To Isolate, or to Share? That is a Question for Intel SGX

Youren Shen, Yu Chen, Kang Chen, Hongliang Tian and Shoumeng Yan

August 14, 2018

To Isolate, or to Share? That is a Question for Intel SGX

Youren Shen*[†]
Tsinghua University
syr15@mails.tsinghua.edu.cn

Yu Chen
Tsinghua University
yuchen@tsinghua.edu.cn

Kang Chen
Tsinghua University
chenkang@tsinghua.edu.cn

Hongliang Tian*
Intel Corporation
hongliang.tian@intel.com

Shoumeng Yan
Intel Corporation
shoumeng.yan@intel.com

ABSTRACT

One cornerstone of computer security is hardware-based isolation mechanisms, among which an emerging technology named *Intel Software Guard Extensions (SGX)* offers arguably the strongest security on x86 architecture. Intel SGX enables user-level code to create trusted memory regions named *enclaves*, which are isolated from the rest of the system, including privileged system software. This strong isolation of SGX, however, forbids sharing any trusted memory between enclaves, making it difficult to implement any features or techniques that must share code or data between enclaves. This dilemma between isolation and sharing is especially challenging to system software for SGX (e.g., library OSes), to which both properties are highly desirable.

To resolve the tension between isolation and sharing in system software for SGX, especially library OSes, we propose a *single-address-space approach*, which runs all (user-level) processes and the library OS in a single enclave. This single-enclave architecture enables various memory-sharing features or techniques, thus improving both performance and usability. To enforce inter-process isolation and user-privilege isolation inside the enclave, we design a *multi-domain* software fault isolation (SFI) scheme, which is unique in its support for two types of domains: 1) data domains, which enable process isolation, and 2) code domains, which enable shared libraries. Our SFI is implemented efficiently by leveraging Intel Memory Protection Extensions (MPX). Experimental results show an average overhead of 10%, thus demonstrating the practicality of our approach.

KEYWORDS

Intel SGX, Library OS, Software Fault Isolation, Intel MPX

ACM Reference Format:
Youren Shen, Yu Chen, Kang Chen, Hongliang Tian, and Shoumeng Yan. 2018. To Isolate, or to Share? That is a Question for Intel SGX . In *9th Asia-Pacific Workshop on Systems (APSys '18), August 27–28, 2018, Jeju Island, Republic of Korea.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3265723.3265727

*This work was done while the author was an intern at Intel
†Both authors contributed equally to the paper

APSys '18, August 27–28, 2018, Jeju Island, Republic of Korea
2018. ACM ISBN 978-1-4503-6006-7/18/08…$15.00
https://doi.org/10.1145/3265723.3265727

1 INTRODUCTION

One cornerstone of computer security is hardware-based isolation mechanisms, e.g., inter-process isolation and user-privilege isolation. However, as these isolation mechanisms are most commonly managed by privileged system software, e.g., OSes and hypervisors, their security guarantee is considered *weak*. On the one hand, system software is prone to security vulnerabilities, due to the nature of low-level system programming (e.g., over 650 memory-safety bugs are found in Linux kernel 4.0[20]) and the sheer size of the trusted computing base (TCB) (e.g., Linux kernel has over 4 million lines of code[22]). On the other hand, system software, sometimes, cannot be trusted by users, as is the case of public cloud computing, where a malicious administrator of a cloud provider may act against the interest of cloud tenants[1].

To provide a stronger security than existing hardware-based isolation mechanisms, *Intel Software Guard Extensions (SGX)*[15] is introduced to x86 architecture. SGX enables user-level code to create trusted memory regions named*enclaves*, which are isolated from the rest of the system. Enclaves protect both confidentiality and integrity of the code and data within. This security is achieved by the CPU using a combination of memory access control and transparent memory encryption, defending against both software attacks (e.g., a malicious OS kernel or hypervisor) and hardware attacks (e.g., memory bus snooping). In other words, the TCB of SGX only consists of the CPU and the enclave itself, excluding any privileged software. With this strong security guarantee, SGX is considered a promising hardware-based isolation technology, especially for securing user workloads on public clouds.

The strong isolation enforced by enclaves, however, raises challenges for sharing between enclaves; and, there are plenty of good reasons for sharing. Code sharing, most commonly via shared libraries, has long been taken for granted on modern OSes, and it is also desirable for SGX. For example, code sharing can alleviate memory usage pressure on Enclave Page Cache (EPC), which caches a limited number of trusted memory pages (only 128MB on current SGX hardware) and incurs a significant performance penalty (200X) on cache misses[19]. Data sharing between enclaves are also useful for various reasons, e.g., implementing shared buffers, programming concurrency via shared data structures, or passing zero-copy messages. Unfortunately, all these sharing opportunities are difficult, if not impossible, to realize since SGX forbids any access to the trusted memory of an enclave from outside the enclave, or from other enclaves, even those signed by the same developer.

