DOMAIN-BASED ISOLATION WITH SINGLE-CONTEXT TRUSTED EXECUTION ENVIRONMENT

An Undergraduate Research Scholars Thesis

by

PABLO SAY

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Dr. Chia-Che Tsai

May 2021

Major: Computer Engineering - Computer Science Track

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

I, Pablo Say, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Research Faculty Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

TABLE OF CONTENTS

ABSTRACT

Domain-based Isolation with Single-context Trusted Execution Environment

Pablo Say
Department of Computer Science and Engineering
Texas A&M University

Research Faculty Advisor: Dr. Chia-Che Tsai
Department of Computer Science and Engineering
Texas A&M University

Security continues to be an important topic as more businesses and individuals entrust software with sensitive information. One of the most important areas for security is that of the operating system and hardware of any computer – the most fundamental levels of any computer. Using Intel Software Guard Extensions (SGX), we set out to develop Nested Trusted Execution Environment Library Operating System (NesTEE LibOS), a prototype to build upon preexisting SGX features.

This paper overviews the NesTEE LibOS prototype, documenting performance, features, and feasibility of the proposed system. Currently, SGX does to secure enclave contents through privilege separation design. NesTEE LibOS modifies SGX by adding additional trust levels and a refined control flow of data moving in and out of the enclave. Designing NesTEE LibOS with more security subdomains is a crucial step towards expanding hardware security capabilities.

The subdomain model is as follows. For an application to interact with the enclave, the program must interact first interact with NesTEE LibOS entry code. The entry code separates NesTEE LibOS from the internal SGX application, managing page protections and creating a

separate stack before execution can be handed over to NesTEE LibOS. From this domain, the software can securely perform SGX functions and interact with the outer kernel. After NesTEE LibOS Execution is complete, control is transferred back to the internal application through the NesTEE LibOS exit code. This portion of code changes page permissions, making NesTEE LibOS memory pages inaccessible. By doing so, NesTEE LibOS is protected from tampering. This module relies on three levels of trust. The highest trusted level is that of NesTEE LibOS, followed by the application and kernel. Following with design choices made by SGX, the outer kernel and internal application is least trusted due to the possibility of corruption.

Measuring performance on the SGX versions of mprotect reveal the initialization cost for NesTEE LibOS as being very light. Contrarily, evaluations performed show NesTEE LibOS, though secure, can be relatively expensive in terms of execution time to accomplish common tasks when compared to a standard SGX architecture. Future work will certainly focus on improving the overhead costs to take advantage of NesTEE LibOS.

ACKNOWLEDGEMENTS

Contributors

I would like to thank my faculty advisor, Dr. Chia-Che Tsai, Harpreet Singh Chawla, my friends, and my family for their guidance and support throughout the course of this research.

Thanks also go to my friends and colleagues and the department faculty and staff for making my time at Texas A&M University a great experience.

Finally, thanks to my family for their encouragement, patience and love.

Funding Sources

Undergraduate research was supported by The Department of Computer Science and Engineering at Texas A&M University.

NOMENCLATURE

SGX	Intel Software Guard Extensions
SDK	Software Development Kit
CPU	Central Processing Unit
LibOS	Library Operating System
EENTER	Enclave Enter
EEXIT	Enclave Exit
EMODPE	Enclave Extend Page Permission
EMODPR	Enclave Restrict Page Permission
ENCLU	Enclave User Function
OCALL	Outer Call
IOCTL	Input Output Control
TRTS	Trusted Run Time System
RWX	Read Write Execute

1. INTRODUCTION

Our increased reliance on computing requires continuous security-related improvements to maintain trust between users and their devices. One such protective measure is Intel Software Guard Extension (SGX), a system which leverages hardware for additional security of private information via the CPU. In this paper, we document a proposed improvement to SGX's architecture to further improve reliability.

1.1 A Brief Overview of Intel SGX

SGX is a CPU instruction set which provides hardware level protection against many known attack vectors [4, 15]. SGX provides the ability to create enclaves: portions of memory protected by hardware. This feature is significant for its ability to prevent malicious code and compromised kernel-level software from seeing protected data. For example, sensitive financial details could be stored within an enclave and intra-enclave application code could perform computations on the data without interference from outside the enclave [6]. Should the outer kernel be compromised or hardware memory tapped from traditional means, the enclave contents will remain secure.

SGX uses a variety of techniques to seal the enclave off from the rest of the system. One of the most integral components is the attestation [4,14]. SGX requires external requests for data to provide a hash of enclave contents. Only after verifying the hash as legitimate will SGX disclose its contents. By design, SGX does not trust anything outside a secure enclave, including the outer kernel. The enclave's cautious approach towards the kernel serves to protect itself in the event of a corrupted administrative process. The enclave design SGX uses not only secures

information, but also protects code which are to perform computations on the data. As an added layer of security, SGX encrypts its contents if it detects a possible attack.

1.1.1 SGX Limitations

SGX does have some limitations. While the system reduces the scope of successful attack vectors, some still do exist. By design, SGX trusts enclave contents, which can include application code [4]. An issue surrounding application code is developers normally employ third-party libraries to perform certain actions [3, 16, 30]. In addition to their own application code, developers introduce a higher chance of bugs when introducing these external libraries. Hence, there exists the possibility of bugs in the application code corrupting the enclave. Additionally, SGX enclave have no controlling entity to mediate activity within the enclave. Without strict control of SGX functions, certain attack vectors will persist.

