SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs

Jaebaek Seo[*§] , Byoungyoung Lee[†§], Seongmin Kim[*], Ming-Wei Shih[‡], Insik Shin[*], Dongsu Han[*], Taesoo Kim[‡]
[*]*KAIST* [†]*Purdue University* [‡]*Georgia Institute of Technology*
{jaebaek, dallas1004, ishin, dongsu_han}@kaist.ac.kr, blee@purdue.edu, {mingwei.shih, taesoo}@gatech.edu

***Abstract*—Traditional execution environments deploy Address Space Layout Randomization (ASLR) to defend against memory corruption attacks. However, Intel Software Guard Extension (SGX), a new trusted execution environment designed to serve security-critical applications on the cloud, lacks such an effective, well-studied feature. In fact, we find that applying ASLR to SGX programs raises non-trivial issues beyond simple engineering for a number of reasons: 1) SGX is designed to defeat a stronger adversary than the traditional model, which requires the address space layout to be hidden from the kernel; 2) the limited memory uses in SGX programs present a new challenge in providing a sufficient degree of entropy; 3) remote attestation conflicts with the dynamic relocation required for ASLR; and 4) the SGX specification relies on known and fixed addresses for key data structures that cannot be randomized.**

This paper presents SGX-Shield, a new ASLR scheme designed for SGX environments. SGX-Shield is built on a secure in-enclave loader to secretly bootstrap the memory space layout with a finer-grained randomization. To be compatible with SGX hardware (e.g., remote attestation, fixed addresses), SGX-Shield is designed with a software-based data execution protection mechanism through an LLVM-based compiler. We implement SGX-Shield and thoroughly evaluate it on real SGX hardware. It shows a high degree of randomness in memory layouts and stops memory corruption attacks with a high probability. SGX-Shield shows 7.61% performance overhead in running common micro-benchmarks and 2.25% overhead in running a more realistic workload of an HTTPS server.

I. Introduction

Hardware-based security solutions, such as trusted execution environments, are gaining popularity in today's market. Intel SGX is one of such mechanisms readily available in commodity Intel CPUs since the Skylake microarchitecture. It guarantees confidentiality and integrity of applications, even if their underlying components are compromised. More specifically, SGX provides an isolated execution that protects an application running inside a secure container, called an enclave, against potentially malicious system software, including the operating system and hypervisor. It also offers hardware-based measurement, attestation, and enclave page access control to verify the integrity of its application code.

Unfortunately, we observe that two properties, namely, confidentiality and integrity, do not guarantee the actual security of SGX programs, especially when traditional memory corruption vulnerabilities, such as buffer overflow, exist inside SGX programs. Worse yet, many existing SGX-based systems tend to have a large code base: an entire operating system as library in Haven [12] and a default runtime library in SDKs for Intel SGX [28, 29]. Further, they are mostly written in unsafe programming languages (e.g., C and C++) or often in an assembly language to provide direct compatibility with the Intel SGX hardware and to support its instruction sets. Running such a large code base inside an enclave altogether simply makes SGX programs vulnerable to traditional memory corruption attacks, facing the same security challenges as typical computer environments. This not only nullifies the security guarantee that SGX claims to provide, but also, perhaps more critically, allows attackers to exploit isolation and confidentiality to lurk—there is no way to know what the compromised enclave runs, and even worse, no way to analyze or monitor its execution behavior. For example, by exploiting a stack overflow vulnerability in a trusted web server or database server running in an enclave, an adversarial client can launch traditional return-oriented-programming (ROP) attacks [42, 49] to disclose security-sensitive data in an enclave, which violates the confidentiality guarantee of SGX, yet avoiding any runtime analysis or monitoring thanks to its isolation guarantee.

To defeat such attacks in modern computing systems, many modern defense mechanisms (e.g., stack canary [20], DEP [40], CFI [7], etc) have been proposed, implemented, and deployed recently to significantly raise the bar for exploitation in practice. Address space layout randomization (ASLR) is one of the most comprehensive, yet solid defense schemes proven to be effective in the field. In particular, when properly implemented, ASLR can provide a statistical guarantee of preventing all attackers' attempts. Since ASLR hides the memory layouts from adversaries by randomly placing code and data in runtime, it forces the attackers to guess where the victim code or data is located in order to launch control-flow hijack or data-flow manipulation attacks. This probabilistic defense mechanism has demonstrated its effectiveness in thwarting countless exploitation attempts, and now it is a de-facto security solution in today's modern operating systems, including mobile and server environments.

For this reason, Intel also acknowledges the need for ASLR in the SGX environment and includes a simple ASLR

[§]This work is done while these authors were visiting and Ph.D. students in Georgia Institute of Technology.

NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA
Copyright 2017 Internet Society, ISBN 1-1891562-46-0
http://dx.doi.org/10.14722/ndss.2017.23xxx

scheme for SGX in Intel SGX SDKs for Linux and Windows. However, we find that Intel's ASLR design has several critical limitations that invalidate the security guarantees of ASLR (e.g., a whole memory layout is completely known to an adversary, the malicious operating system). We emphasize that these limitations are not implementation issues that can be fixed easily, but originate from fundamental design challenges that result in conflicts between SGX and ASLR.

This paper uncovers four key challenges in securely deploying ASLR for SGX:

- The strong, unique attack model of SGX exposes the enclave memory layout to untrusted system software, leaving SGX programs completely unprotected by ASLR. By design, SGX delegates page mapping managements to untrusted system software, and thus leaks the information of virtual memory mapping to the underlying software. Note that this was never a security issue in non-SGX computing environments where the system software always serves as the trust computing base of user processes.
- SGX provides the limited memory to an enclave; typically 64 MB or 128 MB in total can be supported [29]. Thus, ASLR for SGX cannot fully utilize virtual address space, significantly limiting the degree of randomness and the security of ASLR.
- ASLR requires a dynamic relocation scheme that updates relative addresses in the code and data section, which conflict with the attestation process of SGX; specifically, SGX finalizes the integrity measurement before an enclave execution starts, but the relocation for ASLR must be performed afterwards. This inherent design disagreement results in writable code pages, nullifying another fundamental hardening technique, executable space protection.
- The SGX specification forces the use of a fixed address for some security-critical data in an enclave. For security reasons, SGX makes several data structures within an enclave immutable, exposing such data structures abused for bypassing ASLR.

To address these issues, this paper proposes SGX-Shield, a new ASLR scheme for SGX programs. It introduces the concept of a multistage loader, which pushes back all ASLR-related operations to its secure in-enclave loader, hiding all security-sensitive operations from adversaries. To maximize the degree of randomness of memory layouts, SGX-Shield employs fine-grained randomization by splitting the code into a set of randomization units. SGX-Shield also enforces a software data execution protection (DEP) to guarantee W$\oplus$X (i.e., Write XOR Execute) in enclave's code pages and isolates security-sensitive data structures from adversaries.

We have implemented a prototype of SGX-Shield on Intel SGX running on both Linux and Windows and evaluated its security properties and performance overhead. We also verify that the SGX programs protected with SGX-Shield have a high degree of entropy to thwart memory corruption attacks inside the SGX environment, yet with a reasonable performance overheads: 7.61% on average in the micro-benchmark and 2.25% in the macro-benchmark.

To summarize, this paper makes the following contributions:

Privilege	Type	Instruction	Description
`ring-0`	`EXE`	`ECREATE`	Create an enclave
`ring-0`	`MEM`	`EADD`	Allocate an EPC page to an enclave
`ring-0`	`MEM`	`EEXTEND`	Measure 512 bytes of an EPC page
`ring-0`	`EXE`	`EINIT`	Finalize the enclave initialization
`ring-3`	`EXE`	`EENTER`	Enter to an enclave
`ring-3`	`EXE`	`EEXIT`	Exit from an enclave

TABLE I: Intel SGX instructions. **MEM**: Memory management related; **EXE**: Enclave execution related.

- **New challenges.** We identify fundamental challenges in enabling ASLR for the SGX environment. In particular, we launch ROP attacks to test the effectiveness of the current ASLR implementation in Linux and Windows SDKs for Intel SGX and find that the ASLR is completely ineffective against strong attackers (e.g., untrusted kernel).
- **Defense scheme.** We implement SGX-Shield, a new ASLR implementation for SGX programs that overcomes the fundamental challenges facing the SGX environment. SGX-Shield supports both Linux and Windows environments, and it incorporates a secure in-enclave loader, software DEP, and software fault isolation all together to provide truly secure ASLR in the SGX environment.
- **Evaluation.** We provide a thorough analysis on SGX-Shield; not only do we conduct performance benchmarks on the real Intel SGX hardware, but also we provide the security analysis on our approach.

The rest of this paper is organized as follows. §II provides background information on Intel SGX and ASLR. §III elaborates fundamental challenges in deploying ASLR for the SGX environment. §IV presents a design of SGX-Shield. §V describes implementation details of SGX-Shield, and §VI evaluates security effectiveness and performance overheads of SGX-Shield. §VII discusses impacts of controlled side-channel attacks against SGX-Shield. §VIII describes related work of SGX-Shield, and §IX concludes the paper.

II. Background

Intel SGX. Intel SGX is an extension of the x86 instruction set architecture [39] that allows a user process to instantiate a protected memory region, called an *enclave*, inside its own address space. SGX prevents system components, including the privileged software (e.g., kernel), from accessing the enclave, which guarantees integrity and confidentiality of the enclave. In this subsection, we summarize the enclave setup and the interaction between an enclave and its host program. The related SGX instructions are described in Table I.

***Enclave initialization*:** Since instructions for the enclave initialization must be executed in the `ring-0` mode, the kernel (i.e., SGX device driver) helps a user process initialize the enclave. The enclave initialization can be categorized into the following four procedures: creation, memory allocation, measurement, and finalizing the initialization: (1) `ECREATE` creates an enclave within the address space of a user process. `ECREATE` requires a public key and the signature of the enclave program as a parameter; (2) `EADD` allocates an Enclave Page Cache (EPC), a physical memory to be used for an enclave, and then copies specified memory pages in the host process to the EPC. The size of the total physical EPC memory is predetermined by the BIOS, and the default size is less than 128MB. (3) EEXTEND measures the SHA-256 digest of 512 bytes of an EPC. Multiple EEXTEND can be invoked to measure more EPC memory pages as well. (4) EINIT finalizes the initialization, which lets the CPU verify the integrity of the enclave program using the measurement result and the pair of the public key and signature provided in ECREATE. After EINIT, no EPC page can be added to the enclave and the permissions of EPC pages cannot be changed[1].

