Pagoda: Towards Binary Code Privacy Protection with SGX-based Execute-Only Memory

Jiyong Yu
University of Illinois at Urbana-Champaign
jiyongy2@illinois.edu

Xinyang Ge[*]
Databricks
xinyang.ge@databricks.com

Trent Jaeger
Penn State University
trj1@psu.edu

Christopher W. Fletcher
University of Illinois at Urbana-Champaign
cwfletch@illinois.edu

Weidong Cui
Microsoft Research
wdcui@microsoft.com

***Abstract*—Code disclosure remains a huge threat to the intellectual property (IP) of any software that is deployed in a remote, untrusted environment. In this threat model, attackers have complete control over the software stack, so software-only solutions for preventing code disclosure have been doomed to fail. A natural alternative is to employ trusted hardware, e.g., an enclave-based architecture such as Intel SGX. However, existing SGX frameworks assume the target application is in the trusted computing base, i.e., free of vulnerabilities which can be exploited to leak code. Making matters worse, simply porting to an enclave-based paradigm is impractical for enterprise-scale applications, incurring large performance overheads and compatibility issues.**

In this paper, we take a first step towards building a practical, SGX-based code privacy enforcement framework called Pagoda that supports unmodified applications with minimal performance overhead. The key insight of Pagoda is that placing only application code within the enclave prevents arbitrary code accesses, and at the same time avoids the usual performance and compatibility issues stemming from protecting data within enclaves. Pagoda achieves code privacy throughout the application's lifetime, by loading and decrypting encrypted binaries into the enclave, and enforcing eXecute-Only-Memory (XOM) to block arbitrary accesses to the private code during its execution.

We have built a prototype of Pagoda for Linux-based systems on Intel SGX. The performance evaluation on SPEC CPU2017 benchmarks shows that Pagoda incurs an average of 2.1% performance overhead when compared to native runs. To demonstrate its compatibility, we show that Pagoda can run a wide range of applications, from common server applications such as Lighttpd and Memcached, to complicated graphical applications such as Quake without any source code modification.

I. Introduction

Today's enterprise-scale applications are rife with intellectual property, such as proprietary business logic, analytic functions and algorithms [4]. This has made *code disclosure* attacks a huge industry-wide problem. For example, software piracy [47] whereby a user (pirate) purchases and creates unauthorized copies of proprietary software, causes the software industry to lose billions of dollars every year [3], [9]. In the server space, many small businesses rely on third-party cloud service providers (CSPs) to host their cloud applications, again putting their proprietary code at risk in the face of untrusted cloud providers [14], [52].

At the same time, enforcing code privacy for enterprise-scale applications is non-trivial given that the software in question is often used remotely, and on potentially untrusted platforms. In such a setting, the attacker typically has complete control of the software stack including OS, peripherals, etc. This renders software-only solutions (such as code obfuscation [2], [25], [29], [44]) ineffective, and seemingly necessitates a hardware-based approach to achieve high assurance. For example, Intel SGX [41] provides an *enclave* abstraction to enforce confidentiality/integrity of sensitive code/data from an otherwise untrusted stack. This seemingly solves the problem: to hide sensitive code, run it inside an enclave.

Despite their strong security guarantees, however, enclave-based architectures such as SGX face difficulties in being applied to the code disclosure problem. To start, the enclave programming model often necessitates significant software refactoring, which makes it impractical to use in conjunction with existing large-scale applications. Although several enclave/SGX frameworks have been proposed to support existing applications with no/slight modifications [20], [46], [51], [53], those frameworks are mainly designed for protecting data (not code), suffer from performance issues [50], and are only compatible with applications that require very limited system service support (e.g., command-line applications). In addition, existing frameworks assume that enclave code is a part of the trusted computing base. In other words, the application must be free of bugs and latent vulnerabilities that could be exploited to undermine code privacy. Given the nature of the applications needing protection, this is not realistic [42]. Further, existing attacks such as DarkROP [21], [38] show how it is indeed realistic to launch code disclosure attacks in the enclave-based private-code setting.

A. This Paper

We take a first step towards building a practical, high-performance enclave-based framework that focuses on code privacy enforcement. We call this framework Pagoda and implement it on top of Intel SGX. Our key observation is that, while enclave-based architectures such as SGX were designed to defend both code and data, there are significant performance- and compatibility-related advantages in protecting code alone. Since our goal is code privacy, we can reap these benefits without suffering the usual and obvious pitfalls of not protecting data.

[*]The work was done while the author worked at Microsoft Research.

Based on this observation, and contrary to every prior proposal built on SGX, we design Pagoda to load *only* the application code into the enclave while leaving all the application data, including globals, the heap and even the stack, outside of the enclave. This design allows Pagoda to support all system calls and user-mode DMA *natively* because the data memory is accessible to the host OS in the same manner as an ordinary user-mode process. Further, this design also avoids I/O performance overheads as the communication between the application and the host OS no longer requires expensive data copying across the enclave boundary. Finally, this design avoids enclave memory paging overheads, since the (large) application data memory is not loaded into the (small) enclave protected memory.

Pagoda uses SGX to isolate the private binary code inside the enclave, and ensures that the private binary code is always encrypted while outside the enclave (the latter is similar to [19], [35], [36], [54]). Beyond these base protections, Pagoda must address several challenges to achieve code privacy in the face of program bugs and memory-safety vulnerabilities. For example, prior work such as DarkROP [38] shows how an adversary can leak enclave code using ROP without prior knowledge of that code. Conceptually, this amounts to the adversary controlling the parameters of, and calling, a `memcpy` gadget so as to trick the enclave into dumping itself into non-enclave memory. That Pagoda maps application data outside the enclave further simplifies such attacks.

Pagoda blocks this direct disclosure of plaintext private code inside an enclave by changing the enclave code page permissions to *execute-only* immediately after the encrypted private code is loaded into the enclave and decrypted. The code privacy of Pagoda therefore reduces to maintaining said execute-only memory (XOM) protection throughout the application's execution. To revert the execute-only protection of code pages, the adversary must execute a special enclave operation `EMODPE`. Pagoda prohibits `EMODPE` with a software-only defense for single-threaded applications. For multi-threaded applications, a race condition could enable adversary to bypass the software-only defense. So, for that case, we propose a simple hardware change to SGX (implementable as a microcode update).

We evaluate a prototype of Pagoda on Ubuntu 18.04 running on a SGXv2-capable Intel CPU. To demonstrate its compatibility, we show that Pagoda is able to run different types of applications, including complex graphical applications such as games without any source-code modification. The performance evaluation shows that Pagoda only adds 2.1% overhead over a native run for SPEC CPU2017. We also evaluate Lighttpd and Memcached and show Pagoda decreases the peak throughput by around 60% over vanilla Linux. For the evaluated games, Pagoda causes the average frames-per-second (FPS) rate to drop 8.4% compared to the native runs.

II. Background

A. Intel Software Guard Extensions (SGX)

Intel SGX [16], [41] provides protection for the integrity and confidentiality of (a portion of) a user-level application even when the privileged system software such as the OS/hypervisor is malicious. This is achieved via a trusted execution environment called *enclave*. Enclaves have exclusive access to their memory content, which can also be attested to verify the authenticity of both the enclave application and the underlying hardware. Other hardware vendors have proposed enclave variants with similar security guarantees (e.g., AMD SEV [49], ARM TrustZone [15], RISC-V Keystone [37]), but we focus on Intel SGX because of its wide commercial availability.