SGX does allow sharing *untrusted* memory between enclaves; yet, ironically, this only chance of memory sharing weakens the isolation of enclaves in an unexpected way. That is, the unlimited access to untrusted memory from code inside enclaves opens up opportunities for a new type of SGX-specific, security vulnerability, which we call *out-of-enclave bugs.* Consider one of the most common types of memory-safety bugs, null-pointer dereferencing. For non-SGX programs, this usually crashes the program immediately; yet, for SGX programs, this may jeopardize the security of an enclave since a privileged attacker could map at address 0 with any memory page (assuming address 0 is outside the enclave, as is the case for any enclaves built with Intel SGX SDK[14]). Without preventing *unintended, out-of-enclave* memory access (address 0 is just a special case), victim enclave code may be manipulated by attackers, causing data leakage or even control hijack (see appendix A for sample code).

From the analysis above, there is clearly a tension between isolation and sharing in SGX; and, this tension is even stronger in system software for SGX. System software, e.g., library OSes, has drawn the interest of many SGX researchers[5, 8, 9, 21, 23] since they enable *unmodified* user applications to be ported into SGX by providing a layer of abstraction. This abstraction layer (e.g., syscalls) is usually implemented by a common runtime (e.g., library OS) that supports running multiple user tasks (e.g., user-level processes). Apparently, this offers opportunities for sharing, yet also demands mechanisms for isolating. However, as will be shown later in this paper, no existing system software for SGX achieves both sharing and isolation.

In this paper, we aim at resolving the tension between isolation and sharing in system software for SGX, especially library OSes. To this end, we propose a *single-address-space* approach to constructing library OSes for SGX, which runs all (user-level) processes and the library OS in the same enclave. This single-enclave architecture, which is radically different from the multi-enclave architecture of existing systems, enables various memory-sharing features or techniques, e.g., shared libraries, shared file systems, efficient inter-process communication (IPC), thus improving both performance and usability.

To overcome the lack of hardware-based isolation mechanism inside an enclave, we design a *multi-domain software fault isolation (SFI)* scheme. SFI[24] is a compiler-based technique that isolates untrusted modules (called *domains*) from the rest of the system. Our multi-domain SFI is unique in its support of two types of domains: 1) *data domains*, which enable software-isolated processes (SIPs), 2) *code domains*, which enable shared libraries. One extra benefit brought by our SFI is that the aforementioned out-of-enclave bugs can be prevented automatically. Furthermore, our SFI is implemented efficiently by leveraging Intel Memory Protection Extensions (MPX)[12], a relatively-new x86 feature that emerges prior to SGX, for both improved performance and reduced complexity.

The main contributions of this paper are three-fold:

1. We make a case for the single-address-space approach to constructing library OSes for SGX by pointing out the shortcomings of existing library OSes and highlighting the benefits of a single-address-space library OS (see section 2).

2. We design a multi-domain SFI scheme that can enable inter-process isolation and user-privilege isolation in a single-address-space library OS. It is unique in supporting both code domains and data domains. It leverages Intel MPX for improved performance and reduced complexity (see section 3).

3. We implement a prototype of our SFI scheme. Benchmark results show an average performance overhead of 10%, thus demonstrating the practicality of our approach (see section 4).

Note that while our discussion focuses on library OSes, we do so for the sake of brevity and concreteness. Most of our arguments and contributions apply equally well to other system software for SGX. Here, by saying system software, we mean a common runtime that supports running multiple user tasks; some examples are language runtime, container runtime, and serverless runtime.

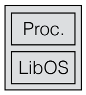
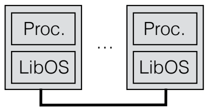
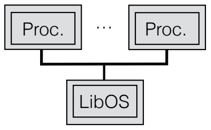
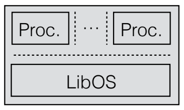
2 SHARING VIA A SINGLE-ADDRESS-SPACE ARCHITECTURE

In this section, we analyze alternative architectures of existing library OSes for SGX and point out how their lack of isolation or sharing impacts usability or performance. Then, we make a case of a single-address-space approach of constructing library OS for SGX by highlighting its benefits.

Since the inception of SGX, building library OSes for SGX[5, 8, 9, 21, 23] has become an important research direction; and it comes with some good reasons. On the one hand, SGX is designed to protect against the potentially malicious host OS, thus all CPU instructions (e.g., `int`, `syscall`, and `sysenter`) that may invoke system calls are forbidden inside enclaves. On the other hand, almost all non-trivial applications rely on system calls. Due to the lack of support for system calls, the C standard library shipped with Intel SGX SDK[14] is stripped of any APIs that must use system calls. Without system calls, existing libraries or applications require enormous refactoring efforts to be ported to SGX. This proves to a great obstacle to developing SGX-protected applications. To solve this lack-of-syscalls problem once and for all, many researchers, including us, believe that it is necessary to introduce library OSes into SGX's enclaves.