While this enclave initialization process indeed guarantees the integrity of an enclave program, the initial contents of an enclave program are completely visible to system components (i.e., the kernel). It is worth noting that the notion of confidentiality in SGX is limited only to the runtime memory contents after the enclave initialization.

***Host-enclave interaction*:** By the design of SGX, an enclave cannot directly invoke system calls in the OS. Instead, using EENTER and EEXIT, each of which allows entrance or exit between an enclave and host execution contexts, the enclave can indirectly invoke system calls. Moreover, because EEXIT allows the enclave to jump into any location of the host, these two instructions are also used to call the host function from the enclave.

EENTER needs a Thread Control Structure (TCS) to specify the entry point of an enclave execution. A TCS is a special EPC, added by EADD with a TCS flag that contains information for a thread execution, such as the entry point, the base addresses of FS/GS segments, the offset of a State Save Area (SSA), etc. An SSA is a buffer used to save the context of a thread (e.g., values of registers) when an interrupt occurs. Since TCS is critical for the security of enclave programs, SGX prohibits an explicit access to the TCS after initialization. We explain the security implication of SSA and TCS in §III.

ASLR. Address space layout randomization (ASLR) is a powerful memory-protection technique, primarily used to guard against memory corruption attacks. Without ASLR, memory corruption vulnerabilities (e.g., buffer overflow) can easily be exploited by attackers to hijack control-flows or manipulate data-flows and execute malicious code. By randomizing the memory layout (e.g., location of executables and data), ASLR makes it hard for attackers to exploit the vulnerabilities because control-flow hijacking or manipulating in-memory data requires understanding the process memory layout. Thus, ASLR provides probabilistic defense. In particular, with ASLR, the operating system randomly places code and data at load time, making it difficult for attackers to infer the location of code and data, which forces attackers to rely on bruteforce attacks on memory layouts.

III. Technical Challenges

In this section, we articulate the technical challenges in designing ASLR on an SGX environment (Figure 1).

C1. Strong adversary. Typical ASLR mechanisms are built on top of an assumption: memory layout is hidden from attackers, which are not valid in an SGX environment. The strong and unique attack model of SGX exposes the process memory layout to untrusted system software, leaving SGX programs vulnerable to traditional exploitation techniques. In initializing enclaves, the *untrusted* kernel should coordinate the launching procedure to allocate and manage system resources, such as EPC pages. From a security standpoint, this initialization procedure relying on an untrusted party seriously weakens the security of ASLR, as page allocations and its virtual address mapping for an enclave are all visible and thus known to attackers.

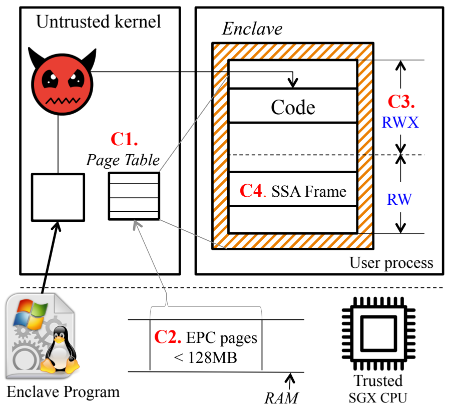

Fig. 1: A workflow of running programs (e.g., executable ELF and PE files) within a SGX enclave. We identify four fundamental challenges in securely performing ASLR in this procedural (i.e., marked from C1 to C4).

Specifically, in order to map a physical address of the EPC region to a virtual address, SGX requires the untrusted kernel's collaboration —the kernel executes EADD, a privileged SGX instruction, with the information on both physical and virtual addresses to be mapped. This design decision is unavoidable and rather natural, as the kernel should be involved in evicting some EPC pages to non-EPC pages if EPC pages are oversubscribed.

However, this results in critical security issues from the ASLR perspective—*the untrusted kernel always knows about the complete memory layout of an enclave application*. Moreover, the base address and the size of an enclave are given to ECREATE as parameters when creating the enclave. Using the base address and the memory layout, the kernel can calculate the exact location of the memory object that does not move during the runtime (e.g., code objects).

This problem is more critical in another popular usage of SGX: hostile cloud environment where people use SGX to securely offload the computation. The current design of Intel SGX always exposes the memory layouts of enclave programs to adversaries, such as cloud providers, where they have full control of underlying software stacks including the kernel, firmware, and all the way down to the SMM program. Under such a strong adversarial model, the greatest care should be taken to design a secure ASLR scheme for SGX. In §VI, we demonstrate that the kernel can succeed in an ROP attack against a vulnerable enclave program with only a single trial in the current Intel SGX SDKs.

C2. Limited memory space. The entropy of the ASLR implementation is inherently limited by the SGX design; SGX has not only a limited memory space overall (i.e., 128 MB

[1] In SGX version 2, there are instructions to add EPC pages and change permissions of EPC pages at the runtime. At the time of writing this paper, there are no available hardware supporting SGX version 2.

EPC [29]), but also the allocated physical memory per enclave is very limited (i.e., an order of 10 MB in typical usages). In these situations, an attacker can easily bruteforce the entire search space to bypass ASLR, as long as they can freely try to mount an exploitation. As such, this limited memory space would significantly reduce the entropy (i.e., the amount of randomness) in enclave programs, compared to what we typically expect in a non-SGX environment: for example, ASLR in x86_64 can utilize the full virtual address space per process (i.e., 48-bit). To address this issue, SGX-Shield takes a fine-grained randomization approach to maximize the randomness of memory layouts.

C3. Writable code pages. Dynamic relocation for ASLR makes it difficult to utilize a powerful, comprehensive defense mechanism against control-hijacking attacks: the No-eXecute (NX) bit [41], which exclusively grants either an executable or writable permission to individual memory pages. This feature is effective in preventing code-injection attacks because attackers cannot directly jump to execute (i.e., executable) the injected code (i.e., writable).

However, there are some situations where both executable and writable flags need to be set at the same time. Just-In-Time (JIT) compilation is a notable exception, as it first writes the compiled code to the memory and executes after that. A typical way to handle this situation is to disable the NX bit for the corresponding memory pages, which are the apparent target for attackers to place the malicious code.

The key feature of SGX, integrity checking (attestation), has a similar problem; it first has to load the code (i.e., writable) and then execute after the measure (i.e., executable). For this reason, the integrity measurement for SGX is only valid for fixed code and data pages, and it cannot be easily extended to support the dynamically changing pages. Specifically, before launching an enclave (i.e., EENTER), the integrity measurement should be finalized and cannot be changed after that. However, implementing ASLR for an enclave program inherently requires changing the permission bit (from writable to non-writable) of code pages after the initial measurement, in particular, the code sections that need to be relocated. Unfortunately, SGX prohibits changing a permission bit after the initial measurement [30]. Therefore, code pages have to be both writable and executable to perform a proper relocation for ASLR after the measurement: the relocation takes place within an enclave after EINIT. We confirm that some Windows enclave programs requiring dynamic relocation contain writable and executable pages. In fact, Intel already acknowledged this issue [27] and further recommended that the enclave code contain no relocation (i.e., no code randomization after the initialization) to enable the NX feature. To properly guarantee the security of ASLR in SGX programs, we need to carefully rethink the design criteria that are compliant with the SGX environment.

C4. Known, fixed addresses. Worse yet, some data structures in SGX do not allow relocation at all. For example, the State Save Area (SSA) frame in SGX does not allow relocation to arbitrary memory addresses; the SSA frame is dedicated to storing the execution context when handling interrupts in SGX. More precisely, the address of SSA Frame is determined by an OSSA field, the offset from the base address of an enclave to SSA, embedded in the Thread Control Structure (TCS). The TCS is initialized and loaded by the kernel through an EADD instruction and contains information for executing an SGX program, such as the entry point of the enclave, the base addresses of FS/GS segments, the offset of SSA, etc. Since TCS is critical for the security of enclave programs, SGX prohibits an explicit access to the TCS after initialization. After initialization, some fields of the TCS might be updated by the CPU during execution (e.g., saving the execution context to the SSA frame in Asynchronous Enclave Exit), and the fields specifying the location of SSA (i.e., OSSA) cannot be updated. In other words, the virtual address of SSA is always known to the untrusted kernel, and the location of SSA cannot be randomized after initialization.

This leaves potential opportunities for abuse. Let's assume two threads T1 and T2 are running concurrently in an enclave. When T1 temporarily exits an enclave due to an interrupt, its execution context, including all register and values, is saved into the SSA. Then, using T2, an attacker can mount an arbitrary memory write to overwrite the field for the instruction pointer in the SSA frame, thereby hijacking control of the T1 thread. Similarly, with an arbitrary memory read, attackers can infer the complete address space layout by following the pointers and instructions from the initial information found in the SSA frame, similar to just-in-time code reuse [57]. In SGX-Shield, we isolate all memory accesses to the known yet security-critical data structures in an SGX program.

IV. Design

In this section, we present the design of SGX-Shield, which fortifies the security aspect of the ASLR scheme in an SGX environment. In particular, we address all challenges highlighted in §III.

- **C1: Strong adversaries.** SGX-Shield introduces the concept of a multistage loader, which can hide ASLR-related security decisions and operations from adversaries (§IV-B).
- **C2: Limited memory.** SGX-Shield employs a form of fine-grained randomization that is tailored to maximize its entropy on the SGX environment (§IV-C).
- **C3: Writable code pages.** SGX-Shield implements a software DEP to enforce W$\oplus$X in an enclave's code pages (§IV-D).
- **C4: Known address space.** SGX-Shield incorporates coarse-grained software-fault isolation (SFI) to protect fixed, security-sensitive data structures from arbitrary memory reads and writes (§IV-E).

For the rest of this section, we start by describing our threat model (§IV-A), and then explain techniques to overcome each challenge in the following subsections: §IV-B shows our multistage loader; §IV-C describes the fine-grained ASLR scheme for SGX; §IV-D explains software DEP; and §IV-E shows our SFI scheme designed for SGX programs. In §IV-F, we introduce performance optimization techniques that we adopt.

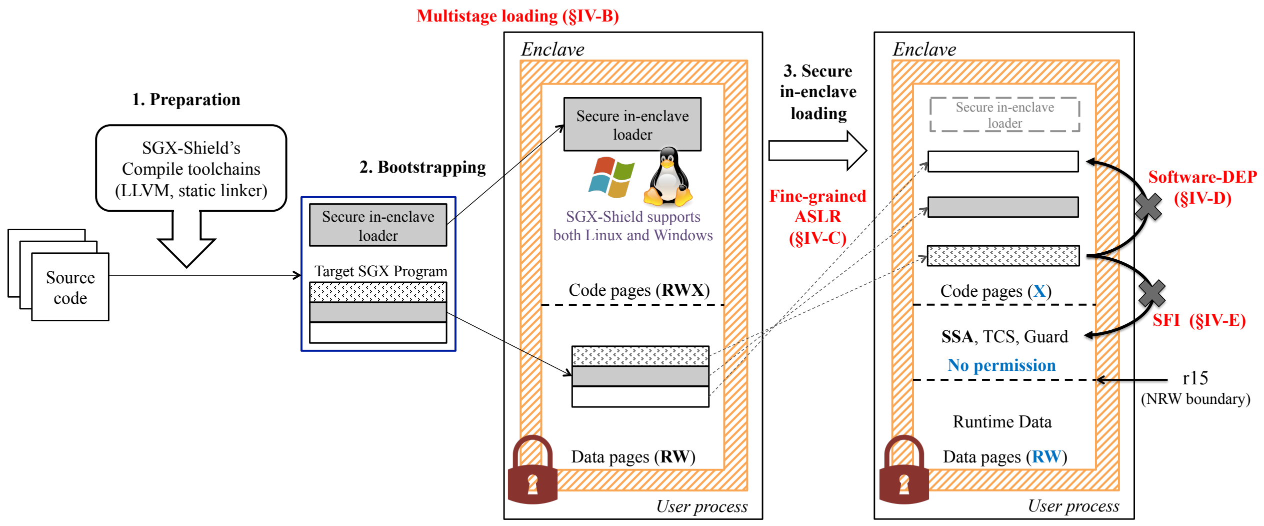

Fig. 2: Overall workflow of SGX-Shield: 1) the preparation phase builds a SGX binary from the target program's source code; 2) the bootstrapping phase loads the secure in-enclave loader into code pages and the target SGX program into data pages; and 3) the secure in-enclave loading phase finally loads the target SGX program.

A. *Threat Model*

SGX-Shield assumes the same attack model as SGX, as our ASLR scheme is designed for SGX programs. Specifically, we assume that only the CPU package with SGX support is trusted and all other hardware components are not. A user runs his or her own target program within an enclave, and all other components in the software stack are not trusted (i.e., other processes, an operating system, and a hypervisor). Our attack model consideration focuses on an attacker who wishes to exploit a vulnerability, a memory corruption vulnerability in particular, in the target program running in the enclave. While completely addressing side-channel issues is not the primary goal of this paper, SGX-Shield provides a barrier to guess the memory layout of an enclave against the attack based on the page fault side-channel (i.e., controlled side-channel) [61]. We discuss the effectiveness of SGX-Shield against the controlled side-channel attacks in §VI-A1 and §VII.

B. *Multistage Loader*

To prevent the untrusted kernel from learning the memory layout of an enclave, SGX-Shield performs all ASLR operations within the enclave, taking advantage of its isolated execution. SGX-Shield consists of three phases, as shown in Figure 2: preparation, bootstrapping, and secure in-enclave loading.

First, the preparation phase builds the target SGX program that a user wants to deploy. This built executable contains a secure in-enclave loader in its code section and the target SGX program in its data section, where the secure in-enclave loader will load the target SGX program later. This phase can be carried out anytime before deployment and does not have to be performed on the same SGX machine in which the target program will be run.

Second, in the bootstrapping phase, SGX-Shield performs the first part of multistage loading. The primary role of the bootstrapping phase is to create an enclave and initialize the secure in-enclave loader with the help of the untrusted kernel. Because the memory layout of an enclave is assumed to be visible to the non-trusted party in this phase, it is designed to make as minimal decisions on resource provisioning as possible and defer all security-sensitive decisions to the secure in-enclave loader. This phase allocates two types of enclave data pages with read and write permissions and code pages with read, write, and execute permissions. The read/write permissions granted to code pages enable the secure in-enclave loader to write the target SGX program into an enclave memory (performing the relocation as well) and then execute it. While this design decision facilitates multistage loading, it ends up having both writable and executable memory pages, similar to the challenge C3 (§III). To address this issue, §IV-D presents how SGX-Shield removes read and write permissions from these pages using a software-level enforcement.

Finally, the secure in-enclave loader loads the target SGX program into the memory space from its data pages. The secure in-enclave loader randomly picks the base address using the `RDRAND` instruction, which relies on the non-deterministic on-processor entropy. Then, it loads each section of the target program, where the address of each section is further adjusted independently at random. Before finishing the loading, SGX-Shield resolves all relocation information, which includes global variables, static variables, and the destination of all branches. As a last step, SGX-Shield wipes out the secure in-enclave loader from the memory space, and then jumps to the entry point of the target SGX program to hand over the execution.

Because the target program is loaded within an enclave by the secure in-enclave loader, SGX-Shield completely hides the address space layout information from the untrusted kernel. The random value is directly obtained from the CPU, and all the following computations and decisions for ASLR of the target program are performed inside the enclave.

It is worth noting that our multistage loading scheme is fully compatible with SGX's attestation scheme. At the moment a measurement for an enclave is finalized by an `EINIT` instruction (i.e., between the bootstrapping and secure in-enclave loading phase), all required resources for SGX-Shield are finalized

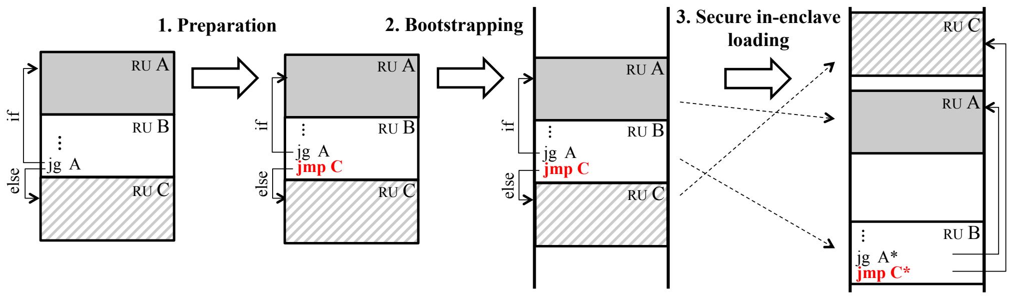

Fig. 3: Fine-grained ASLR scheme based on a randomization unit. `jg A*` and `jmp c*` represent the relocated instructions of `jg A` and `jmp C`, respectively. The preparation phase instruments an unconditional branch (i.e., `jmp C`) next to a conditional branch (i.e., `jg A`). As a result, during a secure in-enclave loading phase, the following unit (i.e., `RU C`) can be randomly placed independently to the location of the instrumented unit (i.e., `RU B`) by resolving relocation.

and fully measured. Thus, from the perspective of performing attestation, SGX-Shield is the same as typical SGX programs—SGX-Shield simply runs code, the secure in-enclave loader, with data, the target SGX program. Specifically, all memory pages for SGX-Shield including the secure in-enclave loader in code pages and the target SGX program in data pages are added to EPC pages and extended for measurement through `EADD` and `EEXTEND`, respectively.

C. Fine-grained Randomization for Enclaves

SGX-Shield employs fine-grained randomization schemes [13, 23, 24, 32, 45, 59] to maximize the ASLR entropy. In the following, we describe how SGX-Shield is designed to randomize the memory space layout across three phases.

Preparation. To enable fine-grained ASLR for code, SGX-Shield relocates code at smaller granularity, called a randomization unit. Randomization units are of fixed size that can be configured. Our implementation supports 32- and 64-byte units. SGX-Shield modifies commodity compilation and linkage procedures because they support only simple module-level (i.e., section-level) randomization. During the compilation, SGX-Shield ensures that the terminating instructions of randomization units are not fall-through cases. This is because fall-through assumes that randomization units are placed consecutively, which is not true when they are relocated for ASLR. Thus, for each fall-through case, SGX-Shield appends an unconditional branch instruction that points to the entry point of the next randomization unit (i.e., the randomization unit pointed to by the fall-through case).

For example, as shown in Figure 3, right after the conditional branch instruction (i.e., `jg A`), an unconditional branch instruction (i.e., `jmp C`) is added, allowing `RU C` to be randomly relocated independently to the location of `RU B`. Note that this instrumentation pass cannot be done naively at the intermediate language (IR) level. Even when IR does not have conditional branch instructions with fall-through features (e.g., LLVM IR), the compiler backend may automatically introduce this. For example, the Intel x86-64 architecture always uses fall-through with conditional branch instructions.

Finally, the size of the randomization unit introduces a trade-off between security and performance. When the size of the randomization is small, there will be more candidate slots to place the randomization unit, increasing the entropy of ASLR at the cost of more frequent branching and decreased spatial locality. We evaluate this trade-off in §VI.

Stage 1: Bootstrapping. We let the loading scheme in the bootstrapping phase over-estimate the memory space required to load the target program, as this size is directly related to the ASLR entropy. Strong adversaries, including the untrusted kernel, always know of ranges of truly active memory space. Thus, unlike traditional ASLR settings where an attacker needs to bruteforce the entire virtual address space, the strong adversary needs to bruteforce only a small space based on her/his prior-knowledge. To this end, we over-estimate both code and data pages, where both are configured as 32 `MB` in the current version of SGX-Shield.

Stage 2: Secure in-enclave loading. Using the target SGX program in data pages, the secure in-enclave loader starts to place each randomization unit into previously allocated memory spaces. SGX-Shield fully utilizes over-estimated memory space, reserved for loading the target program, to randomly scatter each randomization unit, which in turn maximizes the ASLR entropy. SGX-Shield randomizes all data objects as well, which includes stack, heap, and global variables. Specifically, SGX-Shield performs the following steps: (1) for a stack area, SGX-Shield picks the random base address and reserves continuous memory space from this base; (2) for a heap area, it randomly picks k memory pools from the rest of the data pages, where the size of each memory pool is configurable (i.e., 1 `MB` in the current version of SGX-Shield); (3) global and static variables are randomly placed into the rest of the data pages.