Enclave Operations SGX is exposed to software developers through two instructions: `ENCLS` for kernel-level enclave operations, and `ENCLU` for user-level enclave operations. The actual enclave operation performed by the two instructions depends on the "leaf index" stored in the `rax` register. The SGX programming model requires the host OS to allocate resources and initialize each enclave. Each enclave has one pre-defined, immutable entry point named `OENTRY`. The user-mode code can then enter (`EENTER`) the enclave, execute code within the enclave starting from the instruction at `OENTRY`, and exit (`EEXIT`) the enclave.

Hardware-based Isolation Enclaves are isolated from non-enclave software with a dedicated DRAM region, a subset of which—the Enclave Page Cache (EPC)—stores the enclave code and data. Enclave software can allocate enclave code and data inside a contiguous region in its virtual address space called ELRANGE, which gets mapped to the EPC. Code and data within an enclave is encrypted and integrity protected from the rest of the system.

Remote Attestation Remote attestation proves to a trusted remote party that the enclave is initialized in an expected state on a legitimate SGX hardware. To perform remote attestation, the enclave software invokes the `EREPORT` operation (`ENCLU` with `rax`=0) to generate a *report*. A report contains a measurement of the entire enclave state when the enclave is initialized, and a signature. The report is sent to the remote party by the non-enclave software.

Enclave Exceptions An Asynchronous Enclave eXit (AEX) occurs when an interrupt/exception happens during the enclave execution. Executing enclave-prohibited instructions like system calls, or instructions outside the enclave memory are common causes of AEX. Upon an AEX, SGX hardware saves the current processor context (including the current instruction pointer `rip`) into a per-thread, in-enclave data structure, called the State Save Area (SSA), and then transfers the control to the system's exception/interrupt handler with a non-secret, synthetic register context. The addresses of the SSAs are stored in the thread control structure (TCS) inside the enclave. Each enclave thread has its own TCS. The enclave execution can be resumed from AEX with a user-mode `ERESUME` operation.

Configuring Enclave Page Permission at Runtime While set prior to enclave initialization, enclave page permissions can be adjusted at runtime (only on SGXv2). An OS kernel can restrict enclave page permissions with `EMODPR` (`ENCLS` with `rax`=14). For security, SGX only allows the enclave to relax its own page permissions, by invoking `EMODPE` (`ENCLU` with `rax`=6) during the enclave execution.

B. Deploying Encrypted Code Using SGX

Earlier work [19], [35], [36], [54] uses encryption to protect secrets in enclave binaries. Since Pagoda also benefits from code encryption to partially fulfill the code privacy goal, we describe the procedure of deploying SGX applications with encrypted code introduced by the earlier work.

The software developer encrypts the code binary before distributing the application to end users. An SGX machine is necessary for the end user to validate (through remote attestation) and use the application. A special in-enclave loader is responsible for loading the binary into the enclave, and performing the remote attestation, as described in §II-A, to attest to a remote trusted verification platform (e.g., the software developers themselves) that the enclave is initialized in the correct state on a certified SGX platform. Once the remote attestation succeeds, the in-enclave loader establishes a secure communication channel with the remote platform to receive the decryption key, and decrypts the encrypted code inside the enclave. With the code now decrypted, the loader can finalize the program loading and transfer control to the application.

III. Threat Model

Pagoda assumes an adversary who owns the system that runs the application. Such an adversary can be a desktop user who purchases and uses the application on their own machine, or a third-party cloud provider who hosts proprietary cloud software for other companies. The adversary has arbitrary control of the system, including the BIOS settings, the hardware peripherals, any privileged software such as the OS or the hypervisor, and any unprivileged software components without the SGX enclave protection.

Pagoda aims to provide code privacy for existing applications with no source code changes. Specifically, Pagoda prevents the adversary from accessing (reading/writing) instructions in the private code binary in plaintext form. Preventing reading/writing private code must be guaranteed throughout the binary's lifetime, including when code is *in transit* (transferred from the developer to the untrusted platform), *at rest* (stored in the storage of the untrusted platform), and *in use* (executed on the untrusted platform). Other types of code leakage, such as indirectly inferring executed instructions via observing their impact on the process state or microarchitectural state, which we discuss in §VII-A, or breaking the schemes used for code encryption, are out of scope.

Pagoda trusts SGX to operate as described in §II. Pagoda also trusts the application developer, and the verification platform involved in the SGX remote attestation process. Pagoda does not assume that the unmodified application code is bug-free, i.e., any attacks capable of exploiting vulnerabilities in SGX programs are possible.

Since the goal is code privacy, Pagoda does not protect the application's data. We discuss how Pagoda can be further extended for data protection in §VII-B. Pagoda also does not guarantee the integrity of the program's execution, nor the absence of denial-of-service attacks. However, Pagoda must guarantee that undermining integrity cannot lead to unauthorized read/write access to the private code.

IV. Pagoda Design

A. Key Idea

In this section, we describe the design of Pagoda by focusing on the security mechanisms required for its code privacy goal and how it supports unmodified applications running within SGX enclaves. Our key observation is that, while SGX was designed to defend both code and data, there are significant performance- and compatibility-related advantages in protecting code alone. Further, since our goal is code privacy, we can reap these benefits without suffering the usual and obvious pitfalls of not protecting data.

Putting the above together, as Figure 1 demonstrates, Pagoda maps all application data outside of the protected enclave memory and places only private code, including the code from the main application and the code from any private shared (dynamic) libraries plus a small trusted runtime, inside the enclave memory. Since application data is visible to the host software and the OS, this enables the entire system call interface and user-mode operations like DMA to be supported natively. Further, storing data outside of the enclave memory avoids the need to copy data into/out of the enclave memory and eliminates paging overheads stemming from the limited enclave memory.

Based on this key idea, for the rest of the section, we present the design of Pagoda by describing how it achieves code privacy throughout the lifetime of an application (§IV-B), how it maintains compatibility with unmodified applications with the support of dynamic linking/loading for shared libraries (§IV-C), and how it supports communication across the enclave boundary (§IV-D).

B. Code Privacy Protection with XOM

Figure 2 shows the important steps during the program loading process of Pagoda that establish code privacy protection for the target application. Pagoda initializes the enclave with the enclave entry point `OENTRY` pointing to the Pagoda trusted runtime. The in-enclave Pagoda loader, as part of the Pagoda trusted runtime, performs the program loading procedure and launches the application binary.

As a first step towards code privacy, Pagoda uses the code encryption mechanism proposed by prior work (§II-B). That is, all binaries with private code are distributed with the code encrypted. After entering the enclave, the Pagoda loader performs remote attestation, and receives all keys necessary for decrypting encrypted binaries from the trusted remote party.

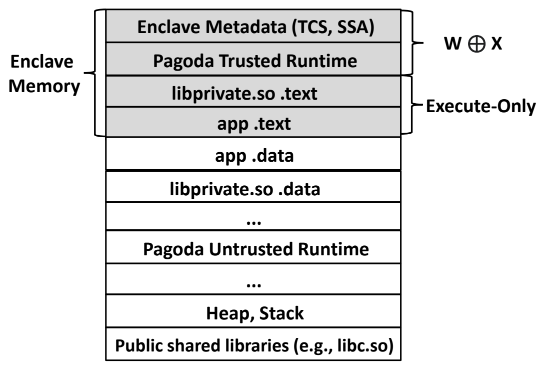

Fig. 1: Memory layout of application using Pagoda. Shaded gray color indicates the enclave memory region ELRANGE, which only covers up to the application code segments. libprivate.so represents private dynamic libraries.

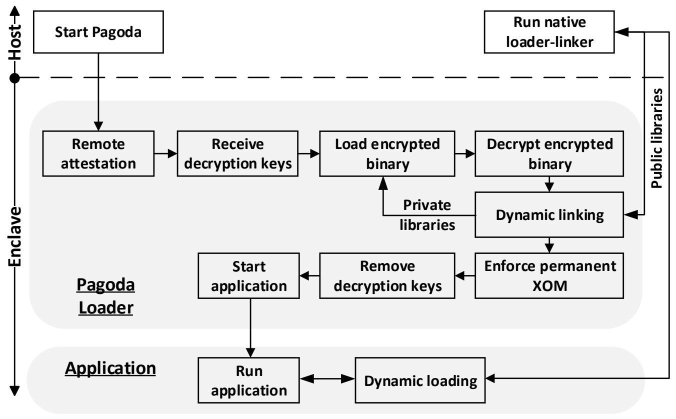

Fig. 2: Different actions that Pagoda takes during program loading for code privacy protection.

Then, the encrypted application file is copied from outside the enclave, and decrypted into plaintext format. Private dynamically libraries can be loaded and decrypted in the exact same manner later during dynamic linking, as shown in Figure 2.

When loading the main application binary, Pagoda maintains its original binary memory layout (where code and data are stored contiguously), and at the same time only places the code segment into the enclave while leaving the data segment outside. Pagoda calculates the base address of the application binary image such that the end of the code segment matches the end of the ELRANGE, as shown in Figure 1. Dynamically linked libraries require a different (non-standard) memory layout. Since it is not possible to layout their code and data segments contiguously (subject to the constraint that only the code segment is stored within enclave memory), their code and data segments are mapped non-contiguously. We further explain the mapping process of private libraries in §IV-C.

The solution so far is sufficient to provide basic isolation from non-enclave software. First, encryption and remote attestation guarantee that code in transit/at rest is only visible as ciphertext from outside the enclave. Non-enclave software can neither access the decryption keys inside the enclave, nor receive the keys from the remote party without performing the enclave remote attestation. Second, the enclave memory isolation prevents any direct accesses from non-enclave software to the private code inside the enclave (code in use).

Code encryption/secure distribution (§II-B) is not sufficient for code privacy, however, when the target application contains bugs/latent memory-safety vulnerabilities [42]. These vulnerabilities can enable non-enclave software to control enclave execution in unintended ways. Although Pagoda by default enforces $W \oplus X$ to block direct code injections, DarkROP [38] and several similar attacks [21], [24] demonstrate how simple memory vulnerabilities like stack buffer overflows can enable the adversary to blindly search for gadgets in the enclave code, such as ROP gadgets and arbitrary read/write gadgets, in a way similar to the classic Blind ROP attack [22]. The fact that Pagoda places the application's data – including stack and heap – outside the enclave further exacerbates such attacks, since it enables the adversary to trivially and arbitrarily manipulate said data.

The above introduces challenges in protecting both code in use and code in transit/at rest, respectively. We now discuss the problems and their solutions.

1) Protecting Code in Use: First, the privacy of code in use is undermined as the adversary can craft *code disclosure attacks* with gadgets capable of dumping any readable enclave code pages to non-enclave memory. To mitigate those attacks, Pagoda must prevent direct reads/writes to said pages from even enclave instructions, which implies enforcing execute-only memory (XOM) protection on all private enclave code pages. Although multiple implementations of XOM have been proposed, they either require the support from privileged software [18], [28], [31], which is insecure under our attack model, or use expensive code instrumentation techniques such as software-fault isolation (SFI) [23], [48]. Luckily, SGX provides hardware support for directly configuring enclave pages as execute-only with negligible performance overhead. We apply this enclave capability and mark all private enclave code pages as execute-only before transferring control from the trusted runtime to the application, when the adversary has the opportunity to launch the attack.

While XOM mitigates the code disclosure attack in a straightforward manner, enforcing the XOM property permanently (i.e., throughout the lifetime of the enclave) faces an additional problem: SGX also supports enclave page permission relaxation through EMODPE operation, which is an ENCLU instruction with rax=6 (§II-A). Although a genuine Pagoda execution never uses ENCLUs as EMODPE, unfortunately, with code reuse attacks, an adversary may abuse existing ENCLU instructions in the enclave code. The adversary could craft a ROP chain with an ENCLU in the end to form an EMODPE operation and grant read/write permission to an execute-only enclave code page. Note that Pagoda needs to enforce $W \oplus X$, which also demands the absence of EMODPE.

Therefore, Pagoda must prohibit the use of EMODPE to enforce the XOM protection permanently before starting the application. Depending on whether the application is single-threaded or multi-threaded, Pagoda applies different approaches to prohibit the use of EMODPE, before transferring control to the application. In §V, we explain why the entire scheme so far achieves the security goal of Pagoda, and introduce the two approaches to eliminate EMODPE for single-/multi-threaded applications, respectively.

2) Protecting Code in Transit/at Rest: Malicious enclave memory accesses/gadgets also jeopardize the privacy of both the code in transit and the code at rest if such accesses obtain the private code decryption keys.

We adopt the following two approaches to ensure that the decryption key is unreachable after the control is transferred to the application. First, Pagoda erases the decryption key left in the enclave memory after the code decryption finishes. Second, every time Pagoda is invoked to run a protected application, Pagoda can only receive decryption keys from the trusted remote party once, after the remote attestation. In this way, even if the adversary corrupts enclave execution and repeats the remote attestation and key request process, the trusted remote party will simply reject those requests. Consequently, the remote party must differentiate invocations of Pagoda on the same application. To solve this problem, Pagoda uses a *nonce* at a given location in the initialized enclave memory space. The value of the nonce varies during each invocation of Pagoda on a given application. Since the enclave measurement used in remote attestation also covers this nonce, the remote party can be notified whether the key request is from a different enclave.

Notice that the precondition for the approaches above is that, all encrypted binaries, including the main application and the private libraries, have to be decrypted before starting the application. §IV-C explains how Pagoda achieves this requirement for the private libraries.

C. Dynamic Linking/Dynamic Loading

Dynamic linking takes place during the loading of the main application, right before Pagoda enforcing the permanent XOM protection and disallowing accesses to decryption keys, as shown in Figure 2. Pagoda supports dynamic linking with both public and private libraries. After the loader identifies a library to dynamically link to, it first checks whether the library is private. Public libraries, such as libc, are loaded outside the enclave by directly calling the native loader-linker outside of the enclave. Private libraries, which Pagoda assumes are encrypted and shipped together with the main application, follow the same loading procedure as the main application. If the private libraries are encrypted using the same key as the main application, the loader can skip the remote attestation and the key receiving steps and reuse the previous key. Otherwise, the loader must request new keys from the remote trusted party. The XOM enforcement is also applied to the code of private shared libraries.

The only difference between the loading of the main application and private libraries, as Figure 1 illustrates, is that Pagoda splits the private library image into code half and data half, and only maps the code half inside the enclave memory. Such code-data split complicates the program loading, because binaries are normally compiled with an assumption of a contiguous memory layout. Mapping the code half and the data half separately breaks instructions that assume a fixed distance between the code half and the data half. In x86, such instructions are instructions with code-to-data references using RIP-relative addressing. Specifically, these instructions use RIP+offset to denote the address of the data, with RIP indicating the address of the instruction, and offset being an immediate value representing the original distance between the data and the instruction. Since the code-data split expands the distance between code and data, all offset values must be adjusted with the additional distance induced by the code-data split. Pagoda expects the developer of the private library to provide a file listing the addresses of all RIP-relative offsets. The developer can either rely on the compiler to emit the location of the offsets, or disassemble the binary file and search for all the occurrences. The developer must encrypt and distribute the file together with the private library files.

Unmodified applications may also opt for dynamic loading to load shared libraries in the middle of the application execution. However, dynamic loading for private libraries is not supported since the decryption key is no longer obtainable after the application starts, as mentioned in §IV-B. For private libraries that require dynamic loading, Pagoda expects the application developer to specify all of those libraries, and Pagoda will locate and load them during dynamic linking. After the private library is loaded successfully, the loader generates a descriptor containing the information about the library, such as its base address. When the application uses dynamic loading to load the library (e.g., through dlopen()), this descriptor is returned. Dynamically loading public libraries is handled by the native loader-linker.

D. Cross-Boundary Communication

The execution of the application involves interaction with the public libraries and the operating system outside the enclave. However, directly calling public library functions or making system calls will abort the enclave execution as branch instructions are not allowed to cross the enclave boundary. This section describes how Pagoda facilitates cross-boundary calls.

1) Enabling Cross-Boundary Calls: The upper half of Figure 3 illustrates how Pagoda transforms exceptions caused by prohibited cross-boundary calls into successful system calls or public library function calls. Since the procedures of enabling system calls and public function calls are identical, we focus on how Pagoda handles system calls as follows.

System calls are classified as illegal instructions inside the SGX enclave. Consequently, an enclave exception (AEX) will happen when the application makes a system call directly from within the enclave. To achieve compatibility with unmodified applications, Pagoda allows this AEX to occur (without any change to the system call instructions inside the enclave) and performs the system call outside the enclave, finally resuming enclave execution. By design, all the data memory needed to service a system call resides outside of the enclave and hence is already accessible to the host, so Pagoda only needs to

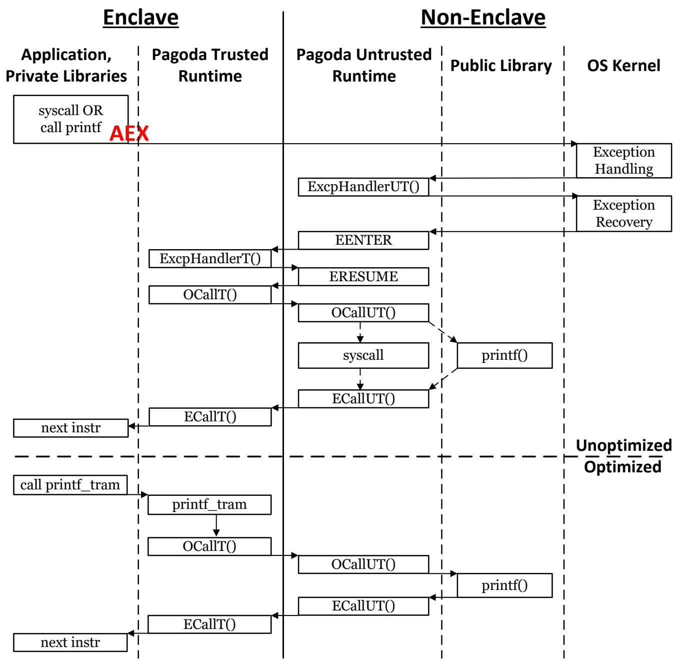

Fig. 3: The upper half shows how Pagoda enables system calls and public library functions calls based on the exception handling. The bottom half shows the optimized public library function call handling, with only two enclave boundary crossings.

additionally expose the parameters stored in the registers such as the system call number stored in `rax`, instead of performing a deep copy for related data structures.

When executing a system call, the enclave triggers an AEX and is immediately trapped into the OS kernel. Next, the OS invokes the exception handler `ExcpHandlerUT` registered by Pagoda. `ExcpHandlerUT` recognizes the exception is from the enclave, and forwards the exception to its counterpart `ExcpHandlerT` in the enclave. `ExcpHandlerT` needs to redirect the control flow of the enclave execution to an external system call site while preserving the system call parameters stored in the registers. To do so, `ExcpHandlerT` modifies the instruction pointer saved in SGX's state save area (SSA) to an in-enclave stub called `OCallT` before resuming the enclave execution with `ERESUME`. `ExcpHandlerT` also derives the address of the next instruction after the system call, and saves the address for continuing the application's execution after the system call handling completes. `ERESUME` restores the register context back to the state when the system call is invoked, except for the instruction pointer, which points to the `OCallT`. The `OCallT` function and its counterpart `OCallUT` form a typical OCALL procedure, allowing the program to proactively exit the enclave via `EEXIT` and land on the target address, which, in this case, is a system call instruction inside the untrusted runtime. Once the system call completes, the control is transferred to an `ECallUT` routine. `ECallUT`, together with its counterpart `ECallT`, forms a typical ECALL procedure that transfers the control back into the enclave. The `ECallT` function eventually jumps to the previously stored address of the next enclave instruction after the system call.

Pagoda's handling of public library function calls follows the same procedure as the system calls handling. Note that the external library can also call into an enclave-protected function, with a process symmetric to what is described above: the direct call triggers an exception. This exception is properly handled and leads to an ECALL process that reaches the target enclave function. The control returns back to the non-enclave program through an OCALL process.

2) Optimizing Public Library Function Calls: Cross-boundary calls based on exception handling are expensive. Specifically, a single call takes a total of six enclave boundary crossings and four ring crossings, so the overhead can significantly impact the overall performance when calls across enclave boundary are frequent. Ideally, each function call should only take two enclave crossings, one for the call and the other for the return, with no ring crossing/exception. As Figure 3 (upper) illustrates, the exception with the subsequent exception handling process is the culprit of the redundant boundary crossings. To eliminate the exception and achieve the ideal two enclave crossings, we apply the following optimization as shown in the bottom half of Figure 3.

First, during relocation, Pagoda patches the function pointers for each imported library function (e.g., the global offset table in ELF) so that they point to a trampoline function instead of the actual target function in the external library. These per-function trampolines are generated as part of the Pagoda trusted runtime when the loader resolves the dynamically-linked dependencies for the application. Each trampoline saves the address of the actual target function onto the program stack and calls the `OCallT` stub. The OCALL process allows the control to reach the target function with only one enclave boundary crossing. Once the target function returns, `ECallUT` receives the control, and activates the ECALL process. Note that since the per-function trampolines are generated by the loader, only the function pointers known to the loader can be patched, such as public library functions imported to the application. Pointers to non-enclave functions that are generated dynamically at runtime cannot benefit from this optimization.

3) Multi-threading: Pagoda provides multi-threading support for running unmodified programs inside the enclave, in addition to delegating the `clone` system call to the non-enclave software. As mentioned in §II-A, each thread must have its own TCS before it can enter the enclave. Pagoda users can specify the (maximal) thread number and Pagoda will create the TCSs before initializing the enclave. When the application spawns a new thread, Pagoda intercepts the `clone` system call and assigns a pre-allocated TCS to the new thread. Note that Pagoda currently does not support forking child processes for multi-process applications. We believe Pagoda can be extended to support `fork` similar to prior work [51], [53].

V. Security

Pagoda's security goal is to prevent arbitrary read/write accesses to the private application and library code. This security goal is partially fulfilled by protecting the code in transit and at rest with code encryption and carefully handling decryption keys discussed in §IV-B. Therefore, this section mainly focuses on how Pagoda protects private code in use, by satisfying the following property throughout the application's execution:

Property 1. *Private code is mapped to a fixed set of enclave code pages and remains execute-only.*

Recall that the private code pages are only readable and writable during the program loading and before the application starts execution. Since the loading is handled entirely by the loader inside the trusted runtime, we carefully engineer the loader code, such that 1) it has no exploitable vulnerability, and 2) it fully resides within the enclave memory. Therefore, it is reasonable to assume the loader is trustworthy. Using a trusted loader, Pagoda establishes Property 1 before the application starts, by configuring all enclave pages as execute-only, and all other pages as non-executable, according to §IV-B.

§IV-B highlights that the key point in enforcing Property 1 throughout the entire execution of the application is to eliminate `EMODPE`, as SGX defines `EMODPE` as the only method for relaxing any enclave page permission. Furthermore, since the decryption key is unreachable after the execution starts, the enclave code cannot be expanded to include more plaintext private code.

The rest of the section explains how Pagoda eliminates `EMODPE` for both single-threaded and multi-threaded applications, with two different approaches.

1) Eliminating `EMODPE` for Single-Threaded Applications: The straightforward approach to eliminate `EMODPE` is to remove all occurences of the `ENCLU` instruction in all enclave code pages. However, this is impractical for three reasons. First, the application/library code might accidentally include the 3-byte sequence {`0x0F, 0x1, 0xD7`} that is interpreted as the `ENCLU` opcode even when the application is not written for SGX, i.e., due to the variable-length nature of the x86 ISA. Second, `ENCLU` serves multiple purposes depending on the leaf index value specified in `rax`. Some uses of `ENCLU`, for example `EEXIT`, are still required by the Pagoda trusted runtime, meaning the `ENCLU` instructions in the trusted runtime must be preserved. Third, the adversary could also abuse existing `ENCLU`s as `EACCEPT` (with `rax`=5) for dynamically allocating new enclave pages, which can contain additional `ENCLU`s.

Pagoda uses a combination of software-only mechanisms to address all issues above for single-threaded applications. At a high level, Pagoda ensures that a) only the Pagoda trusted runtime contains `ENCLU` instructions, b) any use of `ENCLU` as `EMODPE` in the runtime is disallowed, and c) any use of `ENCLU` as `EACCEPT` in the runtime is disallowed, thus no new executable `ENCLU` can be added dynamically. For a), after the code decryption, the trusted loader scans the application or library code pages to ensure that the code segments do not contain the 3-byte sequence that can be interpreted as the `ENCLU` instruction. Correspondingly, Pagoda expects software

```
enclu         ; enclave operation depends on rax
cmp rax, 6    ; checks if ENCLU is used to
              ; extend page permissions
je abort
cmp rax, 5    ; check if ENCLU is used to
              ; add new enclave pages
je abort
```

Fig. 4: Pagoda inserts two simple checks after every `ENCLU` in the trusted runtime. This prevents the adversary from abusing `ENCLU` to subvert the execute-only memory protection or adding adversary-crafted pages.

developers to ensure that their code does not contain the 3-byte sequence[1]. If this 3-byte sequence appears in the application or library binary, the developer can leverage an existing code rewriting technique to eliminate the occurrence of this 3-byte sequence that does not require either source code change or recompilation [55]. For b), the Pagoda trusted runtime mediates each use of the `ENCLU` instruction to detect the leaf index that subverts XOM protection (when `rax`=6). To detect an illegal use, the Pagoda runtime adds a check after every such instance as shown in Figure 4. We add the check after each `ENCLU` instead of before because with control of the program data, the adversary can easily hijack control-flow to skip any check before the `ENCLU` instruction. However, the adversary cannot skip the `ENCLU` instruction itself, which is immediately followed by the check to abort the application execution should it be abused. Similarly, for c), Pagoda additionally detects the usage of `ENCLU` as `EACCEPT` and aborts accordingly.

However, due to exceptions/interrupts, each `ENCLU` and its succeeding checks are not guaranteed to execute atomically. It is possible to interrupt the victim enclave thread right between `ENCLU` and the check. With the enclave thread interrupted, the non-enclave software can return to the enclave in two ways. First, non-enclave software can call `ERESUME` to resume the enclave execution from the check, which is the expected behavior under a benign interrupt (e.g., a normal context-switch). Second, non-enclave software can call `EENTER` to enter the enclave through `OEENTRY` and start new enclave execution without handling the previous AEX. We recognize that this second case allows the adversary to bypass the check after abusing an `ENCLU` as `EMODPE`. To prevent such an attack: Since the `OENTRY` is hooked to the Pagoda trusted runtime, we carefully design the Pagoda trusted runtime to scan the interrupted address stored in the SSA (if any) immediately at the enclave entry point. If this scan identifies that any pending interrupt occurred between `ENCLU` and the checks, the enclave execution is immediately aborted, since a benign execution would resolve the interrupt through `ERESUME` instead. In this way, each `ENCLU` and its following checks are guaranteed to be atomic from the enclave's perspective.

2) Eliminating `EMODPE` for Multi-Threaded Applications: The above approach is insufficient for multi-threading scenar-

[1]The probability of this 3-byte sequence appears in a binary is extremely low. As an example, only 3 files with a total of 7 appearances of this 3-byte sequence are present among all library files under /usr/x86_64-linux-gnu and /usr/lib/x86_64-linux-gnu in our evaluated machine.

ios, due to the concurrency between threads. The adversary can suspend a thread indefinitely between an ENCLU (after it has been used as EMODPE) and the following checks, and use other enclave threads to arbitrarily read/write the enclave code page with its permission relaxed. Although the classic mitigation for such an attack is to enforce mutual exclusion between enclave threads, the use of mutex/conditional variables in an enclave is vulnerable to the arbitrary read/write gadgets controlled by the adversary, and Pagoda cannot rely on the malicious OS kernel for blocking threads.

Therefore, we propose a microcode change to eliminate the use of EMODPE directly without any effort from §V-1. Microcode updates/patches are a means to modify the hardware in existing Intel CPUs [33]. SGX leaf functions are implemented with XuCode [13], which is similar to microcode and can be modified with microcode patches.

Selective EMODPE Enabling The basic idea is to ensure EMODPE, the instruction to extend page permissions, can be selectively disabled after the enclave initialization (EINIT). First, we modify the SGX Enclave Control Structure (SECS), a data structure created by the enclave user for storing per-enclave metadata, and the Enclave Signature Structure (SigStruct), a data structure created and signed by enclave developers to verify the correctness of the enclave state at EINIT. Both SECS and SigStruct will now include a field en_emodpe for enabling EMODPE at runtime. This design allows enclave software developers concerned about software piracy to define SigStruct with EMODPE disabled, whereas other enclave developers may set en_emodpe to allow reconfiguring enclave page permissions at runtime. Second, the workflow of EINIT additionally verifies whether the fields in both structures match. If not, the enclave user is potentially an adversary who wishes to perform EMODPE at execution time. After EINIT, SGX hardware enforces the immutability of SECS by default. Lastly, we modify the XuCode implementation of EMODPE to check en_emodpe field in SECS and add logic to abort if en_emodpe is unset.

VI. Evaluation

A. Pagoda Implementation

We implement a prototype of Pagoda from scratch for Linux. Pagoda consists of a total of 10K lines of code in C/C++ and assembly. Pagoda does not depend on Intel SGX SDK [27] because Pagoda needs to control the memory layout and handle all enclave events on its own. It uses an existing SGXv2 Linux kernel driver [34] to perform enclave operations. Pagoda currently only supports 64-bit x86 binaries.

B. Experimental Setup and Methodology

We run all experiments on an Intel NUC Kit (NUC7PJYH) equipped with a Intel Gemini Lake processor[2] (1.5GHz, quad-core, no hyper-threading) and 8GB RAM. We reserve the maximum allowed size of enclave memory of 128MB in the BIOS. We use Ubuntu 18.04 with Linux version 5.11.0. All benchmarks are compiled with gcc/g++ 8.4.0.

[2]We use this relatively low-end processor due to its support for SGXv2. Aside from this processor, only the most recent Ice Lake processors support SGXv2.

We evaluate the performance and compatibility of Pagoda using a combination of microbenchmarks, standardized benchmarks and real-world applications.

First, we run microbenchmarks to understand the overhead induced by Pagoda's handling of cross-boundary calls. The microbenchmarks invoke different types of system calls and library function calls.

Second, we use SPEC CPU2017 [10] with input size *ref* for evaluating the performance of Pagoda on typical CPU-bound programs. The performance is measured by the execution time averaged over five identical experiments. All benchmarks are statically-linked and position-independent, since Pagoda must determine the base of the binary image at runtime. Consequently, the result does not include several Fortran benchmarks (e.g., bwaves) given Fortran compiler does not support compiling static, position-independent binaries.

We use two types of real-world applications to demonstrate the benefit of Pagoda for both cloud and desktop applications. We evaluate two popular multi-threaded server applications, Lighttpd [5], and memcached [7]. We compile both applications as dynamically-linked libraries, and evaluate their performance with the libraries mapped outside the enclave (since those libraries are in fact public) and inside the enclave (as if the libraries are private). We test both applications with four threads.

We also evaluate two desktop games since video games can directly benefit from code confidentiality for anti-piracy purpose. We choose Quake [11] because of its wide use in research [30], [45], [56], [57]. We include another game WitchBlast [12], to demonstrate Pagoda's compatibility. Both games are dynamically-linked because they have dependencies (e.g., OpenGL) that are only distributed in the form of dynamic libraries. The libraries are deemed public and mapped outside the enclave, since they are publicly available in plaintext code. We evaluate the gaming performance using frames-per-second (FPS). Note that these games only use CPU, unlike modern AAA games that heavily rely on GPU.

Throughout the evaluation, we do not include the overhead of the loading process. The experiment of each workload is sufficiently long to amortize the cost of the loading, therefore the measured performance overhead is mainly contributed by SGX and the Pagoda runtime.

We do not include the performance evaluation of the hardware change we proposed in §V-2. This hardware change should induce negligible runtime overhead, since it only adds one simple lookup in the workflow of EINIT and EMODPE. Pagoda only invokes EINIT once throughout the lifetime of an enclave. A valid Pagoda execution never uses EMODPE.

Compatibility We demonstrate Pagoda's compatibility by showing no source code change (for both the main application and dependencies) is needed to run any of the evaluated programs, including the two games with complicated

Event	Overhead per call
System call (§IV-D1)	54K CPU cycles
Unoptimized cross-boundary function call (§IV-D1)	54K CPU cycles
Optimized cross-boundary function call (§IV-D2)	15K CPU cycles

TABLE I: Pagoda's overhead in handling system calls, unoptimized (exception-based) and optimized (trampoline-based) cross-boundary function calls.

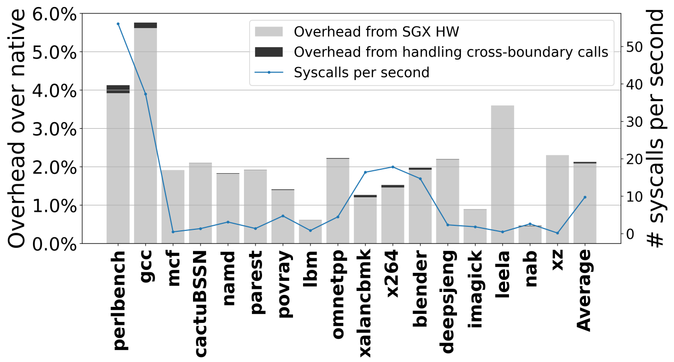

Fig. 5: The breakdown of Pagoda's performance overhead over native Linux execution for SPEC2017 benchmarks.

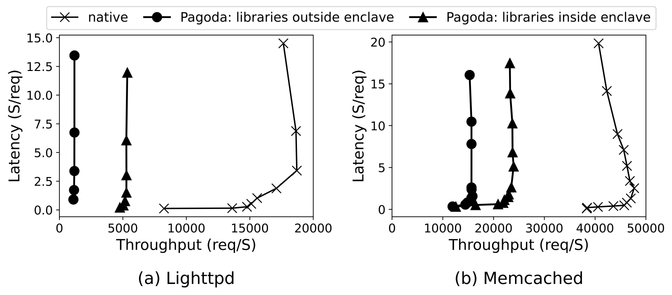

Fig. 6: Throughput vs. Latency of Lighttpd and Memcached. We run both applications with three configurations: bare-metal Linux, Pagoda with all shared libraries outside the enclave (treating all libraries as public), Pagoda with all libraries inside the enclave (treating all libraries as private).

dependencies and system service requirements. Noticeably, Pagoda is the first work which supports GUI applications without requiring source code modifications, as later discussed in §VIII.

C. Measuring Cross-Boundary Call Overheads

We first measure Pagoda's performance overhead in handling system calls and public function calls, and show the result in Table I. These numbers reflect the execution time cost of the communication routines described in §IV. Notice the overhead numbers are independent of the actual system/-function calls and the parameters, since the cross-boundary calls handling procedure does not take the system/function call parameters into account. System calls and unoptimized function calls have larger performance overhead compared to optimized cross-boundary function calls due to multiple enclave/ring boundary crossings caused by exceptions.

D. SPEC 2017

We use SPEC 2017 [10] to drive the evaluation for single-threaded, CPU-bound applications. We show the runtime overhead of Pagoda over the native Linux execution for each evaluated SPEC17 benchmark in Figure 5. On average, Pagoda adds 2.13% runtime overhead over the native execution.

The overhead comes from two major sources – the system call overhead introduced by Pagoda, since statically-linked binaries only communicate with non-enclave software through system calls, as well as the inherent slowdown caused by the SGX hardware (e.g., memory encryption engine, SGX-specific memory access control). To compute the overhead caused by Pagoda's system call handling, we count the number of requested system calls per second by each benchmark, and show the numbers in Figure 5. This number is multiplied by the per-system call overhead reported in Table I to measure how much Pagoda's system call handling contributes to the overall overhead. On average, Pagoda's system call handling only adds 0.04% performance overhead, while the remaining 2.09% is attributed to the SGX hardware. For example, in perlbench and gcc, the overhead contributed by system call handling is higher than other benchmarks due to more frequent system call invocations.

E. Server Applications

We evaluate the performance of Pagoda on two common network-heavy server applications, Lighttpd and Memcached. We use ApacheBench [1] and Memtier [8] to increase the concurrent requests until the throughput reaches the limit. We also evaluate the performance implications of mapping shared libraries inside the enclave when the libraries requiring dynamic linking/loading are private by running two Pagoda configurations, one with all libraries mapped inside the enclave, and the other leaving all libraries outside the enclave. For Lighttpd, the peak throughput of running Lighttpd on bare-metal Linux is 18.7K. The number becomes 1.2K when using Pagoda with all shared libraries mapped outside the enclave, and 5.4K when all libraries are inside the enclave[3]. We see similar trend with Memcached: when running on bare-metal Linux, the peak throughput is 47K. The peak throughput drops to 24K when Pagoda maps libraries inside the enclave, and further down to 16K when libraries are outside. The large performance difference between Pagoda and native execution is due to the frequent cross-boundary calls, as both applications are IO-intensive. Mapping shared libraries inside the enclave shows a more significant performance benefit than mapping libraries outside, due to reduced cross-boundary calls. As an example, when running Lighttpd with libraries outside the enclave, each request causes about 73.5 public library function calls. When all libraries are mapped inside the enclave, the program invokes around 3.42 system calls per request. To summarize, when the frequency of the main application calling library functions is higher than the frequency of the called

[3]We notice that Pagoda reports a much larger throughput degradation than other SGX systems such as GrapheneSGX [53]. We attribute the cause of the discrepancy to that we use a mobile-class processor for our performance evaluation. When we test GrapheneSGX on our platform, we observe worse throughput compared to Pagoda.

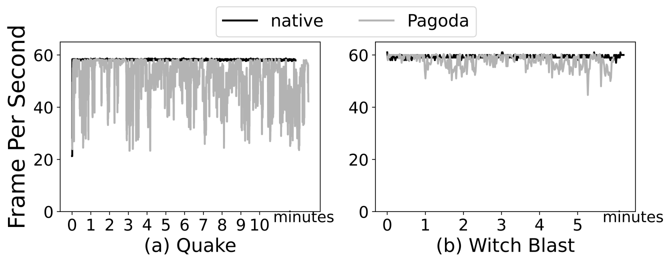

Fig. 7: Comparing the change of Frame-Per-Second over time between native Linux execution and Pagoda.

functions making system calls, mapping shared libraries inside the enclave can show a performance advantage due to fewer cross-boundary calls, aside from the security benefit of protecting the privacy of library code. Future work can leverage the asynchronous system call approach proposed by [17], [43] to largely reduce this overhead, by processing IO requests in batch.

F. *Desktop (Gaming) Applications*

We run desktop games to evaluate both the performance and the compatibility of Pagoda on client graphical applications. Video games demand higher compatibility because they often depend on more complicated OS abstractions (e.g., user-mode DMA). In our evaluation, Pagoda is able to run both games without any source code changes. Both games are dynamically linked with over 100 libraries mapped outside of the enclave. For each game, we manually play it for a few minutes and repeat it for the native execution and Pagoda.

We show the real-time FPS in Figure 7. For Quake, native execution achieves an average FPS rate of 58.1, with standard deviation of 1.20. When using Pagoda, the average FPS drops to 50.1, and the standard deviation increases to 8.37. The result of WitchBlast is similar, despite being slightly better than Quake: Pagoda achieves an average FPS of 57.5, compared to 59.3 with native execution, and the standard deviation is 2.56, compared to 0.57 when running on bare-metal Linux.

Pagoda causes lower and more unstable FPS rate due to the cross-boundary calls. For Quake, Pagoda makes 1547.8 cross-boundary function calls per frame, in which 255.7 are unoptimized since those functions are from dynamically-loaded libraries and cannot be optimized by the Pagoda loader. For WitchBlast, we observe 1504 cross-boundary function calls per frame, similar to the number of Quake. However, only 0.7 out of all 1504.0 calls are unoptimized, which explains why the FPS rate of WidthBlast is higher and more stable compared to Quake.

VII. Discussion

A. *Indirect Attack*

Pagoda enforces code privacy by preventing direct, unauthorized reads and writes to the private enclave code pages at anytime. However, we recognize that there exist other ways that an attacker can use to indirectly infer private enclave instructions, by monitoring other information in its view of the enclave's execution. We classify such indirect leakage into two categories, depending on whether the attacker monitors program data or micro-architectural states.

First, not only can attacker control non-enclave data, with the gadget exploration and the code reuse attack discussed in §IV-B, we envision that attacker could also observe and modify enclave data via its control over the arbitrary read/write gadgets, and additionally corrupt the enclave's control flow. Ideally, this enables attackers to reverse-engineer an arbitrary sequence of enclave instructions, by creating an enclave state, jumping to the target instructions, and observing the output enclave state after the execution of the target instructions. Pagoda is susceptible to this type of attack as this attack does not require breaking the XOM protection. To mitigate this attack, future work can leverage control-flow integrity techniques to prevent the attacker from probing arbitrary instructions with attacker-crafted enclave state. For example, Intel CET [26] enhanced with fine-grained control-flow guarantee [6] can be directly combined with SGX to provide strong control-flow integrity even when program data is corrupted by attacker.

Likewise, microarchitectural side channels provide additional information about the executed enclave instructions. Whether the side channels can be practically exploited is an interesting question that we leave for future work. Importantly, all prior side-channel attacks assume a public program (or algorithm), and rely on this assumption in a fundamental way for the attack to succeed. For example, cache side-channels rely on the attacker's knowledge of the cache access pattern as a function of the secret data. Thus, side-channel attacks on a private program cannot work "out of the box". Whether it is possible to reverse-engineer the binary starting from a private program is a non-trivial and interesting question that requires new techniques. For example, large classes of instructions, such as simple arithmetic and logic operations, result in indistinguishable micro-architectural side effects.

B. *Data Protection*

Data protection is outside the scope of Pagoda, as code privacy alone is beneficial in many scenarios. However, future work can combine the security mechanism of Pagoda with the data protection approach adopted by prior SGX systems, such as GrapheneSGX. While this may sacrifice the performance and compatibility advantage of Pagoda, future work may explore partitioning unmodified application to only include secret-dependent data into enclaves, similar to [39], [40]. We envision that such techniques may be employed to determine how to configure applications prior to deployment on Pagoda to achieve intended security.

VIII. Related Work

SGX Frameworks for Unmodified Applications Several existing SGX systems support unmodified applications by performing system calls with an in-enclave Library OS (LibOS) [20], [46], [53]. A user only needs to specify the main binary with all dependencies, and those systems map the executable and all libraries, together with the LibOS inside the enclave. The primary benefit of the LibOS-based system is that most library functions and system calls are handled by the LibOS alone inside the enclave, without incurring expensive enclave-host communications. However, existing LibOS-based systems only support applications with limited set of dependencies and system call requirements (e.g., command-line applications dependent on libc). Terminal applications such as vim and graphical applications such as games, which require more dependencies and system service supports, are not currently supported by any LibOS-based systems. Panoply [51] is similar to Pagoda as it provisions a runtime interface to delegate library calls to non-enclave software. However, Panoply requires significant software rewriting to both the application and the dependent libraries, which is impractical for large legacy applications. Other systems like SCONE [17] and Ryoan [32] are designed for specific types of applications, so therefore do not support arbitrary Linux applications. More importantly, all existing SGX systems focus primarily on data protection without making code privacy their goal.

Code Protection with SGX The original Intel SGX SDK [27] and other SGX frameworks were designed for data protection. Later, people realized that code privacy protection is missing in the landscape and proposed different solutions for achieving code privacy. SGXElide [19] requires the programmer to identify secret code, and encrypts the secret code before the program is shipped to the untrusted users. SGXElide designs a complete attestation and decryption process, including remote-attestation to verify the authenticity of the enclave software, and protecting the decryption key with SGX's sealing. Pagoda adopts and extends this process into its loading procedure. A drawback of SGXElide is that it requires rewriting application with Intel SGX SDK. Similarly, Intel SGX PCL [35], a confidential code loader developed by Intel to augment the original Intel SGX SDK, also requires code refactoring. TEEShift [36] and SGXCrypter [54] propose similar solutions but all suffer from the burden of code refactoring. In conclusion, none of the prior code privacy solutions consider confidentiality for code in-use (instead they require code refactoring to produce vulnerability-free enclave programs), and preserve compatibility with unmodified applications.

Memory Vulnerability Attacks on SGX As memory vulnerabilities are common in large applications, there is a rich line of work exploiting vulnerabilities in enclave software for leaking enclave code/data. DarkROP [38] leverages exception handling to blindly search the enclave software for vulnerabilities and different types of gadgets, including arbitrary read/write gadgets (`memcpy`), ROP-gadgets, etc. By constructing adversary-controlled `memcpy`, arbitrary readable enclave content can be leaked. Biondo et. al. recognize that enclave software usually contains an enclave-specific runtime (especially for applications written with Intel SGX SDK), and this runtime contains enough gadgets to mount ROP attacks [21]. TeeRex [24] further automates the vulnerability and gadgets exploration for enclave programs. Attackers can leverage these attacks to copy arbitrary readable enclave content into non-enclave memory.

IX. Conclusion

This paper proposes Pagoda, a practical enclave-based framework for code privacy protection of unmodified applications with a low performance overhead. Pagoda does not follow the conventional SGX design paradigm that protects both code and data with enclave memory. Instead, Pagoda presents a new way of using enclaves that achieves high performance and compatibility, and simultaneously enforces code privacy protection by applying SGX-enforced XOM on private code binaries.

Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments and suggestions. We are grateful for the help and feedback from Andrew Baumann, Tony Chen, Daniel King, Ben Niu, Marcus Peinado and Bo Zhang from Microsoft. This work was funded in part by a Microsoft Research Ph.D. Fellowship, and by NSF grants 1816282, 1909999, 1942888, 1954521 and 1956374.

References

[1] ab - apache http server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[2] Assassin's creed origins redlining some cpus at 100 percent. https://www.extremetech.com/gaming/258173-assassins-creed-origins-redlining-cpus-100.

[3] Can video game piracy be stopped in two years? https://www.cnbc.com/2016/01/14/can-video-game-piracy-be-stopped-in-two-years.html.

[4] Confidential computing. https://www.ibm.com/cloud/learn/confidential-computing.

[5] Lighttpd. https://www.lighttpd.net/.

[6] The llvm compiler infrastructure. https://github.com/intel/fineibt_llvm/.

[7] Memcached. https://memcached.org/.

[8] memtier_benchmark: A high-throughput benchmarking tool for redis & memcached. https://github.com/RedisLabs/memtier_benchmark.

[9] Microsoft tempts software pirates with 50 percent discount on office. https://www.theverge.com/2021/12/9/22825774/microsoft-office-pirated-software-discount-offer.

[10] SPEC CPU2017. https://www.spec.org/cpu2017.

[11] vkquake. https://github.com/Novum/vkQuake.

[12] Witch blast. https://github.com/Cirrus-Minor/witchblast.

[13] Xucode: An innovative technology for implementing complex instruction flows. https://software.intel.com/content/www/us/en/develop/articles/software-security-guidance/secure-coding/xucode-implementing-complex-instruction-flows.html.

[14] Cloud Security Alliance. Top threats to cloud computing version 1.0. 2010.

[15] T Alves and D Felton. TrustZone: Integrated hardware and software security. *ARM white paper*, 2004.

[16] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, volume 13, page 7. ACM New York, NY, USA, 2013.

[17] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L Stillwell, et al. Scone: secure linux containers with intel sgx. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, pages 689–703, 2016.

[18] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1342–1353, 2014.

[19] Erick Bauman, Huibo Wang, Mingwei Zhang, and Zhiqiang Lin. Sgxelide: enabling enclave code secrecy via self-modification. In *Proceedings of the 16th International Symposium on Code Generation and Optimization*, pages 75–86, 2018.

[20] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, pages 267–283, 2014.

[21] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The guard's dilemma: efficient code-reuse attacks against intel sgx. In *Proceedings of the 27th USENIX Conference on Security Symposium*, pages 1213–1227, 2018.

[22] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, pages 227–242. IEEE, 2014.

[23] Kjell Braden. Leakage-resilient layout randomization for mobile devices. In *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[24] Tobias Cloosters, Michael Rodler, and Lucas Davi. Teerex: discovery and exploitation of memory corruption vulnerabilities in sgx enclaves. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 841–858, 2020.

[25] Christian S. Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE Transactions on software engineering*, 28(8):735–746, 2002.

[26] Intel Coporation. Control-flow enforcement technology specification. 2019.

[27] Intel Corporation. Intel® software guard extensions sdk for linux* os (developer reference). 2016.

[28] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical code randomization resilient to memory disclosure. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, pages 763–780, 2015.

[29] Wes Fenlon. Denuvo drm cracks seem to be happening faster and faster. https://www.pcgamer.com/denuvo-cracks-2019/.

[30] Vladimir Gajinov, Ferad Zyulkyarov, Osman S Unsal, Adrian Cristal, Eduard Ayguade, Tim Harris, and Mateo Valero. Quaketm: parallelizing a complex sequential application using transactional memory. In *Proceedings of the 23rd International Conference on Supercomputing*, pages 126–135, 2009.

[31] Jason Gionta, William Enck, and Peng Ning. Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 325–336, 2015.

[32] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. *ACM Transactions on Computer Systems (TOCS)*, 35(4):1–32, 2018.

[33] Intel. Affected processors: Transient execution attacks and related security issues by cpu. https://software.intel.com/content/www/us/en/develop/topics/software-security-guidance/processors-affected-consolidated-product-cpu-model.html.

[34] Intel. linux-sgx-driver. https://github.com/0xabu/linux-sgx-driver.

[35] Intel. Intel® software guard extensions (intel® sgx) protected code loader (pcl) for linux. 2018.

[36] Titouan Lazard, Johannes Götzfried, Tilo Müller, Gianni Santinelli, and Vincent Lefebvre. Teeshift: Protecting code confidentiality by selectively shifting functions into tees. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, pages 14–19, 2018.

[37] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the 15th European Conference on Computer Systems*, pages 1–16, 2020.

[38] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. Hacking in darkness: return-oriented programming against secure enclaves. In *Proceedings of the 26th USENIX Conference on Security Symposium*, pages 523–539, 2017.

[39] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. Glamdring: automatic application partitioning for intel sgx. In *Proceedings of the 2017 USENIX Conference on USENIX Annual Technical Conference*, pages 285–298, 2017.

[40] Jed Liu, Owen Arden, Michael D George, and Andrew C Myers. Fabric: Building open distributed systems securely by construction. *Journal of Computer Security*, 25(4-5):367–426, 2017.

[41] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–1, 2013.

[42] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitraş. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, pages 692–708, 2015.

[43] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless os services for sgx enclaves. In *Proceedings of the 12th European Conference on Computer Systems*, pages 238–253, 2017.

[44] Alessio Palumbo. Assassin's creed origins denuvo + vmprotect combo hacked three months after launch. https://wccftech.com/assassins-creed-origins-denuvo-hacked/.

[45] Seonghyun Park, Adil Ahmad, and Byoungyoung Lee. Blackmirror: Preventing wallhacks in 3d online fps games. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 987–1000, 2020.

[46] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. Sgx-lkl: Securing the host os interface for trusted execution. *arXiv preprint arXiv:1908.11143*, 2019.

[47] Kathleen Reavis Conner and Richard P Rumelt. Software piracy: An analysis of protection strategies. *Management science*, 37(2):125–139, 1991.

[48] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. Sgx-shield: Enabling address space layout randomization for sgx programs. In *Proceedings of the Network and Distributed System Security Symposium*, 2017.

[49] AMD SEV-SNP. Strengthening vm isolation with integrity protection and more. *White Paper, January*, 2020.

[50] Kripa Shanker, Arun Joseph, and Vinod Ganapathy. An evaluation of methods to port legacy code to sgx enclaves. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1077–1088, 2020.

[51] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-tcb linux applications with sgx enclaves. In *Proceedings of the Network and Distributed System Security Symposium*, 2017.

[52] Hassan Takabi, James BD Joshi, and Gail-Joon Ahn. Security and privacy challenges in cloud computing environments. *IEEE Security & Privacy*, 8(6):24–31, 2010.

[53] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-sgx: a practical library os for unmodified applications on sgx. In *Proceedings of the 2017 USENIX Conference on USENIX Annual Technical Conference*, pages 645–658, 2017.

[54] Dimitrios Tychalas, