Abstract

Intel Software Guard eXtensions (SGX), a hardware-based Trusted Execution Environment (TEE), has become a promising solution to stopping critical threats such as insider attacks and remote exploits. SGX has recently drawn extensive research in two directions—using it to protect the confidentiality and integrity of sensitive data, and protecting itself from attacks. Both the applications and defense mechanisms of SGX have a fundamental need—flexible memory protection that updates memory-page permissions dynamically and enforces the least-privilege principle. Unfortunately, SGX does not provide such a memory-protection mechanism due to the lack of hardware support and the untrustedness of operating systems.

This paper proposes MPTEE, a memory-protection mechanism that provides flexible and efficient enforcement of memory-page permissions in SGX. The enforcement relies on our elastic cross-region bound check technique which uses only three bound registers but provides six memory permissions. To defend MPTEE against potential attacks, we further develop an efficient mechanism that exploits the in-place bound-check technique to ensure the integrity of the memory protection. With MPTEE, developers can enhance the protection for data and code in SGX enclaves and enforce the least-privilege principle such as Execute-no-Read memory readily. We have implemented MPTEE and extensively evaluated its effectiveness, utility, and performance. The results show that MPTEE incurs a performance overhead of only 2%–8%, and is effective in ensuring memory protection and in defending against potential attacks.

1 Introduction

Hardware-based trusted execution environments (TEEs), as a way to protect the confidentiality and integrity of data and code, emerge in today’s market. In particular, Intel has provided SGX in its commodity processors, which supports a secure region, namely enclave, to protect the internally loaded code and data. Given its important and practical protection, SGX has been extensively studied and used in practice. For example, SGX-Shield [37] designs a memory-randomization scheme for SGX environments, and SGXBOUNDS [22] provides an efficient memory-safety approach for the security of objects in SGX.

Another line of research is to protect SGX itself from attacks. While being useful and practical, SGX still suffers from a variety of attacks [5, 7–11, 24, 25, 36, 52, 54]. It is not only vulnerable to various side-channel attacks [8–11, 25, 36, 52, 54], but also traditional memory-corruption attacks [5, 24] because the code inside SGX may still be vulnerable. To defend against these attacks, researchers have attempted to harden SGX [22, 37, 45, 46]. For example, SGX-Shield [37] designs a memory-randomization scheme for SGX environments, and SGXBOUNDS [22] provides an efficient memory-safety approach for the security of objects in SGX.

Both the SGX applications and the defense mechanisms have a fundamental need—flexibly and securely enforcing memory-page permissions, such as write (W), read (R), and execute (X), in a least-privilege manner. For example, SGX-ELIDE [3] and SGXCrypter [49] ensure enclave-code confidentiality with code packing or encryption. Code generation at runtime thus requires to remove the W permission for code pages—code pages must be non-writable to be compatible with the R permission. On the other hand, defense mechanisms for SGX also require changes to memory permissions. For example, SGX-Shield requires to remove the W permission of code pages after randomization. This way, it can ensure that it would not introduce the traditional code-injection attacks [29]. Overall, a flexible and secure memory-permission control is required to enforce the least-privilege principle and to prevent attacks.

While flexible memory-permission enforcement is important, SGX, unfortunately, does not provide such a mechanism due to two main reasons. First, Intel provides very limited hardware support due to security considerations. The current SGX does not provide instructions for modifying the permis-
sions of the Enclave Page Cache (EPC) after the enclave has been initialized [14] [32]. This is to prevent the untrusted OS from changing the access permissions of EPC to facilitate attacks at runtime. Second, permissions for memory pages of an SGX program are statically decided by the GCC compiler which however does not support flexible permissions such as execute-only memory. One cannot modify the permission configuration of the generated SGX program due to the signature-verification mechanism. The signature [33] is computed for the content and the layout of enclave memory, and its page security properties at build time. The SGX loader will check the signature at load-time and reject the program if the signature cannot be verified.

As SGX does not support the change of memory permissions, existing applications have attempted multiple solutions which are however insecure or inefficient. Specifically, SGX-Shield [37] uses software-based DEP to create an NRW boundary (i.e., non-readable and -writable boundary) to remove the R and W permissions for code pages. While wasting the R15 register, it also incurs an extra overhead for computing the range to check. Meanwhile, the NRW boundary using a general register can be shifted [5]. SGXELIDE [3] modifies the p_flags field in the program header entry to make the section writable throughout the enclave’s lifetime. This is insecure because code pages are subject to code-injection attacks after adding writable permission. SGXDecrypter [49] relies on the OS page table to remove the W permission, which is incompatible with the SGX security model, namely the fact that the page table is managed by the untrusted OS. In summary, all of these works are only for a single permission change, and their designs waste registers, incur a significant performance overhead, or introduce security issues.

Supporting a flexible, efficient, and secure memory protection mechanism for SGX is in fact challenging for two reasons. **R1. Limited hardware support.** SGX currently does not have hardware support for flexible memory-permission enforcement. Although it is possible to implement the enforcement with a software-based solution, the runtime and memory overheads tend to be significant, needless to mention, the size of the code running in SGX will also be significantly increased. **R2. Strong adversary.** The security model of SGX assumes that the privileged software (e.g., OS, hypervisor) is untrusted. We thus cannot simply ask the OS or hypervisor to change the memory permissions. In addition, SGX programs themselves might be vulnerable and thus subject to a variety of attacks that may abuse the permission changes.

In this paper, we present MPTEE, a memory-protection system that provides flexible, efficient, and isolated memory-permission enforcement for SGX. MPTEE realizes memory-permission enforcement by bound-checking memory accesses of different permissions, using Memory Protection Extensions (MPX). For example, by bound-checking all memory reads against a specific range, we can ensure that only the memory region specified by the range is readable. Unfortunately, MPX provides only four registers, but a flexible memory protection requires at least six permission combinations. Spilling bounds from registers to memory will significantly incur performance overhead and require the integrity protection for the bounds in memory. To address these problems, we propose a novel cross-over memory-layout design that uses only three bound registers to offer the six common memory permissions (e.g., execute-only or read-only memory) efficiently. We name the design elastic cross-region bound check (CRBC). CRBC is generic; it is also applicable to embedded systems that lack the Memory Management Unit (MMU) but require flexible memory protection.

Since the memory-protection mechanism runs in the same address space as the potentially vulnerable SGX code, a remaining problem with MPTEE is that adversaries may abuse the mechanism to maliciously change memory permissions, invalidating the permission enforcement. We thus further provide the enforcement integrity technique which employs memory isolation and control-data integrity (CDI) to protect the memory permission enforcement mechanism from attacks. Note that, while memory isolation and CDI are well studied, MPTEE exploits the in-place CRBC technique to further improve their performance and security. With MPTEE, developers can readily enforce memory permissions flexibly, efficiently, and securely.

We have implemented a prototype of MPTEE and evaluated its security, effectiveness, utility, and performance using representative SGX programs. The evaluation results show that the MPTEE can provide efficient permission settings to prevent existing known attacks, and can resist against potential attacks that try to bypass or abuse MPTEE. MPTEE’s protection has a small runtime performance average overhead—6.6% for SQLite and 2.18% for Memcached. Moreover, the enforcement-integrity mechanism, which includes memory isolation and control-data integrity, incurs less than 1% runtime performance overhead, benefiting from the in-place CRBC technique. We believe that MPTEE is a practical and secure memory-protection mechanism that is ready for protecting SGX applications and SGX itself.

We have the following research contributions in this paper.

- **The novel cross-region bound check technique.** We propose a new technique to flexibly and dynamically enforce six common memory permissions of SGX using only three MPX bound registers. The cross-over layout design of the technique does not require OS support or specific hardware features. It can also be ported to bare-metal systems in embedded devices that do not provide memory protection [12].
- **An efficient and secure design.** We design and implement MPTEE to efficiently realize the cross-region bound check technique for SGX. MPTEE also employs CDI and memory isolation to ensure the integrity of permission enforcement. More importantly, relying on the in-place cross-
region bound-check technique, MPTEE further improves the performance and security of CDI and memory isolation.

- **Case studies and extensive evaluation.** We provide multiple use cases that can benefit from MPTEE. By applying MPTEE to representative SGX programs, we thoroughly evaluate its effectiveness, utility, and performance.

2 Overview

In this section, we first present the background and threat model of MPTEE, and then introduce the MPTEE approach.

2.1 Background

**Intel MPX.** Intel MPX is a new instruction set architecture (ISA) extension, a hardware-assisted full-stack solution to protect against memory safety violations [28]. It provides new instructions and registers for software-based bounds checking, making it much more efficient. Specifically, MPX provides four dedicated bound registers (BNDK, BND3) and instructions for setting (bndmov), moving (bndmov), and checking (bndc, bndc1) against bound registers for addresses. MPX also provides a bound table, similar to a two-level page table structure, to extend the number of bounds. Due to the scarcity of bound registers, a typical use of MPX is to use a bound table to store a large number of bounds for objects.

**Memory-permission enforcement.** The memory-page permission of traditional applications is enforced by the permission bits of the OS page table. In Linux syscalls, mmap() and mprotect() are used to set or update the memory page permission by updating the bits of page-table entries. In the x86 architecture, each page-table entry contains 2 permission bits; one is the R/W bit which presents whether the page is executable, and the other is the R/W bit which presents whether the page is writable. Under the existing x86 architecture, memory pages are readable by default.

The SGX hardware provides three new permission bits for SGX pages, R, W, and X bits which are contained in the SGX enclave control structure (SECS)—the metadata structure of an EPC. These three bits should be initialized only once during the load-time. They are checked at each address translation, enforced by hardware, so even an untrusted OS cannot break it. In comparison, the permission bits in page tables cannot limit whether the memory page is readable, but SGX permission bits can. Further, page tables are managed by the OS, which is untrusted in the SGX threat model.

2.2 Threat Model

In MPTEE, we assume that the adversary can control all the software (e.g., OS kernels and hypervisors) and hardware except the SGX component. That is, we assume that only Intel SGX itself is trusted, and all other software and hardware components are untrusted. The adversary can freely read and write the content in memory. We also assume that the code running inside SGX may have any kind of vulnerabilities such as buffer overflows that can be exploited by adversaries. The adversary can perform any static or dynamic analysis to find any patterns in the SGX code. Denial of service [20] and side-channel [10, 11, 25, 54] attacks, such as power and timing analysis, are out of the scope.

2.3 The MPTEE Approach

In general, memory protection can be realized with either hardware-based or software-based approaches. Hardware-based approaches use hardware features to control access rights such as the special permission bits [26] [50, 53]. By contrast, software-based approaches use software-fault isolation (SFI) mechanisms [51] to restrict memory accesses [30, 38] or use exception mechanisms to check the access permissions [2]. The hardware-based approaches tend to have lower overhead but lack flexibility, while the software-based are the opposite.

In MPTEE, we aim to propose a solution benefiting from both the software-based and hardware-based approaches. Our solution focuses primarily on software-based approaches and requires minimum hardware support (e.g., only some registers), so that it is flexible, efficient, and potentially generic. Since permission enforcement itself may be attacked when the enclave code has vulnerabilities, we also need to protect the enforcement. Figure 1 depicts MPTEE’s overview. MPTEE is mainly comprised of two components: permission enforcement and enforcement integrity.

2.3.1 Permission Enforcement

The permission-enforcement component of MPTEE aims to flexibly enforce memory permissions, which includes two parts: dynamic permission enforcement and adaptive permission enforcement. Dynamic permission enforcement provides the ability of runtime permission changes for SGX applications while adaptive permission enforcement effectively optimizes the performance of permission enforcement.

**Dynamic permission enforcement.** Our key technique for achieving dynamic permission enforcement is elastic cross-region bound check (CRBC). Its intuition is that, by bound-checking memory accesses, including read, write, and execute, we ensure that they are restricted in the correct permission regions. For example, the bound-checking for reads ensures that all memory reads can only target a specific memory range. That is, any memory that is out of the range is non-readable. Moreover, the bound is stored in a register. By adjusting the bound of the readable memory, we can realize a readable memory region with a flexible range. This way, we enforce the memory-read permission.

For performance reasons, we use Intel MPX for efficient bound checks. However, there are some limitations to using MPX for efficient permission enforcement. The memory layout of traditional Linux programs includes many non-contiguous memory regions (i.e., sections) with different per-
missions. Such a many-region layout makes the MPX bound checks less efficient—MPX has only four bound registers; additional bounds will have to be stored to bound tables in memory, which will, however, cause two problems. First, it incurs performance issues [22, 28]. The bounds must be frequently loaded from memory, incurring significant performance overhead. Second, it also introduces security issues—the integrity of bounds (saved in memory) must be guaranteed.

Therefore, how to use only a limited number of registers to efficiently and securely complete the enforcement of multiple permissions in the enclave is a challenging problem. To address this problem, CRBC relies on a novel cross-over design. The design is based on our key observation that the same permission memory range is continuous in an enclave (details are presented in §3.1). CRBC requires only three bound registers but offers six memory regions with different permission combinations—RWX, RW, RX, R, X, non-permission. Note that the WX combination is disallowed to be aligned with the NX enforcement, and the write-only permission is also disallowed because it does not have a practical use scenario.