Despite the initial success reported by existing library OSes, we find that these library OSes have limitations inherent to their architecture. According to the architecture, existing systems can be classified into three categories (see table 1). The simplest architecture (a) is adopted by the first library OS for SGX, Haven[9], and several later works[5, 8]. However, due to the single-enclave design, these library OSes cannot support multi-process applications. To overcome this critical limitation on usability, Graphene-SGX[23] and PANOLY[21] are designed as architecture (b), in which each process is run in a different enclave. This multi-enclave design requires one instance of library OS running in each enclave, making it difficult to synchronize between multiple instances of the library OS. As a result, a library OS of architecture (b) has difficulty in implementing an unified, encrypted file system to all processes. To overcome this difficulty, the file system proposed in [6] adopts architecture (c), which keeps the library OS and processes each in its own enclave. However, as all data passed between enclaves must be encrypted and decrypted, its system calls and IPC become extremely expensive.

Table 1: Comparison between alternative architectures of library OSes for SGX

	Alternative achitectures			
	(a)	(b)	(c)	(d)
				
Related work	[5, 8, 9]	[21, 23]	[6]	**This Work**
# of processes vs # of enclaves	1 : 1	$N : N$	$N : N + 1$	$N : 1$
Process creation (e.g., `fork`)	No support	Expensive	Expensive	**Cheap**
IPC (e.g., `pipe`)	No support	Expensive	Expensive	**Cheap**
Shared file systems (e.g., `read`)	No support	No support	Expensive	**Cheap**
Shared memory (e.g., `mmap`)	No support	No support	No support	**Cheap**

The shortcomings of the architecture above stem from the fact that they either have no isolation at all inside the enclave, as in architecture (a), which leads to the lack of multi-process support, or use enclaves as natural boundaries for isolation, as in architecture (b) and (c), which requires a secure communication channel, thus resulting in inefficient syscalls or IPC. To overcome the limitations of these existing architectures, we propose a *single-address-space approach* to constructing library OSes for SGX, as shown in architecture (d), which is radically different from traditional approaches in that

(1) it runs multiple (user-level) processes and the library OS inside a single enclave to enable memory-sharing features or techniques (see the remaining of this section); and
(2) it leverages software fault isolation (SFI) to realize inter-process isolation and user-privilege isolation (see the next section).

In other words, our single-address-space approach can achieve both sharing and isolation.

By running all processes and the library OS within the same enclave, a single-address-space library OS can leverage memory sharing to implement system calls more efficiently. Some examples are listed below:

`fork`. Due to the high cost of creating an enclave, `fork` in Graphene-SGX is measured to be nearly 10,000X slower than that that in Linux[23]. In contrast, a single-address-space library OS can create a new process without creating a new enclave; hence, `fork` in a single-address-space library OS becomes significantly cheaper.

`pipe`. Pipes is a common way to transfer a stream of data from one process to another. Traditional library OSes must encrypt the stream and copy the encrypted data to a shared (untrusted) buffer outside enclaves. A single-address-space library OS, however, can allow both producer processes and consumer processes to access the underlying (trusted) buffer of the stream directly, thus no encryption and copying.

`read`. File operations from inside enclaves require authenticated encryption to guard both the confidentiality and the integrity of persistent data. To this end, it is desirable to implement a secure file system shared by all processes because 1) a common cache of plaintext data can be used to reduce the number of encryption or decryption operations, 2) file metadata (e.g., keys) can be shared for efficient concurrent access, and 3) a consistent view of the file system can be provided to all processes. Such a secure file system is hard to be implemented efficiently in traditional library OSes, but not in a single-address-space library OS.

`mmap`. Shared memory mapping can reduce the memory overhead of multiple processes that access the same file, e.g., shared libraries. Yet, this sharing semantic of `mmap` is difficult, if not impossible, to realize in traditional library OSes, as sharing trusted memory between enclaves is forbidden by SGX. It is, however, possible to emulate the sharing semantic of `mmap` (to some degree) in a single-address-space library OS. This brings an immediate benefit of enabling shared libraries (see more in the next section).

On the feasibility of the single-address-space approach, two facts are worth noting. First, executables can be compiled as position-independent executable (`-fPIE` for gcc) to be loaded at any address and this practice has become increasingly common for the purpose of address space layout randomization (ASLR). So, loading multiple executables in the same address space won't be a problem. Second, the size of an enclave (i.e., `ELRANGE`) is a value of 36 bits, which means a maximum of 64GB addressable memory in an enclave. While a size of this magnitude may not seem impressive compared to the whole address space of a process on 64-bit machines, it is actually far bigger than the size of EPC, which is as small as 128MB on current SGX hardware platform. So, in most cases, this 64-GB address range is not a limiting factor for loading multiple processes.

3 ISOLATING VIA A MULTI-DOMAIN SFI SCHEME

This section describes how single-address-space library OSes for SGX can enjoy the benefits of isolation by introducing software isolated processes, which are enabled via software fault isolation.

The x86 architecture provides various hardware-based isolation mechanisms, among which enclaves is arguably the one of the finest granularity: an enclave cannot be further partitioned by hardware means. Segmentation is a well-known hardware feature for isolation inside a process; unfortunately, it is neither available on x86-64

Table 2: The two isolation levels for SIPs in a single-address-space library OS

Isolation levels for SIPs	L1 (Sandboxing stores only)	L2 (Sandboxing stores as well as loads)
Typical use cases	SIPs in a single-tenant enclave	SIPs in a multi-tenant enclave
Security objectives for SIPs		
O1. Inter-process isolation	A SIP cannot be tampered by other SIPs	Plus, a SIP cannot be snooped by other SIPs
O2. User-privilege isolation	The library OS cannot be tampered by SIPs	Plus, the library OS cannot be snooped by SIPs
O3. Out-of-enclave-bug mitigation	SIPs cannot store to memory outside the enclave	Plus, SIPs cannot load from memory outside the enclave
How SIPs use domains		
Data domains	Per-SIP data domains sandbox stores only	Per-SIP data domains sandbox both stores and loads
Code domains	All SIPs share a single code domain	SIPs of the same provenance (e.g., tenant) share a code domain

nor inside enclaves. A less-known feature named Intel Memory Protection Keys (MPK)[13] can divide the address space of a process into memory domains that are isolated between one another. However, MPK is incompatible with the security model of SGX, since it must be configured in page tables, which are managed by the (potentially malicious) host OS.

To overcome the lack of hardware-based isolation mechanisms inside enclaves, we introduce *software-isolated processes (SIPs)* to single-address-space library OSes. Contrary to traditional hardware-isolated processes, SIPs are isolated by software means so that CPU instructions are confined in data access and control flow. With SIPs, a single-address-space library OS can achieve the following security objectives:

O1. Inter-process isolation (protecting a SIP from other SIPs);
O2. User-privilege isolation (protecting the library OS from SIPs);
O3. Out-of-enclave-bug mitigation (preventing SIPs from unintended access to memory outside the enclave).

Note that the traditional architectures, as shown in table 1, cannot achieve O2 or O3: architecture (a) and (b), no O2 or O3; architecture (c), no O3. In this sense, by introducing SIPs, our approach improves the security.

To construct SIPs that achieve the security objectives above, we design a *multi-domain software fault isolation (SFI)* scheme. SFI[24] is a software-instrumentation technique that isolates untrusted modules (called *domains*) from the rest of the system by inserting necessary checks before memory-access and control-flow CPU instructions. Compared with existing SFIs[17, 24, 26–28], our multi-domain SFI is unique in its support of multiple domains of two complementary types: *data domains*, for sandboxing data access (stores to or loads from memory), and *code domains*, for sandboxing control flow. This separation between sandboxing data and code enables multiple SIPs to execute shared code (e.g., shared libraries) while operating on private data.

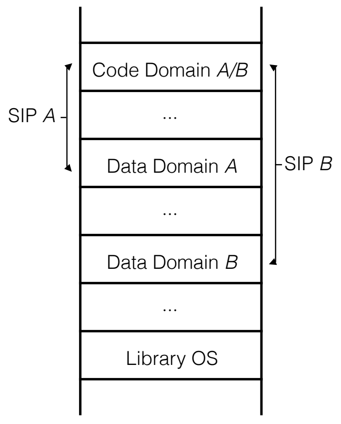
Our multi-domain SFI can be implemented efficiently by leveraging Intel Memory Protection Extensions (MPX)[12], a relatively-new x86 feature that emerges before SGX, which we have repurposed for use in SFI. The original motivation of MPX is to accelerate bound checks for object-level memory safety (e.g., detecting buffer overflows); unfortunately, MPX's current hardware implementation and software support are shown to be immature, incurring a considerable overhead yet offering only limited usability[18]. Although MPX is inadequate to implement an efficient, fine-grained memory safety mechanism, we find it well suited for a coarse-grained one, such as SFI. Our multi-domain SFI is implemented to take advantages of the bound checking instructions and registers provided by MPX, thus improving the performance and reducing the complexity.

In the remaining of this section, we first describe the security model of SIPs and how the security of SIPs can be reduced to that of data and code domains. Then, we elaborate on how to implement data domains and code domains by leveraging MPX.

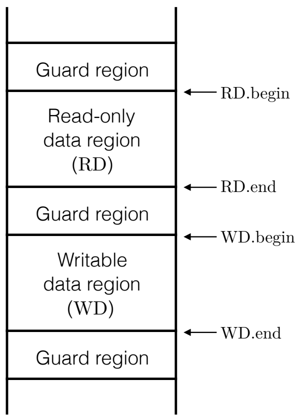
3.1 Security Model of SIPs

SIPs may contain *arbitrary* user code and data. As the code could be faulty or even malicious, the data may be accessed or modified arbitrarily. We assume the code cannot be made writable and the data cannot be made executable (this assumption is to be removed in future work, see section 6). We also assume the library OS itself is implemented correctly and safely to prevent system calls from being abused by SIPs, for example, using `mmap()` to make arbitrary data executable. We assume the program loader of the library OS ensures that all the executables and shared libraries loaded for SIPs are properly instrumented according to our multi-domain SFI scheme (as will be described later). To do this, one can add a binary validator to the program loader, as done in [26].