1.2 Need for More Sophisticated SGX Models

While SGX offers sufficient protection methods, it is limited in its ability to further separation within the enclave [4]. Enclave contents are not separated from one another, meaning application code within the enclave could potentially compromise the enclave itself. For instance, SGX deployment to compute sensitive information on a larger scale would require robust internal application code. Failure to do so by the developer could result in a compromised enclave and a potential data leak. This could also leave machines susceptible to malicious actors who could take control of the outer kernel. Should this occur, intervention would be required to regain control of the hardware. Without a more complex design, SGX's effectiveness is limited against some attack vectors.

Previous research has had a focus on different forms of protection [7,8,9]. Solutions explored includes encryption, attestation, and auditing of the system. However, none of these

solutions address the issue of no memory separation within an enclave. Without a solid form of internal enclave protection offered by SGX, users would spend considerable time investing in cybersecurity and similar methods to guarantee privacy in their services. Yet, sophisticated attackers could still bypass these protections should they somehow override the outer kernel of the machine itself or corrupt internal application code.

In this paper, we propose a modified SGX architecture which remedies this enclave vulnerability. Our solution is Nested Trusted Execution Environment Library Operating System (NesTEE LibOS), a modification of the SGX architecture to support memory separation within the kernel which further secures the enclave. Since the solution builds upon SGX, NesTEE LibOS provides additional hardware protection against a hostile kernel or rouge application though the use of nested kernel design and privilege separation.

The refined control flow we propose sees select application code and data inside the enclave as usual. However, the enclave will also contain NesTEE LibOS, a trusted execution framework which will handle enclave operations. NesTEE LibOS will handle interactions with the outer kernel and respond accordingly. To ensure NesTEE LibOS remains protected, memory permissions managed by the enclave will include that of NesTEE LibOS. Anytime the application wishes to interact with NesTEE LibOS, the enclave will update the page permissions as necessary using custom NesTEE LibOS functions. Should these custom functions be tampered with, NesTEE LibOS access will remain locked through an interrupt of the entry code.

Evaluations of the prototype support the proposed architecture of NesTEE LibOS. Results recorded for several use cases of NesTEE LibOS describes a trade off in performance for a more complex and secure version of SGX architecture.

2. METHODS

2.1 Threat Model

NesTEE LibOS adopts the same threat model as that of SGX itself alongside previous work on nested kernels [4, 5]. In designing the overall architecture, we identified several attack vectors. First, we assume there will come a time the outer kernel will attack the application within the enclave or NesTEE LibOS. Additionally, just as is possible with SGX, we assume the enclave application is imperfect and can sabotage NesTEE LibOS. Given these two threat models, it is not a stretch to also assume it is possible for both the outer kernel and enclave application to attack NesTEE LibOS memory pages together. Hence, NesTEE LibOS does not trust the outer kernel nor does it trust any enclave contents outside the LibOS itself. As such, any functions the internal application would normally perform is now handled by NesTEE LibOS.

In the design of NesTEE LibOS, we trust the base version of SGX to function properly and not fail against any attack vectors it has addressed in its design. We also trust the SGX SDK has not been tampered with. This is important as it is used extensively in the design of the exit gate. Additionally, hardware is trusted to not have any significant vulnerabilities.

2.2 Goals of NesTEE LibOS

This paper explores the thought process behind the creation of NesTEE LibOS and the viability of the solution. For NesTEE LibOS to be a viable solution, it must be built upon SGX. For us to create a solution from the ground up would produce errors and disregard the work already done to create a robust hardware framework. Regarding security, NesTEE LibOS must protect itself and enclave data against malicious actors. The option to handle security could be done using C code, however this approach can leave NesTEE LibOS vulnerable to attacks such

as buffer overflows [22]. Since nested kernel principles will be addressed, NesTEE LibOS will also serve as a control monitor for enclave activity. NesTEE LibOS can have excellent security, but it would find little viability if it is not easy for developers to use with their applications. Hence, a balance between security and usability must be maintained [23-25].

2.3 Technical Principles in NesTEE LibOS

For NesTEE LibOS to prove a viable solution, it should securely implement memory separation design into the enclave while having low impact to performance. Since we are building upon pre-existing SGX frameworks, our solution must also properly handle our additions in the event of an attack on the enclave.

2.3.1 Privilege Separation

An explored approach to improving SGX is the implementation of privilege separations and partitioning of enclaves. In essence, this approach rectifies one potential attack vector: faulty application code sabotaging its own enclave. Privilege Separation relies on additional security properties to function properly: least privilege, single enclave isolation, and secure data sharing [3]. SGX has commands which handle page protections which can be leveraged to implement this concept.

2.3.1.1 Least Privilege

When application code runs, it may only require a small portion of the dataset stored within the enclave. Hence, an approach to further secure the data is to partition it off into separate secure compartments within the enclave itself [3]. This allows the enclave to regulate which functions can access which portions of sensitive data parts of the application has access to.

2.3.1.2 Single Enclave Isolation

In order to enforce least privilege security principles within a single enclave, the enclave data needs to be portioned off in a secure manner [3, 29]. Securing the sensitive data using software-only techniques may prove mute if the enclave itself were to be compromised. Hence, hardware level techniques can be implemented to guarantee security of in-enclave compartments.

2.3.1.3 Secure Data Sharing

Least Privilege implementation cannot separate each privilege level completely. We must assume data will need to be shared between levels to accomplish certain tasks. To allow flexibility as application code requests secured content within an enclave, the enclave can dynamically adjust access privileges based on which parts of the application are requesting information [3].

2.3.2 Nested Kernel

Nested Kernels is a fairly young subject in research which explores separating a small portion of kernel code from the rest of the kernel [5]. This design makes gaining full control by attacking the outer kernel becomes more difficult. We find previous research mature enough that we believe it can be applied to an enclave security context. There are two main elements of the nested kernel architecture proposed that we wish to include in our design: a skeptical security policy and separate execution stack.

The nested kernel security policy is to assume the outer kernel is untrusted. Thus, to ensure the nested kernel data remains separate, it is initialized with read-only permissions on its own separate memory pages. In these pages are the sensitive instructions the nested kernel executes only upon verifying the outer kernel or memory has not been tampered with. To guard the nested kernel, entry and exit gates – portions of code which guard protected regions – are

included to adjust the permissions and interrupt to allow nested kernel code to execute without external interference. The entry gate will adjust permissions to allow the nested kernel to execute code safely, while the exit gate resets all permissions and settings.

When executing nested kernel code, a separate execution stack is initialized for nested kernel use. An execution stack is what normal kernels use for executing instructions sequentially. Should the nested kernel share the same stack as the outer kernel, the possibility for stack-based attacks such as a buffer overflow attack is left unchecked. If left unchecked, a successful buffer overflow attack could make the nested kernel vulnerable. In creating a separate stack, the outer kernel stack would be set aside and the nested kernel would use its own stack. This approach further ensures the nested kernel is separated from the outer kernel.

2.3.2.1 Privilege Separation between Kernels

In normal use, the kernel operates with ring 0 privileges (that is, the highest possible privileges on a computer). As discussed at length in previous works, this attribute makes the kernel a prime target for attackers. SGX already has a form of privilege separation based on its ability to lock information away from the kernel [4]. With this in mind, a nested kernel can be placed within the enclave to handle management of the application.

2.3.2.2 Managing Interactions with Application

One possible attack vector towards a nested kernel within an enclave is that of the application inside the enclave [6]. The application is assumed to be stable, though in reality the application can sabotage the nested kernel just as it can sabotage SGX. To protect against rouge internal application code, a method similar to that proposed by previous research can be implemented [3,5]. As described before, entry and exit gates establish a boundary between the

outside and the nested kernel. While the assembly code shown in previous research is viable, it must be modified to support SGX commands as well.

2.4 NesTEE LibOS Architecture

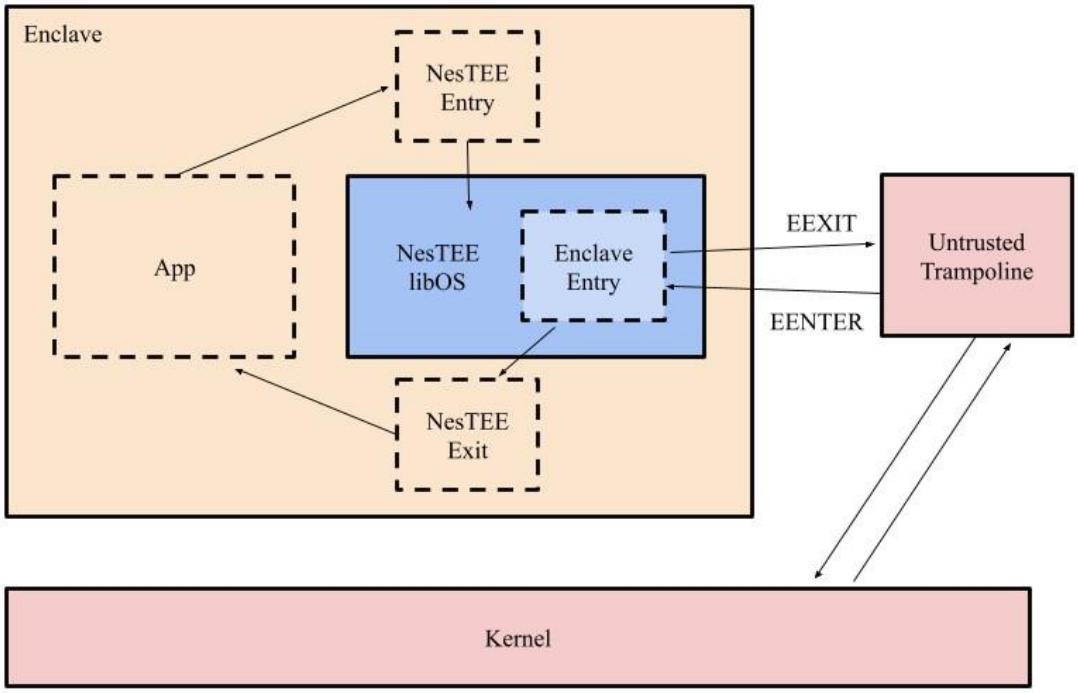

Figure 2.1: NesTEE LibOS Architecture Control Flow

Using principles from previous works, this section overviews the implementation of a nested kernel within an SGX enclave [3,4,5,17,18,19]. With the defined threat model in mind, NesTEE LibOS applies nested kernel and privilege separation principles by setting NesTEE LibOS as manager of enclave activity without interfering with SGX activity. Figure 2.1 overviews the control flow, describing interactions with the outer kernel using EEXIT and EENTER commands just as SGX normally would, which accounts for external threats [1]. To manage interactions between NesTEE LibOS and the application, NesTEE LibOS entry and exit gates are implemented to protect NesTEE LibOS from the application, addressing any internal threats.

2.4.1 Isolation of NesTEE LibOS Memory Pages

Separating the memory pages which the NesTEE LibOS data resides from the rest of the enclave contents is critical. Failing to do so could allow the internal application to modify the NesTEE LibOS data. To ensure NesTEE LibOS pages were separate upon initialization of the enclave, the linkerscript file had to be modified. A linkerscript decides how files are laid out in memory and converts their contents into an executable [20]. By modifying how NesTEE LibOS was laid out in memory, we ensure the memory is separate and aligned to its own page.

2.4.2 Memory Mapping Implementation

In order to properly implement the NesTEE LibOS design, the enclave would need to support dynamic memory mappings [21]. Without these functions, NesTEE LibOS wouldn't be able to map memory like a normal kernel should. Intel SGX normally does not support these functions within the enclave, so these functions had to be implemented in the SGX source code. Implementation of these functions involved leveraging the OCALL and IOCTL properties of the SGX to call mimic the mmap function using enclave creator functionality. This ensures that the enclave accepts and trusts the accolated NesTEE LibOS memory pages.

Regarding mmap, the function would require changes to trusted runtime system code. Within the enclave call code, trts_mmap is implemented to start page allocation using an OCALL and executes the EACCEPT function for each newly created page. Regarding the OCALL itself, the function creates pages and runs the Linux mprotect function to apply the proper read write and execution permissions. Since the functions of mmap and munmap are similar, the munmap implementation is very similar to that of mmap. The implementation of these functions also serves as an example of how NesTEE LibOS could be modified to add more enclave operations in the future.

2.4.3 Entry and Exit Gates

By taking the nested kernel approach, NesTEE LibOS is out of reach from a potentially hostile kernel ([1,5]). However, application code within the enclave can still potentially sabotage NesTEE LibOS by tampering with the memory pages which contain NesTEE LibOS' data. Success in modifying memory pages would render the benefits of NesTEE LibOS mute, just as it would to a normal SGX enclave. In addition to memory pages, a secure stack is required for NesTEE LibOS. If NesTEE LibOS ran using the user stack, commands in the user stack could potentially tamper with enclave functions. Thus, by separating the stacks, NesTEE LibOS commands can be executed securely. After NesTEE LibOS is no longer needed to perform certain actions, the user stack would be restored and the NesTEE LibOS memory pages would return to having read-only permissions. Failure to relock NesTEE LibOS upon exiting would leave the nested kernel vulnerable. A solution for this attack vector is the implementation of entry and exit gates which separates the stacks of the application from that of what NesTEE LibOS will be using.

2.4.3.1 Entry Gate Implementation

The implementation of the NesTEE entry gate is straight forward from a high level. The entry gate function must properly update the page permissions for NesTEE LibOS as well as set up a secure stack from which to work from. After both these actions have been taken, NesTEE LibOS actions can be taken.

If this were to be done using C functions, we would leave the process vulnerable to an attacker overwriting NesTEE LibOS memory through the use of a buffer overflow attack. To ensure this process is secure, the implementation must be hard-coded using assembly.

```
NesTEE_Entry:
  // Unprotect the NesTEE page
  MOV EMODPE, EAX
  MOV SECINFO_RWX, RBX
  MOV &NesTEE_Page, RCX
  ENCLU
  // Check ENCLU parameters
  CMP EMODPE, EAX
  JNE Crash
  CMP &NesTEE_Page, RCX
  JNE Crash
  // Set up secure stack
  MOV RSP, RBX
  MOV NesTEE_Stack, RSP
  PUSH RBX
  // Go to NesTEE libOS
  JMP NesTEE_LibOS_Start
```

Figure 2.2: NesTEE LibOS Entry Gate Pseudocode in Assembly

The implementation described in Figure 2.2 relies on several SGX commands. To change the page protections to read, write, execute (RWX), we employ the EMODPE command. EMODPE, an SGX CPU instruction, uses the supplied parameters in specified registers to adjust the access right to specified memory pages [1]. In the implementation, the NesTEE LibOS page permissions are changed from its default read-only permission to RWX. After, parameters are checked out of an abundance of caution. The register contents used to set up the EMODPE command are compared to the registers used to execute the command using the SGX CPU instruction ENCLU. Should any of them not match, the program will halt the process [2].

Following the change in page permissions, a secure stack is set up. Prior to executing NesTEE LibOS instructions, a separate secure stack is initialized using the malloc function. This stack is passed to the assembly function, which then adjusts the stack pointer and base pointers

accordingly. Once the stack is securely set up, the program switches to the NesTEE LibOS secure stack.