Since SGX-Shield randomizes all code and data objects, all references to memory objects including the absolute address and the PC-relative address must be determined after placing them. The secure in-enclave loader conducts the relocation for all memory objects after loading them. For example, as shown in Figure 3, instructions `jg A` and `jmp C` are relocated to correctly point to the shuffled locations for randomization units `A` and `B`, respectively.

Considering controlled side-channel attacks [61], our design also randomizes control-flow dependencies upon data values during the secure in-enclave loading phase. More precisely, the secure in-enclave loader randomizes the order of loading and relocation so that simply observing memory access patterns at page granularity would not leak information on which data or code is being loaded or relocated. Actual runtime behaviors

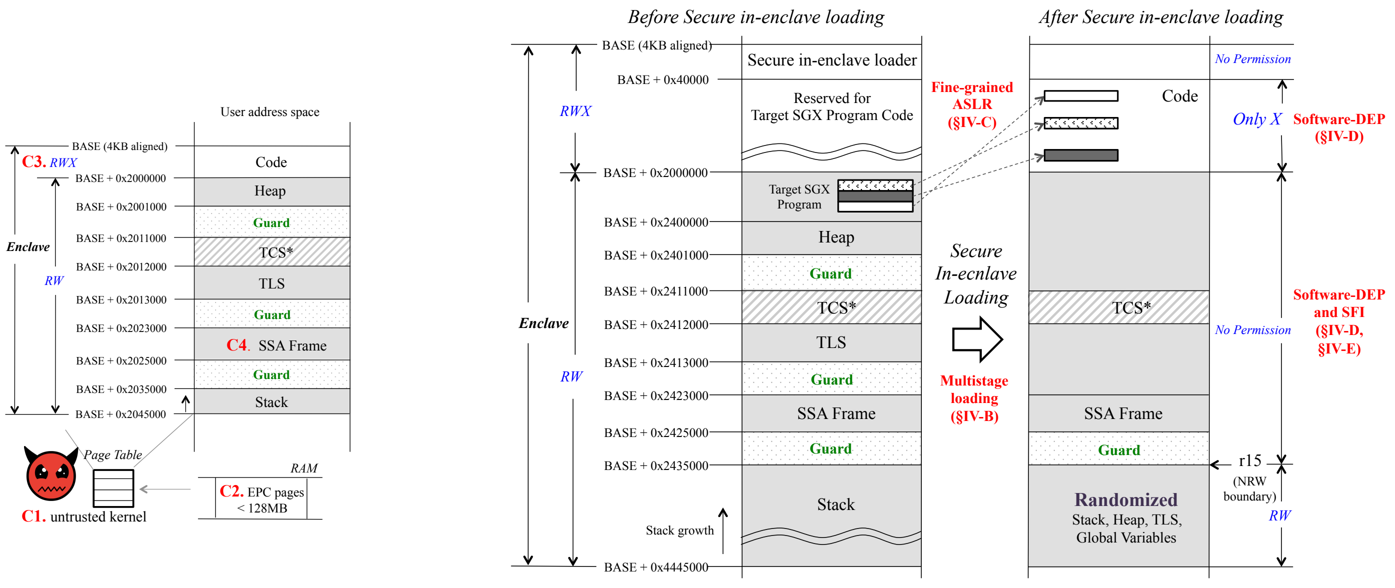

Fig. 4: Runtime memory layouts of an enclave under Intel SGX SDK and SGX-Shield. Both are taken by running Windows 10, but they are similar to those for Linux. After going through secure in-enclave loading phase, SGX-Shield randomizes all code and data pages to maximize the entropy aspects of ASLR as well as implementing software-DEP and SFI.

of the target program might be vulnerable, however, and we describe and evaluate more details of this aspect in §VI-A1 and §VII.

D. Software DEP in Enclaves

As noted in §IV-C, the multistage loading scheme of SGX-Shield leaves code pages both writable and executable. In this subsection, we describe how SGX-Shield removes read and write permissions from code pages by using software-based DEP. Specifically, code pages are granted not only execution but also read and write permissions. As such, SGX-Shield eliminates read and write permissions on code pages once the secure in-enclave loading is finished (i.e., after the target program is randomly mapped to the memory). The key idea behind this is enforcing the *NRW boundary* (i.e., Non Readable and Writable boundary), which is a virtual barrier between code and data pages (See Figure 4). SGX-Shield guarantees this by (1) shepherding all memory access instructions and (2) ensuring only aligned control transfers are made.

Shepherding memory access. In general, there are two types of memory access instructions: (1) explicit memory accessing instructions (e.g., `mov`, `inc`, `add` instructions with memory operands in x86) and (2) stack accessing instructions (e.g., implicit stack pointer adjusting instructions including `push`, `pop`, `call`, `ret`, etc., or explicit stack pointer adjusting instructions including `sub`, `add`, etc. with a stack register operand).

In order to prevent read or write attempts through the first type of instruction, SGX-Shield makes sure that a memory address to be accessed is always higher than the NRW boundary (i.e., the operand should not point to code pages). To avoid extra memory dereferences and thus optimize the performance, SGX-Shield reserves the register `r15` to hold the NRW boundary, which is initialized by the secure in-enclave loader before executing the target program. To minimize the number of instrumented instructions, we transform the original instruction such that it accesses memory using a positive offset from the NRW boundary. We then enforce that the maximum positive offset is smaller than $2^{32} - 1$ to ensure that the instruction never accesses memory beyond the NRW boundary.

```
1  ; Before enforcing non-writable code
2  mov  [rdx+0x10], rax
3
4  ; After enforcing non-writable code
5  ; (r15 is initialized to hold the NRW boundary)
6  ; (enforce rdx >= r15)
7  lea  r13, [rdx+0x10]          ; r13 = rdx+0x10
8  sub  r13, r15                 ; r13 = r15 - r13
9  mov  r13d, r13d               ; r13 = r13 & 0xffffffff
10 mov  [r15 + r13], rax         ; *(r15+r13) = rax
```

Fig. 5: Instrumenting explicit memory access instructions to enforce non-writable code. Since the instruction tries to write to where `rdx` points, SGX-Shield enforces that `rdx` always points to the location higher than the NRW-boundary. `r15` is initialized to hold the NRW boundary value by the secure in-enclave loader. It is assumed that `r13` is an available register (or spilled beforehand) and thus used as a temporary register.

Figure 5 shows how an `mov` instruction that writes to address `rdx+0x10` is instrumented. SGX-Shield enforces that `rdx+0x10` is always higher than `r15`. For this, it first moves the value of `rdx+0x10` to `r13` (line 7), subtracts it from `r15` (line 8), and clears high 32-bits of `r13` (line9). After this point, if `rdx+0x10` $\geq$ `r15`, `r13` will hold the positive offset from the NRW boundary, and the next instruction (line 10) performs the memory write operation as intended. Otherwise, if `rdx+0x10` $\leq$ `r15`, `r13` will hold the `AND`-masked value (line 9) because a subtraction in line 8 results in a negative value (i.e., the most-significant bit is set). Therefore, this offense is properly guarded, as it does not overwrite the code page under SGX-Shield.

To enforce non-writable code pages on stack accessing instructions, SGX-Shield makes sure that a stack pointer (i.e.,

```
; Before enforcing non-writable code
sub  rsp, 0x40

; After enforcing non-writable code
; (r15 is initialized to hold the NRW boundary)
; (enforce rsp >= r15)
sub  rsp, r15                 ; rsp = rsp - r15
sub  rsp, 0x40                ; rsp = rsp - 0x40
mov  esp, esp                 ; rsp = rsp & 0xffffffff
lea  rsp, [rsp + r15]         ; rsp = rsp + r15
```

Fig. 6: Instrumenting stack access instructions to enforce non-writable code. Since a value of `rsp` is changing, SGX-Shield enforces that `rsp` $\geq$ NRW-boundary always holds.

```
; Before enforcing aligned indirected branch
jmp  rax

; After enforcing aligned indirected branch
; (enforce rax % [random unit size] = 0)
and  rax, $-0x20         ; rax = rax && 0xffffffffffffffe0
jmp  rax                 ; jump to the address pointed by rax
```

Fig. 7: Instrumenting indirect branches to enforce aligned jumps. This makes sure that there is no branch to an offset in the middle of bundled instructions, i.e., bypassing the enforcement of the non-writable code.

`rsp`) never points to code pages. To handle instructions that adjust stack pointers implicitly, we simply map a guard page (i.e., no permission is granted) at the top and bottom of the stack area. Because these instructions shift the stack pointer with a small fixed offset and access the stack, the guard page would always be hit if any of them accesses beyond the legitimate stack range. Note, this guarded page scheme on stack instructions optimizes the performance of SGX-Shield. This is because conceptually it replaces a large number of instrumented instructions with two guarded memory pages along with retrofitting the existing exception handling mechanism. In the case of instructions explicitly adjusting stack pointers, SGX-Shield explicitly instruments them, as shown in Figure 6. This is similar to our instrumentation techniques for explicit memory accesses in that both of them compute the positive offset values from the NRW-boundary.

Ensuring aligned control transfer. Because x86 and x86-64 ISA have variable length instructions, code alignment is critical; unexpected instructions can be executed when alignment is broken. This would violate our enforcement on memory accesses, as these instructions would perform memory accesses not guarded by SGX-Shield. SGX-Shield resolves this issue by restricting the control transfers only to the entry point of the randomization unit. As a result, it enforces that there is only one way to decode instructions, ensuring that only shepherded memory access takes place. This enforcement is performed for all control transfer instructions, including indirect branches as well as return instructions. In the case of indirect branches, masking operations are added as shown in Figure 7 so that the destination only points to one of the randomization unit's entry points. In the case of a return instruction, it is first replaced with equivalent instructions, `pop reg` and `jmp reg`, where `reg` can be any available register. Then, the latter `jmp` instruction is instrumented as it is done for indirect branch instructions. Finally, to enable efficient masking on control transfer, our implementation aligns the randomization unit to its size (i.e., if the size of a randomization unit is 32-bytes, an entry point address of a randomization unit ends with five zero bits).

E. *Isolating Access to Security-Critical Data*

By its design, SGX places the page for the State Save Area (SSA) at a known location and does not permit its relocation, as described in C4 (§III). In order to prevent attackers from abusing this non-randomizable data location, SGX-Shield implements software fault isolation (SFI) to isolate SSA. In particular, since we already mark the memory page for SSA as non-executable, we prevent the target enclave program from reading or writing to SSA.

We found that SGX-Shield can easily retrofit its software data execution prevention (DEP) mechanism (§IV-D) to achieve this requirement. Our software DEP mechanism ensures that no read or write accesses are permitted to pages lower than the NRW boundary. Therefore, as shown in §IV-D, we place SSA below the NRW boundary, thereby isolating SSA from being read or written.

F. *Performance Optimization*

A general goal of our optimization is to reduce the number of checks while preserving the same security guarantees, as the checks instrumented by SGX-Shield directly impact runtime performance. In particular, we focus on two types of checks, both of which were identified as major performance bottlenecks during our preliminary performance evaluation: (1) masking operations onto memory read/write instructions for software-DEP (§IV-D) and (2) a jump operation replacing fall-through cases for randomization units (§IV-C).

First, we remove redundant masking operations within a loop. More precisely, we observed that the target address of a memory read or write instruction within a loop either reuses the same address or simply increases through the loop counter, for example, functions manipulating string or buffer (e.g., `memset`, `memcpy`, `memmove`) loops over a buffer using a pointer convoluted with a loop counter. Therefore, we develop a simple loop analysis considering data dependency, which identifies a range of addresses referenced inside. Next, if such a range can be found conservatively, then we replace masking operations inside a loop with two masking operations outside a loop — masking only on the minimum and maximum address value before entering the loop. It is worth noting that this replacement has to be performed in the same randomization unit or more strict control-flow integrity has to be given to this randomization unit. Otherwise, an adversary may jump into the randomization unit, which allows avoiding masking operations before executing memory read/write instructions.

Second, we also minimize the number of fall-through cases if possible. Specifically, we instructed the compiler to avoid emitting jump or switch tables, as we observed that these were a major source of conditional jumps, which results in a huge number of fall-through cases.

V. Implementation

SGX-Shield consists of 23,068 lines of code (see Table II), where 2,753 lines of code contributes to the secure in-enclave loader that is running within an enclave. We implement secure in-enclave loaders (i.e., dynamic loaders) for both Linux and Windows, where the ELF format is used to build an enclave

	# of Files	LoC	Base Framework
Preparation	13	2,304	LLVM Backend
Bootstrapping*	19	1,625	Intel SGX SDK
Secure in-enclave loader*	15	2,753	Intel SGX SDK
Windows version	12	3,514	Intel SGX SDK
Others	71	12,872	-
Total	130	23,068	-

$*$ indicates Linux version of SGX-Shield

TABLE II: The implementation complexity of SGX-Shield. We implement the preparation based on LLVM 4.0. We implement the runtime supports (i.e., bootstrapping and secure in-enclave loader) both in Linux and Windows.

program[2]. Once the enclave program is compiled as an ELF format, we can run it regardless of the platform.

Preparation. The preparation phase includes an LLVM compiler 4.0, a static linker, and a sign tool of Intel SGX SDK for Linux. By modifying the backend of LLVM [6], we insert two kinds of instructions: (1) unconditional jump instructions (instead of fallthrough) at the end of randomization units and (2) instructions to enforce the software-DEP (i.e., masking the target memory address to access). In addition, the LLVM emits each randomization unit as a symbol. The fine-grained symbol information is used in the secure in-enclave loading. The software-DEP currently enforces only the memory write protection. To prevent reading the code, the code page is added as writable and executable, but not readable, through `EADD`. As the relocation does not read the code, non-readable code pages do not cause faults.
The current version of SGX-Shield supports only static linking. We implement a static linker from scratch. While linking relocatables generated by the LLVM, it keeps the fine-grained symbol and relocation information for the fine-grained ASLR. We modify Intel SGX SDK for Linux to provide the enclave program with sufficient code and data pages for shuffling. We embed the binary of enclave program into the binary of secure in-enclave loader as a section using the `objcopy` command. Since the source code for Intel SGX SDK for Windows is not available, we implement a PE editor that adds dummy memory sections to the secure in-enclave loader to provide enough code and data regions.

Bootstrapping. The bootstrapping simply creates an enclave and loads the secure in-enclave loader to the enclave. We implemented a simple program that conducts the bootstrapping in both Linux and Windows.

Secure in-enclave loader. The secure in-enclave loader is a dynamic loader that conducts the randomization unit-level memory object loading and relocations. It resolves the relocation information for all the memory references including the absolute addresses and the PC-relative addresses. We implemented these from scratch and made a best effort to reduce the size of the trusted computing base. In the current version, the core part (i.e., parsing the ELF file, randomly loading, and relocation) is written in a single C file with 384 LoC.

[2] In Windows, the secure in-enclave loader is compiled as a PE format, but it loads the ELF format executable.

```
; gadget #1
pop  rdi         ; src. of memcpy(), the address in enclave to leak
pop  rsi         ; dst. of memcpy(), the address in host
ret              ; jump to gadget2

; gadget #2
pop  rdx         ; len. of memcpy()
ret              ; jump to memcpy()
```

Fig. 8: Gadgets for CFI-bypassing ROP. The attacker needs to correctly guess four address values to launch a successful attack: the address of gadget #1, gadget #2, and memcpy(), and the address in the enclave to leak (i.e., `rdi`). We assume that an attacker already knows the implementation details of memcpy() in that `rsi`, `rdi`, and `rdx` were used for corresponding function parameters.

Windows version. In order to support Windows enclave programs, we implemented a separate PE editor, bootstrapping program, and secure in-enclave loader. The PE editor embeds dummy sections in the secure in-enclave loader to reserve enough code and data pages. The bootstrapping program and the secure in-enclave loader for Windows are almost the same as those for Linux, but we only solve the compatibility issues, including type definitions and system calls.

Others. The rest of the components of SGX-Shield are libraries used by an enclave program and debugging tools. We port `musl-libc` [5] as a `libc` and `mbedTLS` [4] as a `TLS` library to SGX-Shield. Since `libc` code often invokes system calls, we replace those system calls to trampolines/springboards.

VI. Evaluation

In this section, we evaluate SGX-Shield by answering the following questions:

1) How effectively does SGX-Shield defend against various types of memory-based attacks (§VI-A1)?
2) How much randomness does SGX-Shield show in its address space layouts (§VI-A2)?
3) How much performance overhead would SGX-Shield impose in running the micro-benchmarks (§VI-B1)?
4) How much performance overhead would SGX-Shield impose in running typical workloads for SGX (§VI-B2)?

Experimental setup. All our experiments were conducted on Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz (Skylake with 8MB cache) with 32GB RAM. We ran Ubuntu 14.04 with Linux 3.19 64-bits[3], and installed Intel SGX SDK and device drivers released by Intel [28]. In the entropy analysis (§VI-A2) and the micro-benchmark (§VI-B1), we used `nbench` [2] benchmark suites.

A. Security Evaluation

This subsection evaluates how many security guarantees are offered by SGX-Shield. We first evaluate the practical security aspects of SGX-Shield by measuring the possibility of successful memory corruption-based attacks (§VI-A1). Then, we evaluate the theoretical and general security aspects of SGX-Shield by measuring the entropy (§VI-A2).

[3] We also performed the same evaluation in the Windows version of SGX-Shield. While the result is almost same, we do not show it in this paper because of the page limitation.

	Exploitation technique		
Attack model	Ret-to-func	ROP	CFI-bypassing ROP
Remote	$0/2^{14}$ (48-bits)	$0/2^{14}$ (88-bits)	$0/2^{14}$ (108-bits)
Passive kernel	$2^{14}/2^{14}$ (0-bits)	$2^{14}/2^{14}$ (0-bits)	$2^{14}/2^{14}$ (0-bits)
Active kernel	$2^{14}/2^{14}$ (0-bits)	$2^{14}/2^{14}$ (0-bits)	$2^{14}/2^{14}$ (0-bits)

(a) Intel SGX SDK (baseline)

	Exploitation technique		
Attack model	Ret-to-func	ROP	CFI-bypassing ROP
Remote	$0/2^{14}$ (48-bits)	$0/2^{14}$ (88-bits)	$0/2^{14}$ (108-bits)
Passive kernel	$0/2^{14}$ (20-bits)	$0/2^{14}$ (60-bits)	$0/2^{14}$ (80-bits)
Active kernel	$148/2^{14}$ (7-bits)	$0/2^{14}$ (21-bits)	$0/2^{14}$ (28-bits)

(b) SGX-Shield

TABLE III: Security effectiveness of SGX-Shield against memory corruption-based attacks. For each attack model, we launched 2^{14} attacks to the vulnerable enclave program running with either the current Intel SGX SDK or SGX-Shield. In each cell, x/y (z-bits) denotes the following: x - the number of successful attacks; y - the total number of attacks we tried; and z - the theoretical number of bits that the attack needs to bruteforce.

	SGX SDK	SGX-Shield	
		RU-64	RU-32
H^{Rel}	0.0	0.9989	0.9993
H^{Abs}	0.9869	0.9999	0.9999

(a) Code pages

		Stack	Heap	Global
H^{Rel}	SGX SDK	0.0	0.0	0.0
	SGX-Shield	0.9886	0.9995	0.9967
H^{Abs}	SGX SDK	0.9869	0.9869	0.9869
	SGX-Shield	1.0000	1.0000	1.0000

(b) Data pages

TABLE IV: The ASLR entropy on code and data pages while running the nbench binary 1,800 times. The higher entropy value indicates more randomness on address layouts. H^{Rel} denotes entropy for relative addresses and H^{Abs} denotes entropy for absolute addresses. SGX SDK denotes the baseline results using the existing Intel SGX Linux SDK. RU-64 and RU-32 represent the configured size of a randomization unit, 64- and 32-bytes, respectively.

1) Effectiveness against Memory Corruption Attacks: In order to see how effective SGX-Shield is in stopping memory corruption-based attacks, we launched an attack against a vulnerable enclave program while running either Intel SGX SDKs and SGX-Shield. We assume following the three attack models in which each has different prior-knowledge on memory address layouts according to their inherent runtime constraints: (1) a remote attack, which launches an attack through network sockets serviced by a vulnerable enclave program. This model is a blind attack (i.e., it knows nothing related to the address layouts); (2) a passive kernel attack, which has the privilege of an underlying operating system but does not intervene a page fault handling mechanism. Since the kernel executes `EEINIT` and `EADD`, this attack model has information on the base address and the size of an enclave; (3) an active kernel attack, which not only has the privilege of an underlying operating system but also actively intervenes the page fault handling mechanism. The active intervention on the page fault follows the controlled side-channel attack [61], which grants additional information on which memory page is being accessed by an enclave program (more details are discussed in §VII).

In order to focus on ASLR-related issues, we wrote an easily exploitable victim program with a simple stack-overflow vulnerability. Then, for each attack model, we run the following four exploitation techniques, where each imposes different difficulties in guessing address values: (1) return-to-function, which requires inferring a single address value (i.e., a function address); (2) ROP, which requires three ROP gadgets (i.e., need to infer three address values). The gadgets are the same as the ones in RIPE benchmark [60], but we replaced `call` with `syscall` to work in an SGX environment; (3) CFI-bypassing ROP, which requires four ROP gadgets (i.e., need to infer four address values as shown in Figure 8). This CFI-bypassing ROP manipulates only the data flow, so it would not be detected by CFI techniques [7] but requires more gadgets than non-CFI-bypassing ROP.

As expected, the current Intel SGX SDK was effective against a remote attack model, but ineffective against passive and active attack models. As shown in Table III, in the case of the remote attack model, all our attack attempts (2^{14} times) failed, as theoretically an attacker has to try about 2^{47}, 2^{87}, and 2^{107} times to achieve a 50% successful attack probability per exploitation technique, respectively. In the case of passive and active kernel attack models, since the attacker is already in possession of required address values for all three exploitation techniques, exploitation attempts were always successful for all 2^{14} attacks that we tried.

With SGX-Shield, however, the probabilistic defense nature of ASLR is regained for all attack models and exploitation techniques. In the remote attack model, SGX-Shield showed the same security results as the Intel SGX SDK. In the case of the passive kernel attack, all 2^{14} attack attempts failed. If the size of a randomization unit is 32 bytes, there are 2^{20} possible entry points that attackers have to bruteforce for each address value (i.e., 32 MB/32 B = $2^{25}/2^{5}$, as the code or data region size is 32 MB). Thus, theoretically the attacker has to guess 2^{20}, 2^{60}, and 2^{80} address values for each exploitation technique. In the active kernel attack, the attacker now may know which memory page is responsible for executing certain code in the worst case of SGX-Shield. However, since SGX-Shield still shuffles both code and data pages in the memory page, the unknown bits for a single address value would be 7 (i.e., 4 KB/32 B = $2^{12}/2^{5}$, as the memory page size is 4 KB and the randomization unit size is 32 B). Therefore, a theoretical bound of SGX-Shield's security guarantee is 2^{7}, 2^{21}, and 2^{28}, for each exploitation technique, respectively. Accordingly, this theoretical estimation is also evidenced by our real attack trials — while successful attacks were observed 148 times for return-to-function, all failed for ROP and CFI-bypassing ROP. Although SGX-Shield's probabilistic bound against the return-to-function exploitation technique can be a security concern, we believe the security benefit of SGX-Shield is still valuable considering that return-to-function is difficult to be a general exploitation in practice.

2) Entropy Analysis: We measure the randomness of the address space layout using the notion of entropy [15]. The entropy captures the uncertainty of a given random variable, and we apply this by considering possible address values as a random variable. Specifically, let $\mathbf{A}_{\text{RU}}$ be a discrete random variable with the absolute entry point addresses $\{a_1, a_2, ..., a_n\}$ for a certain randomization unit RU across n different runs, and $p(a_i)$ is a probability mass function (pmf). Then, $H^{\text{Abs}}(\mathbf{A}_{\text{RU}})$, the normalized address space layout entropy of the randomization unit RU, is defined as:

$$H^{\text{Abs}}(\mathbf{A}_{\text{RU}}) = -\sum_{i=1}^{n} p(a_i)\frac{\ln p(a_i)}{\ln n}.$$

Moreover, due to the normalization factor $\ln n$,

$$0 \leq H^{\text{Abs}}(\mathbf{A}_{\text{RU}}) \leq 1$$

always holds. $H^{\text{Abs}}(\mathbf{A}_{\text{RU}})$ is zero when the randomization unit is always mapped to the same address for all n runs. It is one when all runs always result in a different address. We also measure the entropy on the relative address to better understand the randomness of our ASLR scheme when the base address of an enclave memory is known. This entropy is resented as $H^{\text{Rel}}(\mathbf{A}_{\text{RU}})$.

Using these two entropy measures, we computed the entropy for code pages and data pages, as shown in Table IV: SGX SDK denotes the baseline results; for the entropy of code pages, we configured the size of the randomization unit as 32- or 64-bytes (i.e., RU-32 and RU-64); for the entropy of data pages, we measured it for stack, heap, and global data objects. Note, especially for data pages, we computed the entropy by replacing an entry point address of a randomization unit into the base address of each data object.

Code page entropy. H^{Rel} in Table IV-(a) shows the effectiveness of SGX-Shield's approach against strong adversaries. As expected, while Intel SGX Linux SDK provides no randomness (i.e., the entropy value is zero), SGX-Shield provides a very high degree of randomness. This is because the SDK picks a random base address and loads the program to the base address in a deterministic way. Smaller randomization units (RU-32) provide a higher degree of randomness (compared to RU-64).

On the other hand, if the attacker is completely blind, then both Intel SGX Linux SDK and SGX-Shield provide good randomness, as shown in H^{Abs}.

Data page entropy. We now describe the entropy of SGX-Shield compared to Linux SGX SDK for data objects, including stack, heap, and global variables, as shown in Table IV-(b). Similar to the code page entropy, Linux SGX SDK shows no randomness on all data pages against strong adversaries (i.e., H^{Rel} is zero for stack, heap, and global). In contrast, SGX-Shield shows very high randomness across all data objects. Stack object shows the least randomness among these. The reason is that SGX-Shield still needs to allocate the continuous space to preserve the functionality of stack, even though it picks stack's base address at random and imposes no alignment on the base address in order to maximize the randomness. Assuming blind attackers with no information, Intel SGX SDK shows reasonable randomness, but SGX-Shield shows close to perfect randomness given the number of sample runs (H^{Abs}). Across all 1,800 runs, SGX-Shield exhibited unique and random base addresses for all data objects.

B. Performance Overhead

We now evaluate the performance overhead imposed by SGX-Shield. In order to understand the performance aspects in the worst-use-cases as well as typical-use-cases, we run SGX-Shield on both the micro-benchmark and macro-benchmark.

1) Micro-benchmark: We run each testcase of the `nbench` benchmark suites [2] 200 times and report the median value. In each run, `nbench` iterates through its task at least 10,000 times, and it returns the average time to perform the task once. To clearly see where the performance overhead comes from, we run SGX-Shield with various settings, changing the size of randomization units and opt out software-DEP and SFI.

Benchmark	Baseline	SGX-Shield			
		RU-64		RU-32	
	(μs)	ASLR	ASLR&DEP	ASLR	ASLR&DEP
Num sort	1262	-1.22%	1.88%	3.65%	2.30%
String sort	6077	2.62%	18.98%	7.29%	31.67%
Fp emu.	12140	0.81%	10.05%	5.78%	29.77%
Assignment	43613	1.28%	1.79%	6.89%	3.89%
Idea	387	-0.14%	-0.55%	-0.72%	-0.67%
Huffman	445	2.85%	15.29%	28.65%	25.96%
Neural net	34618	2.10%	7.26%	8.73%	22.20%
Lu decomp.	1080	0.09%	0.39%	2.08%	2.52%
Average	0.00%	1.05%	6.89%	7.80%	14.71%

TABLE V: Runtime performance overhead of SGX-Shield when running *nbench*. **Baseline** column is a native run under SGX without SGX-Shield and it is measured in microseconds. All columns in SGX-Shield are represented with relative overheads in a percentage compared to the baseline. **RU-64** and **RU-32** denote a size of a randomization unit, 32- and 64-bytes, respectively. **ASLR** and **ASLR&DEP** denote before and after applying SGX-Shield's DEP and SFI techniques, respectively. The average relative standard deviation is 0.71% (the maximum is 2.45%).

Table V shows the performance overhead imposed by SGX-Shield in running `nbench`. An elapsed time in microseconds is represented in the baseline column, while all other columns under SGX-Shield are represented in a relative overhead compared to the baseline in a percentage. In this table, RU-32 and RU-64 denote the size of a randomization unit, 32- and 64-bytes, respectively.

As the size of a randomization unit becomes smaller, the performance overhead increases. More specifically, before applying DEP and SFI, RU-32 imposes 6.75% more overhead compared to RU-64 (i.e., from 1.05% to 7.80%). Once DEP and SFI are applied, RU-32 imposes 7.82% more overhead (i.e., from 6.89% to 14.71%). This additional overhead is expected, as SGX-Shield introduces more randomization units in RU-32, and thus instruments more unconditional branches. Moreover, a smaller randomization unit implies a negative impact on code cache performance, as there will be more frequent control transfers.

DEP and SFI techniques of SGX-Shield also slow the execution of `nbench`. With RU-64, SGX-Shield shows 1.05% overhead if DEP and SFI were not applied. If these were applied together, SGX-Shield showed 6.89% overhead on average. Similarly, with RU-32 SGX-Shield showed 7.80% and 14.71% overhead on average before and after applying DEP and SFI, respectively. In other words, the performance overhead of SGX-Shield's DEP and SFI is 5.84% and 6.91% in RU-32 and RU-64, respectively.