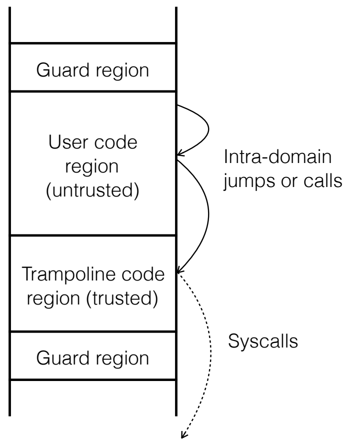
As all SIPs and the library OS reside in the same enclave, our security goal is, certainly, to isolate SIPs; but, one question remains: to what extent? Consider running a single-address-space library OS for SGX to protect user workloads on public clouds. If the containing enclave is owned by a single tenant, then all SIPs inside are most likely to serve this single tenant, and hence the data and code of a SIP are not considered secrets to another SIP. In such a single-tenant enclave, the primary security concern is protecting integrity so that a faulty SIP cannot crash the rest of the enclave. On the other hand, in a multi-tenant enclave, SIPs are probably owned by different tenants; so, protecting the confidentiality of a SIP from other SIPs is necessary. For the reasons above, we define two levels of isolation for SIPs: *L1 for sandboxing stores only*, and *L2 for sandboxing stores as well as loads*. In other words, L1 focuses on integrity, and L2 on confidentiality. Each isolation level achieves the security objectives (O1-O3) to a different degree, as shown in table 2.



(a) An example of two SIPs



(b) A data domain



Guard region
User code region (untrusted)
Trampoline code region (trusted)
Guard region
Intra-domain jumps or calls
Syscalls

(c) A code domain

Figure 1: Software Isolated Processes (SIPs) implemented by a multi-domain Software Fault Isolation (SFI) scheme

The two isolation levels and their corresponding security objectives are realized by isolating SIPs with data domains and code domains, as illustrated in figure 1. SIPs and data domains are mapped one-to-one: a data domain confines its corresponding SIP to access only the data of its own. In contrast, SIPs and code domains are mapped many-to-one: multiple SIPs may share the same code domain, which confines the control flow of these SIPs to the same set of code, including executables and shared libraries. Sharing a code domain between SIPs is useful: for example, a child SIP, after forking from its parent SIP, can reuse the parent's code if they belong to the same code domain.

Now that we have reduced the security of SIPs to that of data domains and code domains, we then describe how data domains and code domains are implemented by leveraging MPX.

3.2 Data Domains

A data domain, as shown in figure 1b, consists of a read-only data region (*RD*) and a read-write data region (*WD*), both of which are surrounded with guard regions of a fixed size (e.g., 4KB). Both of the two data regions are mapped to read-write yet non-executable enclave memory pages. The read-only data region *RD*, which is separated from the writable data region *WD*, is needed, because there are security-critical data that must be modifiable by the trusted program loader or the library OS, but not by the untrusted user code. So the "read-only" of *RD* is only meant for the user code. Guard regions, which are not mapped to any enclave memory pages, trigger hardware exceptions if accessed by any CPU instructions. Guard regions are introduced to facilitate optimization techniques.

A data domain is designed to sandbox data access; more specifically, it adds the following constraints to memory access:

(1) All reads are within the range $[RD.begin - GSize, WD.end + GSize)$, where *GSize* is the size of the guard region.
(2) All writes are within the range $[WD.begin - GSize, WD.end + GSize)$, where *GSize* is the size of the guard region.

To enforce the above constraints, our SFI scheme leverages Intel MPX for bound checking memory addresses. At run-time, upon creating a new data domain or switching to an existing data domain, two of MPX bound registers `bnd0` and `bnd1` are set to the read-only and read-write data regions, respectively, as shown by the following pseudocode:

```
bnd0 ← [$RD.begin, WD.end - 1$]
bnd1 ← [$WD.begin, WD.end - 1$]
```

At compile time, a trusted compiler or binary rewriter inserts MPX bound checks before CPU instructions that read or write memory. For example, a memory write

```
[rcx] ← rax
```

is instrumented as

```
bndcl bnd1, rcx
bndcu bnd1, rcx
[rcx] ← rax
```

where `bndcl` and `bndcu` checks the target address with the lower and upper bounds, respectively.

While it seems expensive to insert bound checks before every memory access, there are several useful techniques to eliminate unnecessary bound checks. First of all, isolation level L1 only protects integrity, thus only memory writes need to be sandboxed, not memory reads. Second, previous works[27, 28] has proposed several effective optimizations, e.g., range analysis and loop analysis, all of which apply to both isolation level L1 and L2. We adopt all of the techniques above to improve the performance.

3.3 Code Domains

A code domain, as shown in figure 1c, consists of two code regions: one for user code and the other for trampoline code. The untrusted user code contains *indirect* control flow instructions (e.g., `call`, `jmp` with runtime-determined destinations, and `ret`) that may jump to arbitrary locations outside the code domain, which must be prevented. Yet, some control flow that tranfers to outside the code domain should be allowed, e.g., system calls to the library OS. To do this, every code domain has special trampoline code to enable the code of this domain to jump to the entry points of the library OS. The trampoline code is assembled by the *trusted* program loader of the library OS, thus no need to be instrumented as user code. So, user code is the subject of the remaining discussion.

A code domain is designed to enforce control flow integrity; more specifically, it must make sure

- The user code cannot jump outside the code domain;
- The user code cannot jump to arbitrary locations of the trampoline code;
- The instrumentation instructions inserted (e.g., the MPX bound checks) cannot be bypassed.

To achieve the control flow integrity above, our SFI scheme introduces a pseudo instruction named `CFI_LABEL`, which has the following properties:

(1) `CFI_LABEL` has a length of 8 bytes;
(2) `CFI_LABEL` has no visible, architectural impact to CPU;
(3) The 4-byte prefix of `CFI_LABEL` does not appear anywhere in the uninstrumented user code or the library OS;
(4) `CFI_LABEL` is unique to a code domain.

To make use of `CFI_LABEL` for control flow integrity, a trusted compiler or binary rewriter makes two kinds of instrumentation. First, before every valid destination of indirect control flow in a code domain, insert a `CFI_LABEL`. Second, before every indirect control flow instruction in a code domain, insert a check that verifies whether the value at the destination address equals to the `CFI_LABEL` of the domain. For example, an indirect call

```
call [rcx]
```

is instrumented as

```
r15 ← -CFI_LABEL
add r15, [rcx]
bndcu bnd2, r15
call [rcx + 8]
CFI_LABEL
```

where `bnd2` has been initialized to a memory range of [0, 0]. So, the MPX bound checking with `bnd2` ensures that `r15` equals to zero, which implies that the value at address `rcx` is a `CFI_LABEL`, thus a valid destination. Note that a `CFI_LABEL` is inserted after the `call` so that an instrumented `ret` can jump back. Other indirect control flow instructions, e.g., `jmp` and `ret`, can be instrumented similarly.

The only remaining question is what exactly is `CFI_LABEL` bitwise. We define `CFI_LABEL` as a special form of multi-byte NOP on x86: the four-byte prefix is `0x0F1F40FF`, and the remaining four byte is the ID of a code domain. The trusted compiler or binary rewriter can ensure that the four-byte prefix is not used by any instructions except `CFI_LABEL`. The library OS assigns a unique ID for each code domain. And when loading code from binaries to text segments in a code domain, the program loader replaces the last four bytes of every `CFI_LABEL` with the actual ID of the target code domain. Thus, `CFI_LABEL`s that appear in the text segments of different code domains have different values.

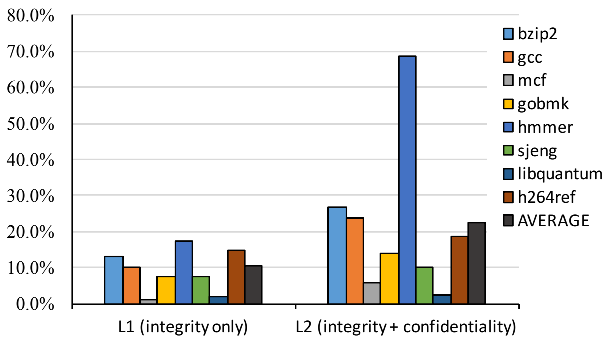


Figure 2: Performance overhead of the multi-domain SFI scheme on SPECint2006

3.4 Interaction with the Library OS

When spawning a new SIP to run an executable (`fork` + `exec`), the library OS needs to do extra work to support the SFI scheme. First, the library OS creates a new data domain so that the SIP owns a private copy of its data. Second, depending upon some configurable security policy, the library OS either assigns an existing code domain to the SIP so that it shares code with other existing SIPs or creates a new code domain for the SIP so that it can own an exclusive copy of code. Now that the data and code domains for the new SIP are prepared, MPX registers can be initialized accordingly.

Context switching between SIPs is essentially context switching between SGX threads since SIPs are executed as SGX threads. As SGX automatically saves and restores the CPU states (including MPX registers) upon context switches between SGX threads, the library OS does not need to explicitly take care of saving and restoring MPX registers upon context switches between SIPs.

SIPs make system calls via the special trampoline code assembled by the trusted program loader. As `CFI_LABEL`s do not appear anywhere inside the library OS, SIPs cannot execute any code inside the library OS directly. The only way to enter the library OS is through the trampoline code, which jumps to a predefined entry point inside the library OS for system calls. The mechanism of our trampoline code is in a sense similar to that of Procedure Linkage Table (PLT) in Executable and Linkable Format (ELF), albeit simpler.

