by the German Science Foundation CRC 1119 CROSSING P3, the German Federal Ministry of Education and Research (BMBF) in the context of HWSec, and the Intel Collaborative Research Institute for Collaborative Autonomous and Resilient Systems (ICRI-CARS).

11 Bibliography

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2005.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *29th IEEE Symposium on Security and Privacy*, S&P, 2008.

[3] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49, 2000.

[4] ARM. ARM architecture reference manual. `http://silver.arm.com/download/ARM_and_AMBA_Architecture/AR150-DA-70000-r0p0-00bet9/DDI0487A_h_armv8_arm.pdf`, 2015.

[5] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2014.

[6] M. Backes and S. Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *23rd USENIX Security Symposium*, USENIX Sec, 2014.

[7] A. Bittau, A. Belay, A. J. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *35th IEEE Symposium on Security and Privacy*, S&P, 2014.

[8] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi. Leakage-resilient layout randomization for mobile devices. In *23rd Annual Network and Distributed System Security Symposium*, NDSS, 2016.

[9] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium*, USENIX Sec, 2015.

[10] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *7th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2006.

[11] N. Chachmon, D. Richins, R. Cohn, M. Christensson, W. Cui, and V. J. Reddi. Simulation and analysis engine for scale-out workloads. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 22:1–22:13, New York, NY, USA, 2016. ACM.

[12] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *14th USENIX Security Symposium*, USENIX Sec, 2005.

[13] Y. Cho, D. Kwon, and Y. Paek. Instruction-level data isolation for the kernel on arm. In *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*, pages 1–6. IEEE, 2017.

[14] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, M. Negro, M. Qunaibit, and A.-R. Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[15] V. Costan and S. Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[16] 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 *36th IEEE Symposium on Security and Privacy*, S&P, 2015.

[17] J. Criswell, N. Dautenhahn, and V. Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *35th IEEE Symposium on Security and Privacy*, S&P, 2014.

[18] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *10th ACM Symposium on Information, Computer and Communications Security*, ASIACCS, 2015.

[19] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi. Pt-rand: Practical mitigation of data-only attacks against page tables. 2017.

[20] L. Deng, Q. Zeng, and Y. Liu. Isboxing: An instruction substitution based data sandboxing for x86 untrusted libraries. In *IFIP International Information Security Conference*, pages 386–400. Springer, 2015.

[21] 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 *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 781–796. IEEE, 2015.

[22] 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 *36th IEEE Symposium on Security and Privacy*, S&P, 2015.

[23] T. Frassetto, D. Gens, C. Liebchen, and A.-R. Sadeghi. Jitguard: Hardening just-in-time compilers with sgx. In *24th ACM Conference on Computer and Communications Security (CCS)*, Nov. 2017.

[24] 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.

[25] X. Ge, H. Vijayakumar, and T. Jaeger. SPROBES: Enforcing kernel code integrity on the trustzone architecture. In *Mobile Security Technologies*, MoST, 2014.

[26] E. Göktaş, R. Gawlik, B. Kollenda, G. Portokalidis, C. Giuffrida, and H. Bos. Undermining information hiding (and what to do about it). In *25th USENIX Security Symposium (USENIX Security 16)*, pages 105–119. USENIX Association, 2016.

[27] D. Hansen. [rfc] x86: Memory protection keys. `https://lwn.net/Articles/643617/`, 2015.

[28] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic generation of data-oriented exploits. In *24th USENIX Security Symposium*, USENIX Sec, 2015.

[29] H. Hu, S. Shinde, A. Sendroiu, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *37th IEEE Symposium on Security and Privacy*, S&P, 2016.

[30] Intel. Intel Itanium architecture developer's manual: Vol. 2. `https://www.intel.de/content/dam/www/public/us/en/documents/manuals/itanium-architecture-software-developer-rev-2-3-vol-2-manual.pdf`, 2010.

[31] Intel. Intel 64 and IA-32 architectures software developer's manual, combined volumes 3A, 3B, and 3C: System programming guide. `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf`, 2013.

[32] Intel. Intel 64 and IA-32 architectures software developer's manual. ch 28, 2015.

[33] Intel. Control-flow Enforcement Technology Preview, 2017.

[34] Intel. Intel 64 and IA-32 architectures software developer's manual. `http://www-ssl.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`, 2017.

[35] Intel. Intel architecture memory encryption technologies specification. `https://software.intel.com/sites/default/files/managed/a5/16/Multi-Key-Total-Memory-Encryption-Spec.pdf`, 2017.

[36] D. Kaplan, J. Powell, and T. Woller. Amd memory encryption. *White paper*, 2016.

[37] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 437–452. ACM, 2017.

[38] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2014.

[39] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. CPI implementation. `http://dslab.epfl.ch/proj/cpi/levee-early-preview-0.2.tgz`, 2014.

[40] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *IEEE/ACM International Symposium on Code Generation and Optimization*, CGO, 2004.

[41] 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 *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[42] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. CCFI: cryptographically enforced control flow integrity. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[43] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *15th USENIX Security Symposium*, USENIX Sec, 2006.

[44] Microsoft. Data Execution Prevention (DEP). `http://support.microsoft.com/kb/875352/EN-US/`, 2006.

[45] Microsoft. Control flow guard. `http://msdn.microsoft.com/en-us/library/Dn919635.aspx`, 2015.

[46] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2009.

[47] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. CETS: compiler enforced temporal safety for C. In *International Symposium on Memory Management*, ISMM, 2010.

[48] OpenBSD. Openbsd 3.3, 2003.

[49] J. Power, M. D. Hill, and D. A. Wood. Supporting x86-64 address translation for 100s of gpu lanes. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 568–578. IEEE, 2014.

[50] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications.

In *36th IEEE Symposium on Security and Privacy*, S&P, 2015.

[51] 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 *18th USENIX Security Symposium*, USENIX Sec, 2010.

[52] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2007.

[53] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2004.

[54] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *34th IEEE Symposium on Security and Privacy*, S&P, 2013.

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

[56] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. Hdfi: hardware-assisted data-flow isolation. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 1–17. IEEE, 2016.

[57] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *34th IEEE Symposium on Security and Privacy*, S&P, 2013.

[58] A. Tang, S. Sethumadhavan, and S. Stolfo. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[59] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *23rd USENIX Security Symposium*, USENIX Sec, 2014.

[60] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal. Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 340–349. IEEE, 2011.

[61] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review*, volume 27, pages 203–216. ACM, 1994.

[62] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, et al. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 20–37. IEEE, 2015.

[63] J. Werner, G. Baltas, R. Dallara, N. Otterness, K. Z. Snow, F. Monrose, and M. Polychronakis. No-execute-after-read: Preventing code disclosure in commodity software. In *11th ACM Symposium on Information, Computer and Communications Security*, ASIACCS, 2016.

[64] Wind River. Simics full system simulator. `https://www.windriver.com/products/simics/`, 2018.

[65] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93. IEEE, 2009.

[66] C. Zhang, W. Zou, T. Wang, Y. Chen, and T. Wei. Using type analysis in compiler to mitigate integer-overflow-to-buffer- overflow threat. *Journal of Computer Security*, 19:1083–1107, 01 2011.

[67] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *22nd USENIX Security Symposium*, USENIX Sec, 2013.