To better understand the performance impacts of SGX-Shield, we also counted the number of executed instructions in runtime while running the benchmarks. The performance overhead that SGX-Shield imposes is directly related to the number of executed instructions. To implement fine-grained randomization, SGX-Shield instruments a terminator instruction at the end of a randomization unit. This alone results in 8.86% or 13.1% more executed instructions on average (See Table VI). Moreover, to implement DEP and SFI, SGX-Shield

Benchmark	Baseline	SGX-Shield			
		RU-64		RU-32	
		ASLR	ASLR&DEP	ASLR	ASLR&DEP
Num sort	5,245 K	6.55%	21.38%	14.68%	28.36%
String sort	38,017 K	81.89%	274.49%	97.29%	314.47%
Fp emu.	66,553 K	7.07%	23.37%	14.46%	35.84%
Assignment	301,104 K	8.16%	7.73%	13.26%	15.95%
Idea	224,000 K	5.80%	6.47%	13.57%	12.79%
Huffman	295,379 K	6.23%	12.07%	13.44%	19.14%
Neural net	263,275 K	8.76%	21.56%	18.86%	32.82%
Lu decomp.	7,967 K	7.43%	9.22%	17.27%	21.73%
Average		16.49%	47.04%	25.35%	60.14%

TABLE VI: The number of instructions executed in runtime while running *nbench*

	Baseline	SGX-Shield			
		RU-64		RU-32	
		ASLR	ASLR&DEP	ASLR	ASLR&DEP
# instr.	29 k	37 k	42 k	39 k	45 k
# RU	-	5,663	5,938	8,430	9,161
Binary size	212 KB	548 KB	584 KB	724 KB	792 KB
— code+data	131 KB	160 KB	177 KB	170 KB	193 KB
— metadata	68 KB	374 KB	391 KB	541 KB	586 KB

TABLE VII: A static overhead of SGX-Shield to the *nbench* binary. Note that the `nbench` benchmark suites contain a single binary, which takes an argument to specify a certain testcase. **# instr.** denotes the number of instructions in a binary; **# RU.** denotes the number of randomization units that SGX-Shield generated. **Binary size** denotes a size of a binary, including code and data as well as metadata; **code+data** denotes a size of both code and data segments in a binary. **metadata** denotes a size of symbal, relocation, and string table in a binary.

instruments many memory accessing instructions. This further increases the number of executed instructions by 30.55% and 34.79% for RU-64 and RU-32, respectively. The results show that DEP and SFI have a stronger impact on the number of executed instructions.

In terms of memory overhead, SGX-Shield actually imposes fixed overhead due to over-estimation. More precisely, in the current version of SGX-Shield, it imposes total 64 `MB` memory overheads (i.e., 32 `MB` for code pages and 32 `MB` for data pages).

Looking into more detail on possible factors of memory overhead, while SGX-Shield preserves the size of data objects, it enlarges the size of code due to the randomization unit-level ASLR and software-DEP. Particularly, ASLR also increases the size of metadata including fine-grained symbol, string, and relocation table entries. Table VII shows that the increased binary size is mainly from more metadata.

Based on these evaluation results, we recommend that RU-64 with DEP and SFI would be a reasonable configuration. The address space layout showed fairly good randomness compared to RU-32, and its runtime performance is 7.82% faster than RU-32.

2) Macro-benchmark: In order to see how SGX-Shield would work with real-world workloads, we ran an HTTPS server within an enclave. We ported `mbedTLS` [4], which is an open source Transport Layer Security (TLS) library. `mbedTLS` also includes a sample HTTPS server, where its work process can be broken into the following two parts: (1) SSL handshaking, an initial and fixed cost for a request and (2) reading an HTML file and then sending it to the client. `mbedTLS`

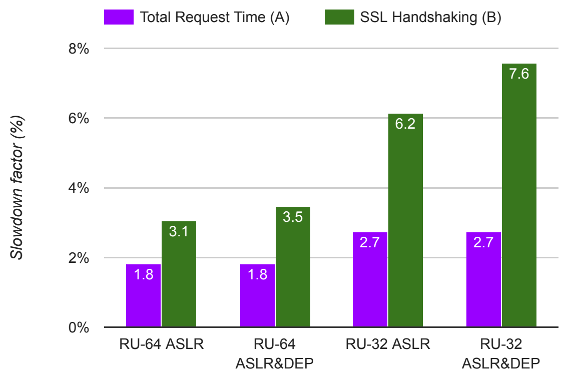

Fig. 9: Performance overheads in running an HTTPS server with `mbedTLS`. Each bar represents the slowdown factor (%) of SGX-Shield compared to the baseline (i.e., Intel SGX SDK for Linux): the bar on the left (marked as `A`) shows the slowdown in the total elapsed time for a request, and the bar on the right (marked as `B`) shows the slowdown in the elapsed time for SSL handshaking. We ran 50 times and report the median value. The median of the total elapsed time in the baseline is 1.1 second, while the one for SSL handshaking is 0.359 second. The average relative standard deviation is 3.35% (the maximum is 5.83%).

provides authentication and key sharing mechanisms for SSL handshaking, and it also encrypts (or decrypts) messages on sending (or receiving). We ran this HTTPS server, which serves the HTML file of 12KB size. The average round-trip time (RTT) between the server and the client was 175.5 `ms`, an average of 50 times running a `ping` command with 0.32% average relative standard deviation.

Figure 9 shows the overheads of requesting the HTML file from the HTTPS server. We computed the median of 50 times of the requests and plotted the slowdown factor as a percentage between the baseline (i.e., Intel SGX SDK for Linux) and SGX-Shield: the first bar (marked as `A`) represents the slowdown in the total request, and the second bar (marked as `B`) represents the slowdown during the SSL handshaking.

Overall, SGX-Shield imposed negligible overheads in the total request time, from 1.8% to 2.7%, depending on the setting. This performance number is much better compared to the micro-evaluation results, and we suspect this is because the total request time is dominated by the network latency, which is independent of SGX-Shield's extra work with instrumentation. This can be supported by the increased slowdown numbers in SSL handshaking, which is less related to the network operations and more related to computational jobs, ranging from 3.1% to 7.6%. Similar to the micro-evaluation results, enabling DEP incurred more slowdowns due to its extra instrumentation for software-based DEP — 0.4% increases in RU-64 and 1.4% increases in RU-32. According to these results, we believe SGX-Shield would be fast enough to run realistic workloads for SGX while providing ASLR security guarantees together.

VII. Discussion on Controlled Side-channel Attacks

The controlled side-channel [61] allows an attacker to infer data values at runtime by observing coarse-grained execution flows (i.e., a sequence of page faults). If an application running within an enclave exhibits control-flow dependencies relying on specific data values, this attack is indeed possible because page resources are managed by adversaries (i.e., the kernel) in the SGX model.

While this attack is not directly related to breaking ASLR, it can still be applied to partially infer address layout information. In other words, although the kernel does not know memory layouts after in-enclave loading, it may infer some layout information by observing (or intentionally triggering) page faults. For example, assume that there are four code objects (`A`, `B`, `C`, and `D`) and a global variable (`v`), and `A` has the branch: if `v` is 1, jump to `B`; if `v` is 0, jump to `C`, and then jump to `D`. In this case, if the attacker observes the number of page faults as 2, she/he may conclude that `v` is 1. On the other hand, if the number of page faults is 3, she/he may conclude that `v` is 0.

SGX-Shield's design decisions on randomization units effectively thwart this side-channel attack, as shown in §VI-A1. Specifically, because SGX-Shield randomly places multiple randomization units in a memory page, when the size of the randomization unit is 32 bytes, an attacker needs to bruteforce 2^7 times (i.e., 4KB/32bytes = $2^{12}/2^5$, as the memory page size is 4 KB) to guess a single address value, while each failure would end up crashing a target enclave program. Considering the number of address values that need to be guessed in practical memory corruption exploits (e.g., three or more in ROP [17, 18, 60]), we believe this probabilistic defense against the controlled side-channel would be effective and reasonable in practice.

VIII. Related work

Secure systems based on Intel SGX. The early adoption of Intel SGX focused on the cloud environment to cover security problems from an untrusted cloud platform (e.g., cloud provider). Haven [12] is a system to securely run the entire library OS (LibOS) in an enclave as a guest OS to prevent access from untrusted software with a malicious purpose. Similar to SGX-Shield, the LibOS loads the actual target programs at runtime. However, it does not support ASLR, which leads to several threats as mentioned in §IV-A. Additionally, the TCB of Haven is very large (more than 200MB) because of the nature of OS, while the TCB of SGX-Shield is only 8KB (1821 instructions in the x86-64 assembly). To reduce the TCB size of the libOS approach, Scone [9] suggests the container based sandbox in an enclave.

Starting from the cloud environment, the systems and frameworks that apply Intel SGX to enhance security have been proposed. VC3 [46] is a secure data processing framework based on the Hadoop framework [1] to keep the confidentiality and integrity of the distributed computations on the untrusted cloud. VC3 suggests a self-integrity invariant that prevents an enclave program from reading from or writing to the non-enclave memory region. It basically aims to avoid data leakages and memory corruptions. This technique is similar to software-DEP or SFI of the SGX-Shield, but the one of SGX-Shield aims to prevent an attack from injecting code to execute. Ryoan [26] also adopts SFI in SGX to guarantee data privacy between multiple distrusted parties. The S-NFV system architecture [52] proposed a new design of the secure Network Function Virtualization (NFV) system based on the Intel SGX. Kim et al. [33] suggest security enhancements of the network systems such as software-defined inter-domain routing and peer-to-peer anonymity networks by adopting the Intel SGX.

OpenSGX [31] is a software platform that emulates the SGX hardware and provides basic software components (e.g., system call interface and debugging support) and toolchains.

Security issues of Intel SGX. While Intel SGX provides the protection of the program against access from privileged software and hardware, several studies argue the vulnerabilities of the enclave program as an open problem. Attackers can perform various types of Iago attacks [19] through the communication channel between the enclave program and the external world. Moreover, potential side-channels [61] (e.g., page fault) exist that helps the untrusted privileged software to guess the secret data of the program. To address this problem, Shinde et al. [54] designed a defense mechanism that enforces a program to access its input-dependent pages in the same sequence regardless of the input variable. This approach, called deterministic multiplexing, ensures that the OS cannot distinguish the enclave execution.Since the performance overhead of Shinde et al. [54] is too expensive in practice (4000 times without developer's help), T-SGX [53] suggests a defense based on Intel Transactional Synchronization Extensions (TSX) to hide page faults against the untrusted kernel.

Also, incorrect use of SGX instructions or bugs related to memory accesss inside the enclave makes enclave programs vulnerable. To handle this problem, Moat [56] suggests a new programming model that checks services related to the security of the SGX program (e.g., remote attestation and cryptographic sealing). It not only verifies the confidentiality of an enclave program, but also checks whether the enclave program actually leaks the data. Rohit et al. [55] introduce a runtime library that offers an interface to securely communicate with the external party of the enclave. It also provides core services for the secure memory management and runtime checks for verification.

Commodity TEEs and software-based solutions. While much commodity hardware, including Intel SGX [30, 39] and ARM TrustZone [8],provide Trusted Execution Environments (TEEs), Sancus [43] designs a hardware architecture for TEEs. To the best of our knowledge, Secure OS of the ARM TrustZone does not support ASLR and software DEP [51]. However, applying ASLR and software DEP to ARM TrustZone is another research issue to be explored with different challenges (e.g., different side-channel) compared to SGX-Shield.

There are several approaches to shield applications from untrusted privileged software in the software manner [22, 25, 35, 63]. Minibox [35] ensures mutual distrust between the program code and the OS on top of a trusted hypervisor with small TCB (pieces of application logic). CloudVisor [63] protects the virtual machines of customers by separating resource management from the virtualization layer. InkTag [25] proposes the defense mechanism against compromised system call interfaces to protect persistent storage, and Virtual Ghost [22] similarily protects the memory from the host OS using compiler instrumentation.

ASLR and runtime re-randomization. ASLR is applied to commodity OS [3, 44] to defend against `return-to-libc` [42] and `return-oriented-programming` (ROP) attacks [49] by obfuscating locations of code gadgets. However, several ways to bypass ASLR have been reported [36, 48, 50, 58], stemming from the low entropy of randomness [36, 50] and memory disclosures [48, 58]. To address the low entropy issue, many fine-grained ASLR techniques [13, 23, 24, 32, 45, 59] claim that randomizing the code in various fine-grained units (e.g., basic block or instruction level) can be a solution. Several studies [10, 21, 38] show that the encryption of visible pointers and non-readable executable pages prevents attackers from abusing memory disclosures.

The runtime re-randomization [14, 34, 37] is a strong defense mechanism against both brute-force attacks and memory disclosure exploits. In particular, RUNTIMEASLR [37] and Oxymoron [11] aim to protect from attacks using random memory corruption tests during the process forks [16]. By re-randomizing the memory layout of child processes, attackers cannot guess the memory layout of the parent process. Similar to process fork, in Android system address space of the user process is copied from a pre-initialized process called *Zygote* that makes the memory layouts of user processes the same at the initial state. Morula [34] re-randomizes the child process to mitigate this problem.

Software DEP. The software DEP design of SGX-Shield is inspired by Native Client (NaCL) [47, 62]. NaCl [62] proposes an efficient SFI mechanism based on masking instructions and adopting the memory segment of an x86 system. The goal of NaCl is to sandbox a memory region in a user process to run a third-party component such as an untrusted library in the region. The next version of NaCl [47] extends it to ARM and x86-64 architectures. The instrumentation of software DEP in SGX-Shield is similar to NaCl on x86-64, but we cannot assume that the base address of data pages is aligned with 4GB, while NaCl for x86-64 makes the assumption. Because of this limitation, our software DEP has a penalty to add one more `sub` instruction for the instrumentation.

IX. Conclusion

In this paper, we identified fundamental challenges in enabling ASLR for the SGX environment. We took the real-world example, Linux and Windows SDKs for Intel SGX, and found its critical security limitations. This paper also proposes a solution, SGX-Shield, a new ASLR implementations for SGX programs. SGX-Shield incorporates a secure in-enclave loader, software DEP, and software fault isolation to provide secure ASLR for SGX. The evaluation that we conducted on the real Intel SGX hardware demonstrates SGX-Shield's effectiveness in both security and performance.

X. Acknowledgment

We thank the anonymous reviewers and our proofreader, Tricia Grindel, for their helpful feedback. This work was supported in part by BSRP (NRF-2015R1D1A1A01058713), Office of Naval Research Global (ONRG), IITP (B0101-15-0557) funded by the Korea Government (MEST), KAIST Venture Research Program for Graduate & Ph.D students, and NSF awards DGE-1500084, CNS-1563848 and CRI-1629851.

References

[1] "Apache hadoop project." [Online]. Available: http://hadoop.apache.org/

[2] "Linux/unix nbench." [Online]. Available: http://www.tux.org/~mayer/linux/bmark.html

[3] "Documentation for the pax project (address space layout randomization)," 2003. [Online]. Available: https://pax.grsecurity.net/docs/aslr.txt

[4] "mbedtls," 2016. [Online]. Available: https://tls.mbed.org/

[5] "musl-libc," 2016. [Online]. Available: https://www.musl-libc.org/

[6] "Writing an llvm backend," 2016. [Online]. Available: http://llvm.org/docs/WritingAnLLVMBackend.html

[7] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, p. 4, 2009.

[8] T. Alves and D. Felton, "Trustzone: Integrated hardware and software security," *ARM white paper*, vol. 3, no. 4, pp. 18–24, 2004.

[9] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. OâĂŹKeeffe, M. L. Stillwell *et al.*, "Scone: Secure linux containers with intel sgx," in *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, Nov. 2016.

[10] 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 *Proceedings of the 21st ACM Conference on Computer and Communications Security*, Scottsdale, Arizona, Nov. 2014.

[11] M. Backes and S. Nürnberger, "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing," in *Proceedings of the 23rd Usenix Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[12] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," in *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014.

[13] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Efficient techniques for comprehensive protection from memory error exploits." in *Proceedings of the 14th Usenix Security Symposium (Security)*, Baltimore, MD, Aug. 2005.

[14] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely rerandomization for mitigating memory disclosures," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, Denver, Colorado, Oct. 2015.

[15] C. M. Bishop and N. M. Nasrabadi, *Pattern recognition and machine learning*. springer New York, 2006.

[16] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[17] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses," in *Proceedings of the 23rd Usenix Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[18] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, Chicago, Illinois, Oct. 2010.

[19] S. Checkoway and H. Shacham, *Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface*, Houston, TX, Mar. 2013.

[20] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th Usenix Security Symposium (Security)*, San Antonio, TX, Jan. 1998.

[21] 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 *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[22] J. Criswell, N. Dautenhahn, and V. Adve, "Virtual ghost: Protecting applications from hostile operating systems," 2014.

[23] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Proceedings of the 21st Usenix Security Symposium (Security)*, Bellevue, WA, Aug. 2012.

[24] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "Ilr: Where'd my gadgets go?" in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[25] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: Secure applications on an untrusted operating system," in *Proceedings*

of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Houston, TX, Mar. 2013.

[26] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," in *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, Nov. 2016.

[27] Intel, *Intel Software Guard Extensions Enclave Writer's Guide*, 2015, https://software.intel.com/sites/default/files/managed/ae/48/Software-Guard-Extensions-Enclave-Writers-Guide.pdf.

[28] ——, *Intel Software Guard Extensions Evaluation SDK for Windows OS*, 2016, https://software.intel.com/en-us/sgx-sdk-support/documentation.

[29] ——, *Intel(R) Software Guard Extensions SDK for Linux* OS*, 2016, https://01.org/sites/default/files/documentation/intel_sgx_sdk_developer_reference_for_linux_os_pdf.pdf.

[30] ——, *Intel Software Guard Extensions Programming Reference (rev2)*, Oct. 2014.

[31] P. Jain, S. Desai, S. Kim, M.-W. Shih, J. Lee, C. Choi, Y. Shin, T. Kim, B. B. Kang, and D. Han, "Opensgx: An open platform for sgx research," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[32] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (aslp): Towards fine-grained randomization of commodity software," in *Annual Computer Security Applications Conference*. IEEE, 2006, pp. 339–348.

[33] S. Kim, Y. Shin, J. Ha, T. Kim, and D. Han, "A first step towards leveraging commodity trusted execution environments for network applications," in *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets)*, Philadelphia, PA, Nov. 2015.

[34] B. Lee, L. Lu, T. Wang, T. Kim, and W. Lee, "From zygote to morula: Fortifying weakened aslr on android," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[35] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry, "Minibox: A two-way sandbox for x86 native code," in *Proceedings of the 2014 ATC Annual Technical Conference (ATC)*, Philadelphia, PA, Jun. 2014.

[36] L. Liu, J. Han, D. Gao, J. Jing, and D. Zha, "Launching return-oriented programming attacks against randomized relocatable executables," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*. IEEE, 2011, pp. 37–44.

[37] K. Lu, S. Nürnberger, M. Backes, and W. Lee, "How to make aslr win the clone wars: Runtime re-randomization," Feb. 2016.

[38] 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 *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, Denver, Colorado, Oct. 2015.

[39] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution." in *HASP@ ISCA*, 2013, p. 10.

[40] I. Molnar, "Exec shield," *new Linux security feature*, 2003.

[41] ——, "Nx (no execute) support for x86, 2.6.7-rc2-bk2," *LWN.net*, 2004.

[42] Nergal, "The advanced return-into-lib(c) exploits: Pax case study," *Phrack*. [Online]. Available: http://phrack.org/issues/58/4.html

[43] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base." in *Proceedings of the 22th Usenix Security Symposium (Security)*, Washington, DC, Aug. 2013.

[44] S. A. T. R. Ollie Whitehouse, Architect, "An analysis of address space layout randomization on windows vista," *White paper*, 2007.

[45] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[46] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "Vc3: Trustworthy data analytics in the cloud using sgx," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[47] 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 *Proceedings of the 19th Usenix Security Symposium (Security)*, Washington, DC, Aug. 2010.

[48] F. J. Serna, "The info leak era on software exploitation," *Black Hat USA*, 2012.

[49] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, Alexandria, VA, Oct.–Nov. 2007.

[50] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, Washington, DC, Oct. 2004.

[51] D. Shen, "Attacking your trusted core: Exploiting trustzone on android," *Blackhat*, 2015.

[52] M.-W. Shih, M. Kumar, T. Kim, and A. Gavrilovska, "S-nfv: Securing nfv states by using sgx," in *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*. ACM, 2016, pp. 45–48.

[53] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-sgx: Eradicating controlled-channel attacks against enclave programs," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2017.

[54] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing your faults from telling your secrets: Defenses against pigeonhole attacks," *arXiv preprint arXiv:1506.04832*, 2015.

[55] R. Sinha, M. Costa, A. Lal, N. Lopes, S. Seshia, S. Rajamani, and K. Vaswani, "A design and verification methodology for secure isolated regions," in *Proceedings of the 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 2016.

[56] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani, "Moat: Verifying confidentiality of enclave programs," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, Denver, Colorado, Oct. 2015.

[57] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2013.

[58] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *Proceedings of the Second European Workshop on System Security*. New York, NY, USA: ACM, 2009.

[59] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 19th ACM Conference on Computer and Communications Security*, Oct. 2012.

[60] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "Ripe: Runtime intrusion prevention evaluator," in *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 2011, pp. 41–50.

[61] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[62] 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 *Proceedings of the 30th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2009.

[63] F. Zhang, J. Chen, H. Chen, and B. Zang, "Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, Oct. 2011.