4 IMPLEMENTATION AND EVALUATION

The core of our single-address-space approach to constructing library OSes for SGX is the multi-domain SFI scheme, for which we have implemented a prototype. Our prototype follows the implementation framework for SFI proposed in [27], which instruments at the intermediate-representation (IR) level of LLVM[3]. Furthermore, we incorporate several optimizations proposed in [27] to improve its performance. Our implementation is built with LLVM 7.0, and the resulting LLVM passes for instrumentation and optimizations can then be used by Clang[2] to build C/C++ programs.

To evaluate the performance, we build and run SPECint2006[10] benchmark with and without the instrumention of the multi-domain SFI. We choose to build the benchmark as non-SGX programs, because 1) the multi-domain SFI works equally well both inside or outside enclaves, and 2) we want to highlight the overhead of the

SFI, not a library OS or SGX. The benchmark programs are compiled with Clang 7.0 and tested on a Ubuntu 16.04 machine with an Intel i7 CPU at 3.5GHz and 32GB of RAM.

Figure 2 presents our preliminary results on the performance overhead for SPECint2006 programs. The average overhead is 10.7% and 22.4%, for isolation level L1 and L2, respectively. We believe L1's small overhead is pratical for most use cases and L2's moderate overhead is acceptable for use cases where enhanced isolation is needed.

5 RELATED WORK

Software Fault Isolation (SFI). Our multi-domain SFI scheme differs with existing SFIs[17, 24, 26–28] in several ways. First, our SFI is flexible, no constraints on the number, the size or the position of domains. In contrast, the mostly studied SFI, NaCl[26], supports only one domain, limits the domain to a 4GB address space, and requires it to be guarded by unmapped memory regions of 40GB, both below and above—all these constraints are unacceptable for our purpose. Second, our SFI provides two types of domains to sandbox memory access and control flow, respectively; previous SFIs have no such separation, making it impossible to share code across multiple domains. Third, our SFI leverages Intel MPX for improved performance and reduced complexity, which is partially inspired by [16].

Software Isolated Processes (SIPs). SIPs is a concept pioneered in Singularity[11] operating system. Unlike traditional OSes that isolates processes by hardware mechanism, Singularity enforces isolation between SIPs by language safety. While offering a low-cost isolation, Singularity's SIPs require the code to be written in a safe programming language Sing#. We believe this lock-in to a specific, safe programming language (plus its usage of a costly garbage collector) attributes largely to why Singularity-style SIPs have never made into mainstream operating systems. Our SIPs are different: they are isolated via SFI, thus offering the freedom of developing in any programming language, especially unsafe languages like C/C++.

6 CONCLUSION AND FUTURE WORK

In this paper, we aim at resolving the tension between isolation and sharing in system software for SGX, especially library OSes. To this end, we propose a single-address-space approach that supports software isolated processes (SIPs) via a multi-domain software fault isolation (SFI) scheme. This approach can achieve both isolation and sharing. Preliminary results on our SFI scheme suggest that our single-address-space approach is practical.

To fully realize our vision in this paper, we are working on a fully-fledged, single-address-space library OS that supports SFI-enabled SIPs. Compared with existing library OSes, the most unconventional part of our implementation is the program loader, which adds most of the run-time support for the SFI scheme. For memory safety, we choose to write the library OS in Rust programming language[4].

In addition to the library OS itself, we are also improving the SFI implementation. One top priority is to write a verifier (similar to [28]) to remove the compiler from the TCB. Another interesting improvement to our SFI is to support Just-In-Time (JIT) compilation and self-modifying code, which can be done in a way similar to [7]. This is important to support language runtimes for high-level programming languages, e.g., Java and JavaScript.

A SAMPLE CODE OF OUT-OF-ENCLAVE BUGS

The minimal sample code of out-of-enclave bugs, as shown in listing 1, is essentially an enclave version of null-dereferencing bugs. Low memory addresses (e.g., those below 64KB) are usually not usable in processes on Linux, thereby outside enclaves. But a privileged attacker (whom SGX promises to defend against) can map the low memory range with any malicious data after setting `/proc/sys/vm/mmap_min_addr` to 0. Thus, unintended access to memory page 0, as demonstrated in the sample code, may lead to data leakage or control hijack.

More generally, a sophisticated, privileged attacker can exploit any unintended, out-of-enclave memory access that triggers a page fault. This can be done by capturing the page fault in a fault handler, which then maps malicious data on the faulting page. The faulting page can be easily determined since SGX only hides the low-twelve bits of the faulting address, thus revealing the page number. This property of SGX is also exploited by controlled-channel attacks[25].