Adaptive permission enforcement. We further reduce the performance overhead of the memory enforcement by proposing adaptive permission enforcement which is based on an observation that memory regions that do not require permission changes can be protected with the SGX permission bits. For example, when we know that the size of executable memory is 2MB and that the size will not change at runtime, we can use the X permission bit to enforce the executable memory, so that CRBC only needs to bound-check memory reads and writes but not executes, which saves one bound register and avoids bound-checking for executes. When all memory regions of different permissions have a fixed size, we can completely avoid bound-checking and thus remove all the performance overhead of CRBC (see details in §3.2). SGX permission bits are checked through the hardware circuit, without any additional instructions, so it is much faster than MPX bound-checking.

As we will show in §3.2, the current compilation tool-chain does not allow the flexible configuration of the permission bits statically, we thus have to redesign the SGX parser and loader, which is also shown in §3.2.

2.3.2 Enforcement Integrity
Since the permission-enforcement component is in the same address space as all other code in SGX that is potentially vulnerable, it can be abused by attackers through the vulnerabilities. Therefore, we also develop enforcement integrity to protect the component, which includes two techniques, efficient memory isolation and control-data integrity (CDI).

The memory isolation mechanism is to prevent data-flow attacks that aim to manipulate variables (e.g., a variable controlling the size of a memory region) that influence the permission enforcement. All such variables will be collected and saved in the isolated memory region, thus will be protected from being manipulated. On the other hand, CDI prevents control-flow attacks that may bypass bound-checks. By ensuring the integrity of control data, attackers cannot hijack control flows.

Although CDI and memory isolation are well studied, our enforcement integrity has some unique advantages. First, it leverages the in-place CRBC to achieve memory isolation for free. Second, since control data like the trampoline table [15] can be readily protected using the non-permission memory offered by the in-place CRBC, we manage to further improve both the efficiency and security of traditional control-data protection. As will be shown in §6, our enforcement integrity technique incurs a runtime performance overhead of less than 1%, and is more secure than CFI techniques like CFCC [47] and Readactor [15].

3 Flexible and Efficient Memory-Permission Enforcement
We first elaborate on the flexible enforcement of memory permissions. The main contribution is the dynamic permission enforcement, which (1) uses only three bound registers to
efficiently offer six permission combinations; (2) allows the flexible changes of the ranges of memory regions at runtime. Its key technique is elastic cross-region bound check which employs a cross-over design. In addition, we propose adaptive permission enforcement which uses SGX permission bits to boost the performance of CRBC.

### 3.1 Elastic Cross-Region Bound Check

**Unique memory layout of enclave.** Before we elaborate on the design of the elastic cross-region bound check technique, we first introduce the unique memory layout of enclaves. We found that enclave programs do not support linking dynamic libraries [33]; all required libraries must be statically linked in the target enclave program. Therefore, enclave programs tend to have a very simple memory layout. That is, the memory sections that have the same permissions are adjacent. The official SGX manual [32] presents a typical enclave memory layout, as shown in Figure 2 (a). We also analyzed the source code of the SGX SDK and found that the layout table has exactly followed this memory layout. To make sure that existing SGX applications also follow this memory layout, we analyzed 15 representative open-sourced SGX programs, including sgx-migration [39], SGXCryptofile [42], TresorSGX [48], SGX-Shield [41], etc. Except for SGX-Shield, all these SGX programs follow this memory layout as well. SGX-Shield adds a new section .sgxcode to store the randomized code, and thus the two executable sections are not adjacent. However, this new section .sgxcode can be placed right after the original code section. In their implementation, although there are some guard pages between sections, these pages do not have any permission. Based on these analyses, we conclude that, in general, SGX programs follow a simple memory layout where memory regions with the same permission are adjacent.

The observation motivates us to design an efficient bound-checking mechanism, elastic cross-region bound check (CRBC), that completely avoids using bound tables but uses only three bound registers to maintain a new memory layout shown in Figure 2 (b).

**Cross-region bounds.** The key idea is to implement a cross-over design with three basic permission regions (i.e., R/W/X). The new layout provides five regions with specific permission combinations and one reserved region without any permission, namely, X (execute-only), R (read-only), RX (read and execute), RW (read and write), RX (all permissions), and non-permission. We do not support permission W because “writing to non-readable memory” does not have practical uses. Since bound registers are changeable, the flexibility can be intuitively realized by adjusting the bounds.

CRBC leverages MPX to efficiently bound-check multiple regions with different boundary registers. Specifically, we use only bnd0, bnd1, and bnd2 to delimit the ranges of executable memory, writable memory, and readable memory, respectively. We carefully design the overlapping between the ranges represented by bnd0, bnd1, bnd2 to achieve six different permission combinations. The representation of each area is shown in Table 1. To bound-check memory accesses, we identify all memory read/write operations and indirect control transfers (i.e., indirect calls). We then instrument them to bound-check their addresses. Through such bound checks, we can effectively prevent invalid accesses such as writing to a non-writable range, reading from a non-readable range, or transferring the control flow to a non-executable range. This way, we eventually realize adjustable memory regions with different permission combinations.

The most important part of the flexible permission enforcement is to properly control the three bound registers.

**Initializing the bounds.** According to the new layout, to initialize the bound registers, we only need to know the size of reserved areas. MPTEE provides two ways of setting the size. First, MPTEE has an automated setting that determines the bounds based on the section information of an ELF file. This automated setting is based on our observation that the dynamic changing of region ranges typically happens in the heap, so it is reasonable to set the size the same as the one of the heap. Second, we also allow developers to specify the size of each reserved area in the configuration file. Manually configuring memory usage is in fact a common practice in SGX. For example, the max size of heap and stack should be configured in the SGX configuration file before running the enclave code. After knowing the size, we can compute the range of each permission region and initialize the bound registers.

**Updating the bounds.** Through adjusting the bounds, we can implement two types of permission changes: permission reduction and permission extension. The reserved areas (see Figure 2 (b)) located at the boundary of regions are used by the memory objects which require to change their permission. Let us take the just-in-time (JIT) code generator as an example. First, the JIT program uses some RWX memory to store generated code, whose start address is bnd1.ub. JIT writes the generated code A at bnd1.ub. When it needs to remove the W permission of code A for security, we just assign bnd1.ub with bnd1.ub – size_of_A. As bnd1.ub has decreased, the write to code A will throw an exception. When JIT continues to generate more code, bnd1 can continue to shrink the upper bound. This way, we implement permission change from RWX to RX by adjusting the bound.