2.4.3.2 Exit Gate Implementation

```
NesTEE_Exit:
    // Protect the NesTEE page
    MOV EMODPR, EAX
    MOV SECINFO_R, RBX
    MOV &NesTEE_Page, RCX
    ENCLU
    // Check ENCLU parameters for EMODPR
    CMP EMODPR, EAX
    JNE Crash
    CMP SECINFO_R, RBX
    JNE Crash
    CMP &NesTEE_Page, RCX
    JNE Crash
    // Accept Page Changes
    MOV EMODPE, EAX
    MOV SECINFO_R, RBX
    MOV &NesTEE_Page, RCX
    ENCLU
    // Check ENCLU parameters for EMODPE
    CMP EMODPE, EAX
    JNE Crash
    CMP SECINFO_R, RBX
    JNE Crash
    CMP &NesTEE_Page, RCX
    JNE Crash
    // Restore user stack
    POP RBX
    MOV RBX, RSP
    // Return to caller
    RET
```

Figure 2.3: NesTEE LibOS Exit Gate Pseudocode in Assembly

The exit gate implementation is very similar to that of the entry gate. From a high-level description, one can compare Figure 2.2 and Figure 2.3 and notice the similarities. The program

flow is after restricting the NesTEE LibOS pages, the EACCEPT function must be executed so that the enclave accepts the page change. Regarding actual implementation, the exit gate is more complex. EMODPR requires more work as it can only be executed outside the enclave [1]. To implement properly for use with NesTEE LibOS, additional SGX source code had to be written that did not depend on untrusted kernel operations. As such, the first half of the implementation cannot be done using assembly alone. To address the potential security risks of using an OCALL to protect NesTEE LibOS, the source code only uses kernel calls when absolutely necessary.

For instance, the existing components for the SGX mprotect function was copied and modified to make a version specifically for NesTEE LibOS. The SGX mprotect execution flow was written generically to cover as many use cases as possible. When copying this code, the opposite approach was taken. By skipping some conditional statements and hardcoding portions of our NesTEE LibOS EMODPR code, we minimized our reliance on kernel functions when outside the kernel. After the OCALL is performed to lock the memory pages which contain NesTEE does program execution go back to inline assembly. After page changes are accepted and verified, the user stack is restored and execution returns to the internal application.

3. RESULTS

3.1 Importance of Evaluating Execution Performance

Evaluating performance of NesTEE LibOS is critical for examining feasibility of the prototype. While the prototype could work, overhead measurements are needed to determine validity. Hence, measuring execution time of NesTEE LibOS is necessary to determine what actions can be taken to further improve the prototype.

3.1.1 Experiment setup

We use an Intel NUC Kit NUC7CJYH desktop with a x86 64-bit Intel Pentium Silver J5005 4-core 1.5 GHz Intel CPU, 16 GB RAM paired with a 240 GB Kingston SV300S3 SSD. The OS used is Ubuntu 18.04.5 paired with Linux kernel 4.18.0-25-generic. Regarding the SGX versions, version 2.6 of the SGX Linux SDK and driver were used [26, 27].

3.2 Mprotect Functionality in NesTEE LibOS

The performance of the SGX mprotect function is necessary as to evaluate the time it would take for changes of the NesTEE LibOS page permissions. Given how NesTEE LibOS page permissions will be frequently updating, timing overhead of this function best describes the feasibility of NesTEE LibOS.

3.2.1 Method for Measuring Performance

To measure performance, it is necessary to simulate the process of memory pages being mapped, permissions altered, then unmapped. To accomplish this, mmap and munmap functions had to be implemented in SGX's source code since these actions require OCALLs. Regarding conditions for testing, we first allocated the maximum number of pages an SGX enclave would allow without crashing – that being around 128 pages. This memory allocation was then aligned

to its page boundary to reflect NesTEE LibOS' actual page layout. Following this, a timing function executes mmap, munmap, and mprotect functions and records the execution times using the standard C time library. It's important to note that when running the clock_gettime function, the CLOCK_MONOTONIC was used for higher precision [28]. After collecting all the times, the average is taken and the confidence interval for the data is computed as well.

Given the design of a regular SGX enclave, the internal C time library is considered untrusted. Hence, OCALLs had to performed whenever time needed to be recorded. To ensure our information was accurate, OCALL execution time was recorded and found to have negligible impact to the results.

3.2.2 Mprotect Performance Results

3.2.2.1 Changing Permissions to Read Only

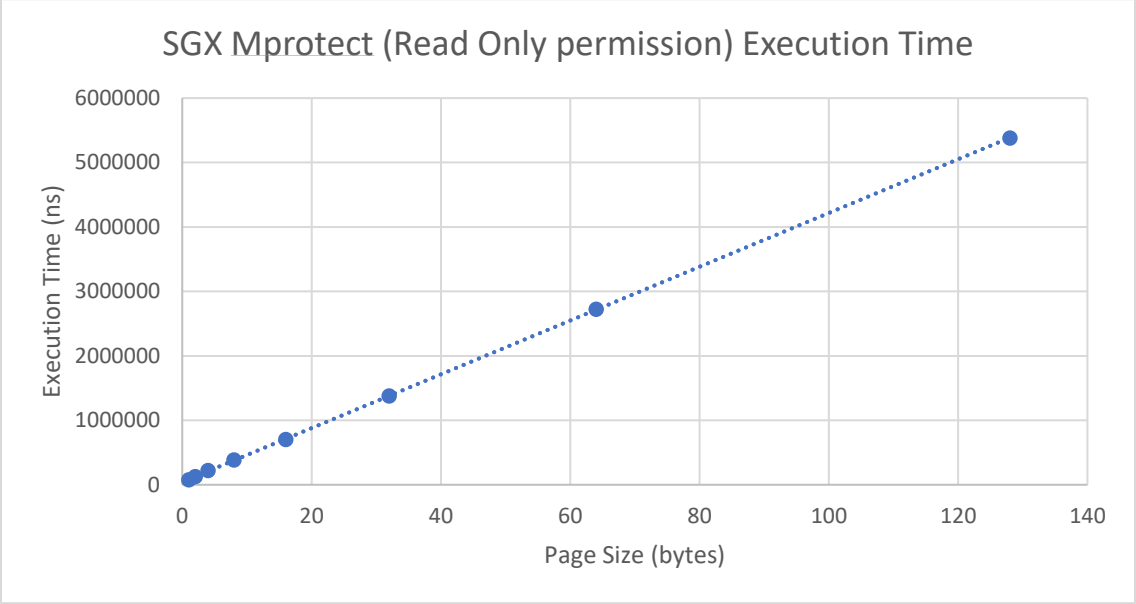

Figure 3.1: Mprotect Execution Time to Read Only Permissions

Table 3.1: Mprotect Execution Time Numbers to Read Only Permissions

Page Size	Time (ns)	Confidence Interval (ns)
1	77637	576
2	126773	701
4	220821	335
8	382452	207
16	703831	474
32	1376871	623
64	2725279	2048
128	5381173	4527

3.2.2.2 Change Permissions to RWX

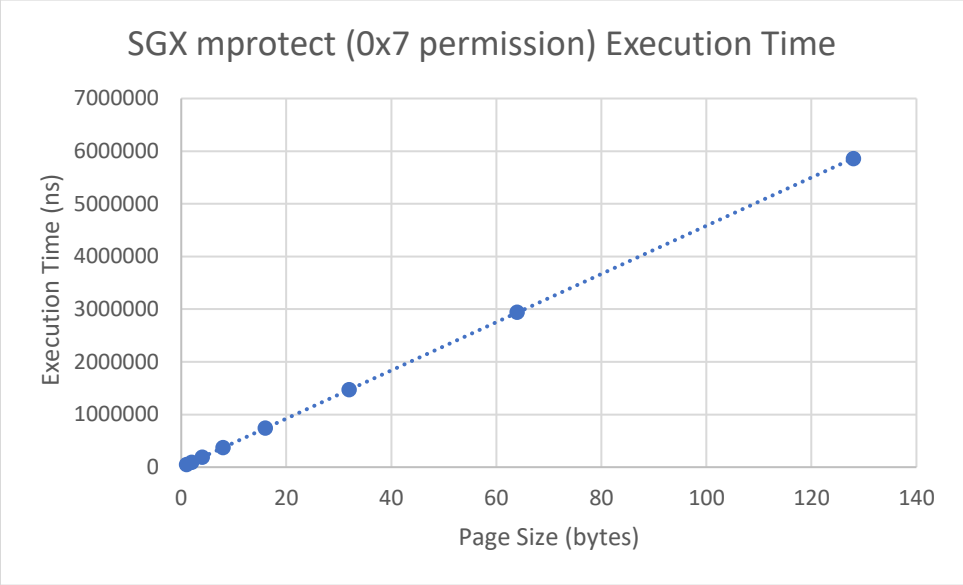

Figure 3.2: Mprotect Execution Time to RWX Permissions

Table 3.2: Mprotect Execution Time Numbers to RWX Permissions

Page Size	Time (ns)	Confidence Interval (ns)
1	46890	208
2	95078	212
4	187833	232
8	372960	328
16	739459	433
32	1473992	1015
64	2943258	1921
128	5860706	10879

The values gathered from timing mprotect show expected values. As expected, mprotect timings when changing from read-only to RWX took the longest, as more permissions settings had to be changed. Changing permissions from RWX to read-only also went as expected, normally being a faster process. In regards to the Linux counterparts to these functions, the timings are naturally larger. This is expected, given the SGX counterparts execute more steps to ensure enclave security remains intact in an OCALL. The linear increase in timing seen in Figure 3.1 and Figure 3.2 is also expected. A larger number of pages means mapping pages, changing their permissions, and complete unmapping would take longer, which mimics the Linux counterparts.

Observing the confidence intervals from both Table 3.1 and 3.2 leads us to conclude the data itself is reliable. Confidence interval measurements range from 208 ns on the low end to 10879 ns on the higher end. These values fall within a reasonable range, giving us further confidence in the readings.

3.3 NesTEE LibOS Evaluations

The purpose of implementing NesTEE LibOS to SGX is to enforce separation between the internal enclave application, the outer kernel, and SGX enclave functions. As such, implementing sample use cases of NesTEE LibOS are necessary to both demonstrate applications of NesTEE LibOS and performance. Currently, the entry gate code is fully implemented but the exit gate is only missing the proper EACCEPT implementation. Given that EACCEPT is merely an enclave function which checks and and accepts page permissions, it is assumed EACCEPT execution time is negligible in the evaluations.

3.3.1 NesTEE LibOS OCALL Applications

Enclave OCALLs are important features of the SGX enclave since they allow for interaction with the outer kernel. However, OCALLs can pose a security risk if used without oversight, especially if the kernel has been compromised. NesTEE LibOS addresses this flaw with its secure implementation. As such, NesTEE LibOS can act as a secure mediator between the enclave and the outer kernel. To demonstrate, a sample OCALL was performed using the standard SGX architecture and the NesTEE architecture.

The OCALL performed by both architectures simply wrote a statement to a text file, an action only possible with an OCALL. The standard SGX implementation consisted of the internal application code simply performing an OCALL. Contrarily, NesTEE LibOS performed the OCALL within the secured pages of the enclave. As such, performing an OCALL involved executing entry gate and exit gate functions.

Table 3.3: OCALL Use Case Performance

Architecture	**Time (ns)**	**Confidence Interval (ns)**
Standard SGX	95990	9490
NesTEE	131172	9759

By keeping the OCALL action simple, measurements better reflect the time each architecture took to execute the OCALL rather than the time taken up by the OCALL itself. Observing values in Table 3.3 reveals NesTEE LibOS adds an overhead of about 130% to a simple OCALL. This overhead is expected given NesTEE LibOS must open and restrict page permissions before returning control to the internal application.

3.3.2 NesTEE LibOS Allocator Applications

Allowing the internal application to allocate memory freely poses a vulnerability should the internal application be corrupted. Malicious allocations could be used to attack the enclave or the memory pages containing sensitive information could be overwritten and corrupted internally. Use of NesTEE LibOS removes this attack vector while adding greater protections towards the allocated memory. Through privilege separation, NesTEE LibOS memory pages are separate from the enclave memory pages, providing stricter access control and further protections to the allocated memory. To evaluate the cost of using NesTEE LibOS in a memory allocation context, two simple memory allocation sequences were set up. The standard SGX architecture was set to allocate memory as normal, while the NesTEE architecture could only allocate memory within the permitted memory pages inside NesTEE LibOS.

Table 3.4: Memory Allocation Use Case Performance

Architecture	**Time (ns)**	**Confidence Interval (ns)**
Standard SGX	9975	229
NesTEE	35640	208

The results shown in Table 3.4 show the NesTEE architecture is about 3 times slower than its standard SGX counterpart. While NesTEE LibOS does provide benefits in the form of protection and control over allocated memory, evaluations show this approach is can be expensive.

3.3.3 NesTEE LibOS Logging Applications

When running important tasks, programs will often produce a log of actions taken while completing a task. If anything goes wrong, a user or developer can intervene and fix the issue.

Given the importance of logging, there has been much subject over maintaining reliability [11, 13]. If an external source modifies the logs before they can be reported back to the user, the logs will be inaccurate [10, 12]. This could potentially lead to a user unaware of an attacker due to tampered logs. To enforce a pen-only policy regarding logs, NesTEE LibOS' separation from the rest of the enclave can keep logs secure. Only after execution within NesTEE LibOS completes are the logs returned and printed. To set up the evaluations, both architecture test cases consisted of pressing strings onto a buffer in the enclave. After control is handed back to the application the buffer contents are printed.

Table 3.5: Logging Use Case Performance

Architecture	**Time (ns)**	**Confidence Interval (ns)**
Standard SGX	10338	36
NesTEE	36741	211

Table 3.5 summarizes the performance for each architecture. In this use case, the NesTEE architecture adds about 350% greater overhead. We find that NesTEE LibOS is an expensive tradeoff for the security it provides. This finding makes sense given the greater complexity NesTEE LibOS contains to enforce stricter control over enclave activity.

4. CONCLUSION

This paper uses previous research over Nested Kernels [7], Privilege Separation [3], and work done by Intel [1] to introduce a modified version of Intel SGX. We provide a brief overview of SGX to establish the necessary context of why improvements are necessary. SGX security is limited due to the lack of memory isolation within the secure enclave. By missing this feature, SGX security remains limited. In addressing these issues, we present NesTEE LibOS, which promises to provide the groundwork for making SGX more applicable while maintaining security.

By going over the principles introduced in previous research over nested kernels and privilege separation we argue its potential to improve SGX to create a more robust security system. Security gate principles taken from nested kernel research prove applicable to SGX, which now protects the NesTEE LibOS from the internal application code. The secure stack idea also finds its way into NesTEE LibOS, protecting enclave instruction execution from the application and outer kernel. The placement of NesTEE LibOS data on its own memory pages further protects against unapproved modifications. These principles applied to SGX results in finer control of enclave behavior, while also protecting against several attack vectors.

Detailing the implementation of these principles in SGX code documents how these principles were translated into actual code and provides a low-level overview to how the overall proposed architecture works. Using SGX built-in commands to implement NesTEE LibOS functions reinforces the reliability of the overall architecture due to reuse of established code. In additional to implementing the architectural support, dynamic memory mapping was also implemented due to the need to perform benchmarks on page permission changes. These new

SGX functions also serve to demonstrate how additional functionality can potentially be added to NesTEE LibOS in the future.

Evaluations of NesTEE LibOS show the increase in complexity for greater control within the enclave comes at a cost. When comparing standard use cases in both architectures, we find the standard SGX architecture performs about 3 times more efficiently than NesTEE LibOS. It is certain the higher complexity principles NesTEE LibOS is based on is a source for overhead. However, the security features may be enough to justify the current overhead and provide motivation for future improvements.

REFERENCES

[1] "64-ia-32-architectures-software-developer-vol-3d-part-4-manual.pdf." Accessed: Feb. 01, 2021. [Online]. Available: https://www.intel.com/content/dam/www/public/emea/xe/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3d-part-4-manual.pdf.

[2] D. Sehr *et al.*, "Adapting Software Fault Isolation to Contemporary CPU Architectures," p. 11.

[3] M. S. Melara, M. J. Freedman, and M. Bowman, "EnclaveDom: Privilege Separation for Large-TCB Applications in Trusted Execution Environments," *ArXiv190713245 Cs*, Jun. 2020, Accessed: Feb. 01, 2021. [Online]. Available: http://arxiv.org/abs/1907.13245.

[4] V. Costan and S. Devadas, "Intel SGX Explained," 086, 2016. Accessed: Feb. 01, 2021. [Online]. Available: http://eprint.iacr.org/2016/086.

[5] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, Istanbul Turkey, Mar. 2015, pp. 191–206, doi: 10.1145/2694344.2694386.

[6] "Overview." https://sgx101.gitbook.io/sgx101/sgx-bootstrap/overview (accessed Feb. 02, 2021).

[7] E. I. Organick. *The Multics System: An Examination of Its Structure*. MIT Press, Cambridge, MA, USA, 1972.

[8] J. H. Saltzer. Protection and the control of information sharing in multics. *Commun. ACM*, 17(7):388–402, July 1974.

[9] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[10] J. Kong. *Designing BSD Rootkits*. No Starch Press, San Francisco, CA, USA, 2007.

[11] N. Honarmand, N. Dautenhahn, J. Torrellas, S. T. King, G. Pokam, and C. Pereira. Cyrus: Unintrusive application level record-replay for replay parallelism. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 193–206, New York, NY, USA, 2013. ACM.

[12] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. *In Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, pages 255–264, New York, NY, USA, 2002. ACM.

[13] N. Honarmand and J. Torrellas. Replay debugging: Leveraging record and replay for program debugging. In Proceeding of the 41st Annual International Symposium on Computer Architecuture, ISCA '14, pages 445–456, Piscataway, NJ, USA, 2014. IEEE Press.

[14] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy at Proceedings of the ACM IEEE International Symposium on Computer Architecture* (ISCA), 2013.

[15] McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C. V., Shafi, H., Shanbhogue, V., and Savagaonkar, U. R. Innovative Instructions and Software Model for Isolated Execution. In *Proc. Hardware and Architectural Support for Security and Privacy* (2013).

[16] Lind, J., Priebe, C., Muthukumaran, D., O'keeffe, D., Aublin, P.-L., Kelbert, F., Reihere, T., Goltzsche, D., Eyers, D., Kapitza, R., Fetzer, C., and Pietzuch, P. Glamdring: Automatic Application Partitioning for Intel SGX. In USENIX ATC (2017).

[17] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 233–247, Washington, DC, USA, 2008. IEEE Computer Society.

[18] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. *In Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 335–350, New York, NY, USA, 2007. ACM

[19] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-VM monitoring using hardware virtualization. In *Proceedings of the 16th ACM Conference on Computer and*

Communications Security, CCS ’09, pages 477–487, New York, NY, USA, 2009. ACM.

[20] “Scripts (LD).” Accessed: Apr. 08, 2021. [Online]. Available: https://sourceware.org/binutils/docs/ld/Scripts.html.

[21] C.-C. Tsai, D. E. Porter, and M. Vij, “Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX,” p. 14.

[22] C. Cowan, F. Wagle, Calton Pu, S. Beattie and J. Walpole, "Buffer overflows: attacks and defenses for the vulnerability of the decade," Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00, Hilton Head, SC, USA, 2000, pp. 119-129 vol.2, doi: 10.1109/DISCEX.2000.821514.

[23] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation* (OSDI), pages 267–283, 2014.

[24] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O’Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure linux containers with Intel SGX. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation* (OSDI), Nov 2016.

[25] S. Shinde, D. L. Tien, S. Tople, and P. Saxena. PANOPLY: Low-TCB Linux applications with SGX enclaves. In *Proceedings of the Network and Distributed System Security Symposium* (NDSS), 2017.

[26] Intel Corporation. Intel software guard extensions for Linux OS - Intel SGX driver. https: //github.com/01org/linux-sgx.

[27] Intel Corporation. Intel software guard extensions for Linux OS - Intel SGX SDK. https: //github.com/01org/linux-sgx.

[28] “clock_gettime(3): clock/time functions - Linux man page.” https://linux.die.net/man/3/clock_gettime (accessed Apr. 08, 2021).

[29] Zheng, W., Dave, A., Beekman, J. G., Popa, R. A., Gonzalez, J. E., and Stoica, I. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *USENIX NSDI* (2017).

[30] Seo, J., Kim, D., Cho, D., Kim, T., Shin, I., and Jiang, X. FlexDroid: Enforcing In-App Privilege Separation in Android. In NDSS (2016)