The use of SFI inside enclaves, as proposed in this paper, can prevent out-of-enclave memory writes/reads, thus mitigating the security risks of out-of-enclave bugs.

```c
// file: enclave_with_out_of_enclave_bugs.c
#include <string.h>
#include <sgx_trts.h>  // Import APIs from Intel SGX SDK
#include "enclave_t.h" // Import ECall/OCall declaration

// A struct for storing and processing secrets
struct secret_struct
{
    void (*func)(const char* str);
    char data[64];
};

// This pointer should have been initialized somewhere but is not
struct secret_struct* secret; // = NULL, as it is not initialized
// Note: most commonly, address 0 is not inside an enclave

// A buggy ECall that consists of two out-of-enclave bugs
void ecall_process_secret_sloppily(void) {
    // Bug 1: unintented, out-of-enclave memory writes
    char secret_bytes[64];
    sgx_read_rand((unsigned char*)secret_bytes, 64);
    memcpy(&secret->data[0], secret_bytes, 64); // => data leakage!!!

    // Bug 2: unintented, out-of-enclave memory reads
    const char* secret_str = "Top Secret!!!\n";
    secret->func(secret_str); // => control hijhack!!!
}
```

Listing 1: Sample code of out-of-enclave bugs

ACKNOWLEDGMENTS

We want to thank Junjie Mao and Bo Zhang at Intel Corporation for their valuable help and early feedback. We would also like to thank the anonymous reviewers for their useful comments. In addition, we are grateful to our shepherd Sang Kil Cha, who is very kind to give constructive feedback and offer concrete suggestions.

This work was supported by National Natural Science Foundation of China (Grant No. 61772303).

REFERENCES

[1] 2010. GCreep: Google Engineer Stalked Teens, Spied on Chats (Updated). http://gawker.com/5637234/creep-google-engineer-stalked-teens-spied-on-chats

[2] 2018. clang: a C language family frontend for LLVM. http://clang.llvm.org/

[3] 2018. The LLVM Compiler Infrastructure. http://llvm.org/

[4] 2018. The Rust Programming Language. https://www.rust-lang.org/

[5] 2018. SGX-LKL Library OS for Running Java Applications in Intel SGX Enclaves. https://github.com/lsds/sgx-lkl

[6] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. 2018. OBLIVIATE: A Data Oblivious File System for Intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February, 2018*.

[7] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff L. Biffle, and Bennet Yee. 2011. Language-independent Sandboxing of Just-in-time Compilation and Self-modifying Code. *SIGPLAN Not.* 46, 6 (June 2011), 355–366. https://doi.org/10.1145/1993316.1993540

[8] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, André Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Peter R. Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. 689–703. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

[9] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. 2015. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst.* 33, 3 (2015), 8:1–8:26. https://doi.org/10.1145/2799647

[10] Standard Performance Evaluation Corporation. 2006. CINT2006 (Integer Component of SPEC CPU2006). https://www.spec.org/cpu2006/CINT2006/

[11] Galen C. Hunt and James R. Larus. 2007. Singularity: rethinking the software stack. *Operating Systems Review* 41, 2 (2007), 37–49. https://doi.org/10.1145/1243418.1243424

[12] Intel. 2013. Introduction to Intel(R) Memory Protection Extensions. https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions

[13] Intel. 2016. Intel(R) Memory Protection Extensions Enabling Guide. https://software.intel.com/en-us/articles/intel-memory-protection-extensions-enabling-guide

[14] Intel. 2018. Intel SGX for Linux. https://github.com/intel/linux-sgx

[15] Intel. 2018. Intel(R) Software Guard Extensions SDK. Retrieved May 27, 2018 from https://software.intel.com/en-us/sgx-sdk/documentation

[16] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*. 437–452. https://doi.org/10.1145/3064176.3064217

[17] Stephen McCamant and Greg Morrisett. 2005. *Efficient, Verifiable Binary Sandboxing for a CISC Architecture.* MIT CSAIL Technical Report TR-2005-030. Massachusetts Institute of Technology, Cambridge, MA.

[18] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches. *CoRR* abs/1702.00719 (2017). arXiv:1702.00719 http://arxiv.org/abs/1702.00719

[19] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*. 238–253. https://doi.org/10.1145/3064176.3064219

[20] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. 2018. Memory Tagging and how it improves C/C++ memory safety. *CoRR* abs/1802.09517 (2018). arXiv:1802.09517 http://arxiv.org/abs/1802.09517

[21] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*.

[22] Linus Torvalds. 2018. Linux 4.17-rc1. http://lkml.iu.edu/hypermail/linux/kernel/1804.1/06654.html

[23] Chia-che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*. 645–658. https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai

[24] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles, SOSP 1993, The Grove Park Inn and Country Club, Asheville, North Carolina, USA, December 5-8, 1993*. 203–216. https://doi.org/10.1145/168619.168635

[25] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 640–656. https://doi.org/10.1109/SP.2015.45

[26] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2010. Native Client: a sandbox for portable, untrusted x86 native code. *Commun. ACM* 53, 1 (2010), 91–99. https://doi.org/10.1145/1629175.1629203

[27] Bin Zeng, Gang Tan, and Úlfar Erlingsson. 2013. Strato: A Retargetable Framework for Low-Level Inlined-Reference Monitors. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*. 369–382. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/zeng

[28] Bin Zeng, Gang Tan, and Greg Morrisett. 2011. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*. 29–40. https://doi.org/10.1145/2046707.2046713