Permission reduction includes RWX to RX, RWX to RW, RW to R, RX to X while permission extension is the opposite. However, adjusting the bounds by itself cannot fully realize permission changes. For example, in the case where B is set to RX, an SGX program may want to revert the permissions of A that have changed from RWX to RX. In this case, increasing the boundary to revert A’s permissions will affect B’s permissions. We emphasize that this is a limitation
with MPTEE—the memory that has contained data cannot
be used for bound extension of other permissions. However,
we believe such a conflict is not common in practice. For
example, both SGX-Shield and SGXELIDE require to only
remove the W permission once. In fact, we can also overcome
this limitation with memory movement. By moving the data
in the conflicting memory to other memory regions, we can
use the memory for the extension. However, all references
to the data must be carefully updated. Currently, the user needs to be aware of the changes in the
memory layout during the memory movement. In the future,
we will enable automatic pointer update during the memory
copy, similar to runtime memory rerandomization [4].

Permission enforcement using CRBC. CRBC provides
four APIs for developers to use, as shown in Table 2. CRBC
provides two ways to implement permission enforcement
through these APIs. First, providing the memory with requir-
ing permission. Reserved areas with different permissions
allow us to place the appropriate data based on the require-
ments of the application. The reserved area can be viewed as
a mini-heap and managed by a uniform memory-allocation al-
gorithm. Developers call the mpt_mmap to apply for the buffer
in the reserved area with the target permissions. mpt_write
is used to write the data to the buffer. When the data is no
longer needed, mpt_munmap frees the buffer. Second, provid-
ing permission changing. mpt_mremap relies on adjusting the
register boundary or memory copy to support dynamic permis-
sion changes. Algorithm 1 shows how to make the decision
in mpt_mremap. In the algorithm, cur_region denotes the re-
region where addr is located. flags_region denotes the region
which has the target permission. neighbor_region denotes
the region which is adjacent to cur_region.

Improving EPC usage. CRBC pre-allocates EPC pages for
the reserved area at initialization. If the reserved area is never
used, e.g., when the enclave program does not change mem-
ory permissions, the EPC pages will never be used, which
degrades the EPC usage. To improve EPC usage, we use the
SGX seal mechanism [33] to free up unused reserved area.
CRBC immediately seals the reserved area out of the enclave
upon the completion of initialization. However, whenever the
reserved area is used, CRBC loads back the memory to con-
tinue the EPC memory accesses. This way, we ensure a good
EPC usage. SGX seal uses AES-GCM (Advanced Encryption
Standard) to encrypt the data; it utilizes Intel Advanced En-
cryption Standard New Instructions (Intel AES-NI) which is
immune to software-based side-channel attacks. In addition,
the seal policy can be used to defend against replay attacks.
As the implementation uses hardware instructions, the per-
formance overhead is low, especially when the operation is
performed only once.

The design of CRBC has the following advantages. First, it
supports all the necessary permission combinations. Second,
it supports to modify the region boundary dynamically, which
enables the change of memory permissions flexibly. Third,
it only uses three bound registers and never stores or loads
bound registers to or from bound tables in memory, thus
minimizing the performance overhead.

3.2 Optimizing CRBC: Adaptive Permission
Enforcement

When one permission region has no change requirement at
time, we employ the SGX permission bits, instead of
bound-checking, to validate memory accesses, which relies
on a hardware check to reduce the performance overhead of
CRBC.

3.2.1 Configuring the SGX Permission Bits

To enable the adaptive permission enforcement, developers
specify the region size and the corresponding permissions.
MPTEE will then properly set the permission bits according
to the configuration.

Table 1. CRBC uses only three bound registers to
realize 6 permissions. bi denotes the lower bound.
ub denotes the upper bound. ImageBase is
the start address of enclave. non-perm. means
that the region does not have any permission.
Algorithm 1: How to change pages permission to flags with using mpt_mremap.

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void *mpt_mmap(size, flags)</td>
<td>Acquires a memory buffer, which is at least size bytes. It is restricted as specified in flags.</td>
</tr>
<tr>
<td>void *mpt_mremap(pages, size, flags)</td>
<td>Changes old_pages to flags. If the pages are located at a region boundary, it changes the permissions; otherwise, it allocates new memory buffer and copy content over.</td>
</tr>
<tr>
<td>void *mpt_munmap(pages)</td>
<td>Frees an acquired memory buffer, the freed memory will be sealed and swapped out.</td>
</tr>
<tr>
<td>void *mpt_write(pages, size, content)</td>
<td>Writes content to the mapped region. The function will not be bound-checked.</td>
</tr>
</tbody>
</table>

The inflexibility with the permission bits. Unfortunately, the current compilation tool-chain does not allow the flexible configuration of the permission bits statically. For example, compilers do not support permissions such as execute-only memory. Moreover, the permission bits contained in the program header do not help CRBC to decide which bound register should be removed. CRBC needs to know the permission region which does not require changes before instrumentation. As such, we have to re-design the SGX parser and loader to support adaptive permission enforcement.

Re-design the SGX parser and loader. Figure 3 shows the new workflow of building an SGX program. Specifically, developers can specify the permissions and size which will not be changed at runtime by setting the Static node in the config.xml. The Static node value is represented with the following simple format.

\[
\begin{align*}
\text{perm} & : R \mid W \mid X \\
\text{num} & : 1, 2, \ldots
\end{align*}
\]

R, W, X represent three permissions while num represents the region size of corresponding permission.

In the process of generating an enclave program, the parser extracts the permissions from the configuration file. As the configuration file will be discarded at runtime, we record this information in SGX metadata. The loader will construct the SECS structure according to the setting permissions, then pass it to the SGX driver which will finally use it to initialize the enclave properly. This way, developers can flexibly configure permissions. After that, we need a script to pass the information to the process of applying CRBC, so that the corresponding bound register will be removed when building the SGX program.

Figure 3. The new workflow of building SGX program after applying the adaptive permission enforcement.

4 Enforcement Integrity Against Attacks

MPTEE enforces memory permissions through cross-region bound check which is subject to two kinds of attacks: control-flow attacks that bypass the bound checks and abuse the permission control, and data-flow attacks that manipulate bounds maliciously. To prevent both attacks, we propose enforcement integrity which incorporates two mechanisms, memory isolation and control-data integrity. The memory isolation protects all the code and data related to the permission enforcement by saving them in an isolated memory region (i.e., the non-permission region offered by CRBC). The control-data integrity (CDI) further prevents attackers from exploiting unintended instructions to bypass the bound checks. Its idea is to ensure the integrity of return addresses and function pointers.

Figure 4 shows the memory components of enforcement integrity: safe region, safe stack, and trampoline table. Safe region is used to protect the code and data for permission enforcement. Safe stack and trampoline table are to ensure the integrity of return addresses and function pointers, respectively. Note that our enforcement integrity shares a similar high-level idea as traditional CFI and isolation techniques such as CPI [23] and Readactor [15]. Our protection employs a conservative analysis, as CPI does, to recursively find all data that requires integrity protection. Therefore, in this section, we focus on two parts of enforcement integrity.
that differentiate from previous works: memory isolation and trampoline table which are more efficient and secure.

Efficient memory isolation using non-permission region.
We first provide data-flow integrity for the parameters used in MPTEE, we also employ a more efficient and secure design for the trampoline table with new instructions that load or store from the safe region. (e.g. overflow/underflow) against the sensitive data will not be prevented from accessing this region, which is enforced through bound checks. Therefore, any malicious modification (e.g. overflow/underflow) against the sensitive data will not change its value stored in the safe region.

**A more efficient and secure design for the trampoline table.** A trampoline table [15, 47] is a common approach for protecting control data such as function pointers—control transfers must go through entries in the trampoline table whose integrity is ensured. In MPTEE, we also employ a trampoline table to protect pointers for indirect calls and jumps. However, our design for the trampoline table is much more efficient and secure.

We first collect, through static analysis, all functions whose addresses are ever taken. At load time, our SGX loader will prepare a trampoline table that contains the actual addresses of these functions. Also, the table is in the safe region. Upon an indirect call or jump, we ensure that the target is one of the entries in the trampoline table. Figure 6 shows the principle of our design. The code pointer has been replaced with an index into the trampolines table. The code is instrumented as shown in Figure 5 to use the trampoline table.

![Figure 5](image.png)

**Figure 5.** The efficient and secure control-data integrity mechanism implemented by instrumenting indirect transfer using a trampoline table. This makes sure that indirect calls can only target a unique entry in the trampoline table.

![Figure 6](image.png)

**Figure 6.** Control data integrity instruments code to replace the indirect transfers with a trampoline address. All indirect transfers will be checked to enforce the aligned transferring.

Our trampoline table is more efficient and secure than previous works. IFCC [47] employs a standard trampoline table to protect indirect-call targets. Its table entry is a jump instruction; its implementation ensures that an indirect call can only point to a jump-table entry by masking the least significant bits. IFCC allows the control to transfer to any entry in the table. At the same time, it suffers indirect disclosure attacks when the code is randomized. Reactor [15] mitigates this issue by further mapping the trampoline table with the execute-only permission using the extended page table (EPT) mechanism. Our trampoline table benefits from the non-permission region offered by CRBC, which is non-writable and non-readable, and does not depend on the system privilege; therefore, its integrity and confidentiality are always guaranteed. In addition, as the permission check of non-permission region is realized by bound-checking the memory accesses of other permission regions, its overhead to ensure confidentiality is negligible compared to Reactor based on virtualization.

### 5 Implementation

We have implemented MPTEE, including its flexible memory-permission enforcement and enforcement integrity, as shown in Table 3.

<table>
<thead>
<tr>
<th>Technique</th>
<th>LoC</th>
<th>Base Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic permission enforcement</td>
<td>421</td>
<td>LLVM Backend</td>
</tr>
<tr>
<td>Adaptive permission enforcement</td>
<td>142</td>
<td>Intel SGX SDK</td>
</tr>
<tr>
<td>Enforcement integrity</td>
<td>1,832</td>
<td>LLVM Backend</td>
</tr>
<tr>
<td>Others</td>
<td>682</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table 3.** The implementation complexity of MPTEE in Linux. Our implementation involves modifying the source code of the SDK and instrumenting application code using LLVM [13].

### 5.1 Permission Enforcement

Permission enforcement involves the modification of SGX SDK and code instrumentation. We implement the four new APIs in the enclave library and modify the `sign_tool`, `parser`, `loader` to change the memory layout of the enclave according to Figure 2. We instrument new `bndmk`, `bndcu`, `bndo` instructions to initialize the bound registers and implement bound checks. Code instrumentation relies on an LLVM
IR pass to identify the memory reads, writes and the indirect transfers. Figure 7 shows an example of instrumentation code for X permission. Currently, the reserved areas are managed by a simple first-fit memory allocation algorithm. mpt_map/mpt_munmap are mapped to the malloc/free to the corresponding reserved areas according to the permission.

```
1  :("ptr_fun")()
2  lea ptr_fun, rcx
3  call ptr_fun
4  call rcx
5  lea ptr_fun, rcx
6  bnd0
```

Figure 7. Example of instrumentation code for permission enforcement. The left-hand side shows the original code. The right-hand side shows the instrumented code after using Intel MPX.

To enable adaptive permission enforcement, we use macros, __MPX_NOR__, __MPX_NOW__, __MPX_NOX__, to control MPX bound checks. If the macro is defined, the corresponding bound check is enabled; otherwise, it is disabled. A python script parses the config.xml to set the Macro. Then the script passes the macro value to the compiler through the -D flag. When applying this pass, the configuration in config.xml leads to different MPX code. If developers specify the X=2MB, -D__MPX_NOX__ will be passed to the compiler to remove the bound check using bnd0.

5.2 Enforcement Integrity

Enforcement integrity works on LLVM IR to identify sensitive data, function pointer assignments, and indirect control transfers. This IR pass constructs the trampoline table and inserts the computation instructions. Similar to CPI [23], to collect all sensitive data, we start from the arguments of the interface functions. We exploit data-flow analysis to trace the variables, which might propagate to the interface functions. Then, we recursively collect the data that may propagate to the previous variables. Through such a recursive analysis, we can conservatively collect all sensitive data that may propagate to the arguments of the interface functions. All of the accesses to the sensitive data will be instrumented to access the new values saved in the safe region.

In the implementation of control-data integrity, we instead collect all control data including return addresses, and targets of indirect calls and indirect jumps. Since we borrow SafeStack to identify and protect return addresses, in the implementation, we focus on the targets of indirect calls and jumps. Specifically, our IR pass analyzes the operand type of store instructions. If it is a function pointer or a label, the IR pass first finds whether this address exists in the trampoline table. If not, it adds its address to the trampoline table and replaces the operand with trampoline.size - 1. Otherwise, it just gets the index and replaces the operand with the index. As the aggregate variables may contain other aggregate variables, we also need to recursively analyze the nested variables.

6 Evaluation

In this section, we extensively evaluate MPTEE from the following perspectives: performance, utility, and effectiveness. Our experiments were performed on a Dell workstation, with an Intel Xeon E3-1225v5 CPU and 8G RAM, running an Ubuntu 16.04 Server, SGX SDK v2.0, SGX driver v0.10, and LLVM v6.0.

6.1 Performance Evaluation

We now evaluate the runtime and memory overheads of MPTEE with both macro-benchmarks and micro-benchmarks.

6.1.1 Macro-benchmark

SQLite is a C-language library that implements a small and fast SQL database engine. It is widely used in mobile phones and computers. The implementation of SQLite contains many memory access operations, we use it to measure the overhead of our system. We adopt a ported SGX version of SQLite [43]. In this experiment, we run speedtest, a performance testing script included in SQLite. We record the time cost for four operations into a database—inserting, selecting, updating and deleting. We run the test 50 times. In each test, we loop each operation 1,000 times and report the median.

![Figure 8. The runtime performance evaluation results of four frequently used operations in SQLite. CRBC denotes enabling cross-region bound check. EI denotes enabling enforcement integrity. MPTEE denotes enabling cross-region bound check, enforcement integrity and safe stack.](image)

Figure 8 shows the performance of the four database operations. Overall, MPTEE imposed an overhead in the total execution time from 2% to 8% with an average overhead of 6.6%. The source code of SQLite uses a large number of structures with function pointers as members. According to our instrumentation log, the address table (trampoline) contains 394 entries, and there are 3,736 address assignments for these entries. Figure 8 also shows the runtime performance impact of different techniques in MPTEE. CRBC imposes a 2%–7% overhead, but enforcement integrity imposes an overhead smaller than 1%, which, we believe, benefits from the extremely efficient design of isolation and control-data integrity. In these four operations, the selecting, updating, and deleting operations have a higher overhead than inserting. All of these three operations have conditions like WHERE b BETWEEN ?1 AND ?2. Intuitively, these operations require
more frequent memory accesses, thus leading to more bound checks. The experiment results are therefore expected.

**Memcached.** Memcached is an in-memory key-value store for small chunks of arbitrary data (strings, objects), which is widely used in production. We evaluated Memcached v1.5.22, which uses memaslap v1.0.18—a load generation and benchmark tool for Memcached servers. We set the get/set operation ratio to 1:1, the key size to 16 bytes, and run memaslap for one hour. After the instrumentation, its address table includes 99 entries, 9,597 bndcu/bndcl instructions. The original throughput is 256,292 (tps), and MPTEE slows it down by 2.18%. There is almost no significant change in the response time because the time taken to transfer data is much greater than the time taken to memory check operations. The results show that MPTEE imposes little performance overhead to Memcached.

According to these results, we believe that MPTEE is able to protect realistic SGX applications and their defense mechanisms efficiently.

### 6.1.2 Micro-benchmark

**Nbench** is a lightweight benchmark for testing CPU and memory performance, which includes a number of well-known algorithms. We use the ported SGX-nbench [40] as a microbenchmark. We run each testcase 100 times and report the median value. In each run, nbench iterates through its tasks 1,000 times and returns the average time cost. To breakdown the performance overhead, we run nbench with two settings: CRBC and enforcement integrity. Figure 9 shows the performance overhead by applying MPTEE in each testcase of nbench. Overall, MPTEE imposed a runtime overhead from 0% to 35% with an average overhead of 20%. The testcase, Fourier, is not affected by MPTEE. However, LU DECOMPOSITION is affected more significantly, with an overhead of 35%.

![Figure 9](image)

**Table 4.** The runtime performance overhead of using MPTEE in nbench. Baseline corresponds to a native run nbench without MPTEE, and it is measured in microseconds. The next three columns are represented with relative overheads in percentage compared to the baseline when we only enable one technique. SS denotes SafeStack while EI denotes enforcement integrity.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Baseline (µs)</th>
<th>CRBC</th>
<th>EI</th>
<th>SS</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMERIC SORT</td>
<td>534</td>
<td>19.46%</td>
<td>0.29%</td>
<td>0.34%</td>
</tr>
<tr>
<td>STRING SORT</td>
<td>537</td>
<td>20.47%</td>
<td>0.29%</td>
<td>0.32%</td>
</tr>
<tr>
<td>FP EMULATION</td>
<td>1,001</td>
<td>22.9%</td>
<td>0.98%</td>
<td>0.18%</td>
</tr>
<tr>
<td>FOURIER</td>
<td>20</td>
<td>0.38%</td>
<td>0.21%</td>
<td>0.24%</td>
</tr>
<tr>
<td>ASSIGNMENT</td>
<td>10,942</td>
<td>27.09%</td>
<td>1.59%</td>
<td>0.54%</td>
</tr>
<tr>
<td>IDEA</td>
<td>65</td>
<td>0.82%</td>
<td>0.77%</td>
<td>1.46%</td>
</tr>
<tr>
<td>NEURAL NET</td>
<td>7,384</td>
<td>27.98%</td>
<td>1.59%</td>
<td>-0.42%</td>
</tr>
<tr>
<td>LU DECOMPOSITION</td>
<td>287</td>
<td>34.28%</td>
<td>0.11%</td>
<td>0.28%</td>
</tr>
<tr>
<td>Average</td>
<td>0.00%</td>
<td>19.17%</td>
<td>0.72%</td>
<td>0.36%</td>
</tr>
</tbody>
</table>

**Table 5.** The number of instructions executed at runtime while running nbench.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Baseline</th>
<th>CRBC</th>
<th>EI</th>
<th>SS</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMERIC SORT</td>
<td>2,434K</td>
<td>19.51%</td>
<td>0.01%</td>
<td>0%</td>
</tr>
<tr>
<td>STRING SORT</td>
<td>3,260K</td>
<td>58.37%</td>
<td>2.21%</td>
<td>0.15%</td>
</tr>
<tr>
<td>FP EMULATION</td>
<td>8,545K</td>
<td>39.08%</td>
<td>-0.4%</td>
<td>0%</td>
</tr>
<tr>
<td>FOURIER</td>
<td>90K</td>
<td>13.5%</td>
<td>0.0002%</td>
<td>0%</td>
</tr>
<tr>
<td>ASSIGNMENT</td>
<td>73,725K</td>
<td>96.95%</td>
<td>0.25%</td>
<td>-0.06%</td>
</tr>
<tr>
<td>IDEA</td>
<td>643K</td>
<td>31.54%</td>
<td>1.47%</td>
<td>-0.04%</td>
</tr>
<tr>
<td>NEURAL NET</td>
<td>45,343K</td>
<td>81.91%</td>
<td>1.4%</td>
<td>0%</td>
</tr>
<tr>
<td>LU DECOMPOSITION</td>
<td>2,106K</td>
<td>117%</td>
<td>-0.08%</td>
<td>0%</td>
</tr>
<tr>
<td>Average</td>
<td>0.00%</td>
<td>57%</td>
<td>0.61%</td>
<td>0.01%</td>
</tr>
</tbody>
</table>

To better understand the performance impacts of MPTEE, we also counted the number of executed instructions in running the testcases of nbench. Table 5 shows the number of executed instructions in percentage. CRBC imposes an average of 58% more executed instructions. Enforcement integrity and SafeStack have a negligible impact on executed instructions. Although the number of executed instructions has increased by 58%, the newly added instructions, bndcu and bndcl, cost fewer clock cycles—the IPC (instructions/cycle) of the testbed is 1.97, and the IPC of bndcu and bndcl is 2. The newly added instructions are shorter than the average instruction length. The binary size actually increased by 48%. Currently, we instrument every memory write/read operation. However, we can reduce the size increase. First, we can check only the memory operations of the indirect address and relative address. Second, we can exploit an instruction optimization technique [30] to reduce the number of instructions. We believe that these optimizations can effectively reduce the binary size.

We also analyze the memory overhead and the change in binary size. Table 6 shows general static information about the binary. In particular, the #Inst column also shows that CRBC inserts more instructions than enforcement integrity. In addition, we can see that the binary size is bigger than the
default. The added part mainly contains the space occupied by the newly inserted instructions, and the added metadata part, such as the trampoline table. In term of memory overhead, MPTEE actually imposes negligible overhead as the memory that is added statically is much smaller than the memory space required at runtime, so the new memory is amortized by regular memory operation and can be considered negligible.

**Overhead of permission setting.** We also evaluate the performance overhead of two operations, `mpt_mmap` and `mpt_mremap`. We allocate objects with randomized permission, X/RX/RWX/RW/R, to compare the time overhead of `mpt_mmap` with normal `malloc`. We also construct two permission change types to compare the time cost of `mpt_mremap` with `mprotect`. First, that the target permission region is adjacent to the current region, and second, that it is not adjacent. We iterate the two comparisons 1000 times.

<table>
<thead>
<tr>
<th>Case</th>
<th>#Inst</th>
<th>#Bndck</th>
<th>#Indirect</th>
<th>Bin. size</th>
<th>code</th>
<th>data</th>
<th>MD. Mem.</th>
</tr>
</thead>
<tbody>
<tr>
<td>nbench</td>
<td>41K</td>
<td>N/A</td>
<td>N/A</td>
<td>290K</td>
<td>233K</td>
<td>36K</td>
<td>23K 5048K</td>
</tr>
<tr>
<td>nb-CRBC</td>
<td>49K</td>
<td>2.272K</td>
<td>N/A</td>
<td>430K</td>
<td>264K</td>
<td>36K</td>
<td>128K 5180K</td>
</tr>
<tr>
<td>nb-EI</td>
<td>47K</td>
<td>N/A</td>
<td>50</td>
<td>425K</td>
<td>264K</td>
<td>36K</td>
<td>125K 5268K</td>
</tr>
<tr>
<td>nb-MPTEE</td>
<td>51K</td>
<td>66,332</td>
<td>50</td>
<td>456K</td>
<td>294K</td>
<td>36K</td>
<td>125K 5380K</td>
</tr>
</tbody>
</table>

Table 6. The memory overhead of MPTEE to nbench. #Inst denotes the number of instructions in the nbench binary; #Bndck denotes the number of the bnduc and bndcl instructions that MPTEE instrumented. Binary size denotes the size of nbench, including code, data, as well as metadata; code denotes the size of the text section in the binary. data denotes the size of the data segment in the binary. metadata denotes the size of symbol, relocation, and string table in the binary.

![Figure 10. The normalized overhead of `mpt_mmap` and `mpt_mremap`.](image)

Figure 10 shows the results from the comparisons. `mpt_mmap` has a larger overhead than `malloc`. The implementation of `mpt_mmap` calls `malloc` after it ensures the heap according to flags. As we sealed the EPC pages out of the enclave, `mpt_mmap` needs to unseal the memory pages first, so `mpt_mmap` costs more time than `malloc`. When we use `mpt_mremap` to change the current permission to target, if they are adjacent, the overhead is less than the usage of `mprotect`. As `mpt_mremap` only needs to adjust the boundary, it is faster than calling `mprotect`, which needs to modify the page table and switch the context. However, if the two permission regions are not adjacent, the overhead is more than 10%. This is because the permission change needs to copy memory. Actually, the overhead depends on the size of the objects. In our experiment, the size is one memory page (4KB).

### 6.1.3 Benefits from adaptive permission enforcement

To show how the adaptive permission enforcement improves the performance of CRBC, we run nbench with different settings. CRBC denotes that we only use the original CRBC to enforce the permissions of different memory ranges. Adaptive-X denotes that we set the executable range statically and use CRBC to enforce the readable and writable memory permission. Adaptive-R denotes that we set the readable range statically and use CRBC to enforce the executable and writable memory permission. Adaptive-RWX denotes that we ensure the memory permission statically and do not need CRBC. Figure 12 shows the performance slowdown of each setting. As CRBC relies on instrumentation and bound-checking to enforce the memory permission, it can incur up to 20% runtime overhead. Adaptive-RWX instead purely relies on hardware for permission checking; it incurs almost zero runtime overhead. One interesting observation is that adaptive-X and adaptive-R have different slowdowns. Adaptive-X used the bound registers, bnd1 and bnd2, while Adaptive-R used bnd0 and bnd2. However, the frequency of bound-checking is different in these two cases, leading to different runtime overhead.

### 6.1.4 Compared with SGX-Shield NRW

SGX-Shield uses the R15 register to create the NRW boundary to remove the read and write permissions for code pages. We use the mbedTLS [27] to compare SGX-Shield with MPTEE. Note that SGX-Shield only supports executable-only permission using the dedicated R15 register. For a fair comparison, we removed the current software-DEP part from SGX-Shield and modify the `enclave_main()` as described in §6.2. Note that, in this evaluation, the adaptive permission enforcement is not enabled. Figure 11 shows the overheads of requesting HTML files from the HTTPS server on the original SGX-Shield and SGX-Shield with MPTEE. The slowdown of SSL handshaking is almost the same, but overall, SGX-Shield-MPTEE has a lower overhead—1.9%, which is 2.2% in the original SGX-Shield. We believe that the reduced number of available general-purpose registers creates additional register spilling. Note that SGX-Shield-MPTEE can be much better if the adaptive permission enforcement is enabled. Moreover, SGX-Shield uses the R15 register to implement the NWR boundary, which can only provide one permission. In comparison, MPTEE provides more permission ranges with adjustable sizes. Meanwhile, MPTEE also provides memory isolation and CDI while the original SGX-Shield does not have these capabilities.

### 6.2 Utility Analysis

SGX SDK provides nearly 90 SGX APIs. Compared to it, MPTEE introduces four additional APIs. The format for configuring regions is also simple. To confirm that MPTEE is an easy-to-use and practical tool, we apply it to two SGX
We now analyze the security of our mechanisms from the perspective of possible attacks in our threat model. There exist two kinds of possible attacks. First, an attacker might hijack the control flow by exploiting vulnerabilities in the enclave code. A control-flow attack might skip the bound check instructions by directly jumping into invalid memory reads/writes to protected regions that do not grant the corresponding permissions. We call this check-skipping attack. In this attack, permission enforcement is broken because the checks are not triggered at all. Alternatively, an attacker can exploit the enclave code vulnerabilities to call bound instructions to modify the bound registers. We call this attack bound-manipulating attack. In this attack, the permission enforcement is broken because bounds have been maliciously tampered.

**Preventing check-skipping attacks.** To successfully issue such attacks, the control flow must be hijacked by corrupting control data such as function pointers. After that, the attacks must use control transfer instructions, including call, jmp, ret, to jump to the invalid memory access instructions following bound-check instructions. Such attacks however cannot succeed in MPTEE because our control-data integrity mechanisms ensure that the control data cannot be corrupted. Therefore, MPTEE is secure from check-skipping attacks.

**Preventing bound-manipulating attacks.** Another possible attack is to manipulate the bounds by abusing the bndmk instruction. The bndmk instruction exists in only two locations. One is in mpt_mremap, and the other is in the enclave initialization. We have a clear rule that both cases cannot be invoked in an indirect way; therefore, we exactly know where the interfaces are called and thus where the bound variables are. Our memory isolation (Figure 4) ensures that all data that ever recursively influences the bound variables are identified as sensitive data and thus are isolated. This way, attackers cannot manipulate the bound variables. A remaining issue is that attackers may employ control-flow attacks to exploit unintended instruction (starting from the middle of a legal instruction) to manipulate bounds. This is also prevented by our control-data integrity mechanism.

### 6.3.2 Case Studies

**Protecting SGXELIDE code.** After we modified the source code of SGXELIDE as mentioned in §6.2, we compile and generate the enclave dynamic library program, enclave.so, using clang with our LLVM IR pass. To validate that we removed the W permission for the code section, we attempted to write the code section in SGXELIDE, which, as expected, triggered a segmentation fault crashing the program.

**Protecting SGX-Shield code.** We then use SGX-Shield with MPTEE to build a demo program. Then, we read the memory on a function address. As expected, we received a segmentation fault, and our program crashed. The test case confirms that the attempt of reading the execute-only memory is prevented. SGX-Shield is not secure because attackers can control the R15 register to shift the NRW boundary thus disabling SFI [5]. However, in MPTEE, values in the bound registers are never stored to memory, and bound-related data is protected using enforcement integrity. Therefore, MPTEE also improves the security of SGX-Shield.
7 Discussion

The current and most widely used variant of SGX is called SGX1. Intel will release the second generation called SGX2. SGX2 provides two special instructions, EMODPE and EMODPR, to support the dynamic permission changes. EMODPE is used to extend the access rights of an existing EPC page, which is running in Enclave Mode [32]. EMODPR is used to restrict the access rights in an initialized enclave, which can only be executed in a privileged mode. In our paper, we focus on SGX1 for two reasons. First, SGX1 is used widely. As a hardware component, SGX1 cannot be easily upgraded to SGX2. It is therefore foreseeable that SGX1 will still be the dominating version for a while. Second, MPTEE’s design is compatible with SGX2. Applications using MPTEE can run seamlessly in SGX2 without modification. More importantly, our permission reduction still only depends on itself, but MODPR depends on the system privilege in SGX2. At the same time, the permission-changing instructions in SGX2 are still subject to bypassing and abusing if the code in the enclave is vulnerable. As such, the enforcement integrity technique in MPTEE is still required, which is also compatible with SGX2.

8 Related work

Page-permission management. Memory Protection Keys (MPK) [53] is Intel’s upcoming hardware feature for controlling accesses to memory on a per-page basis. ERIM [50] exploits MPK to provide hardware-enforced isolation with low overhead in a non-SGX environment. However, MPK techniques have two drawbacks in memory isolation on SGX. First, its permissions only have accessible and writable, not executable. Second, it relies on the page-table structure, which is not consistent with SGX threat model. There is a security risk with modifying the protection key. MPTEE can provide permission enforcement securely.

Execute-only memory enforcement. Backes et al. [2] Gionta et al. [19] employ OS and hardware support to realize execute-only memory. In SGX, the OS is however assumed to be untrusted. Crane et al. proposed Readactor [15] and Readactor++ [16] to implement execute-only memory leveraging the extended page table mechanism (EPT). EPT allows marking pages as (non-)readable, (non-)writable, or (non-)executable. In addition, some other measures are used to provide secure environments. Pomonis et al. [30] proposed kR*X to protect the kernel by leveraging the execute-no-read memory and fine-grained KASLR. The authors rearranged the kernel code and data and used SFI to prevent invalid accesses to kernel code. In comparison, MPTEE offers various permission enforcements including execute-only memory efficiently by employing Intel MPX and SGX permission bits.

Memory isolation. IMIX [18] extends the x86 ISA with a new memory-access permission to mark memory pages as security sensitive, which can be leveraged as a primitive to protect the data of a wide variety of memory-corruption defenses. SGXBounds [22] provides an efficient memory-safety approach for SGX based on a simple combination of tagged pointers and a compact memory layout; it focuses on the memory safety of objects, but not the memory isolation of the program region. MemSentry [21] proposed a framework to isolate safe regions with commodity hardware features. Youren et al. [44] designed a multi-domain SFI scheme to resolve the tension between isolation and sharing in system software for SGX, which is implemented by leveraging MPX. To the best of our knowledge, MPTEE is the first to implement the flexible memory permission management on SGX.

9 Conclusion

In this paper, we presented a flexible, isolated, and efficient memory-protection mechanism, MPTEE, for Intel SGX. We propose multiple techniques that employ recent hardware features such as Intel MPX to flexibly and efficiently enforce different memory permissions (e.g., execute-only memory) for multiple regions in SGX. The elastic cross-region bound check uses only three bound registers but offers six different memory permission regions. The adaptive permission enforcement can further minimize the performance overhead. We also propose an efficient enforcement integrity technique to prevent attacks from bypassing or abusing MPTEE. We implemented MPTEE and thoroughly evaluated its performance and effectiveness. The evaluation results show that MPTEE is able to flexibly and effectively enforce memory permissions with a small performance overhead.

10 Acknowledgment

We would like to thank our shepherd, Rodrigo Rodrigues, and the anonymous reviewers for their feedback and suggestions. This research was supported in part by the National Natural Science Foundation of China (NSFC) under grants 61672421 and 61602363, and NSF award CNS-1931208. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References


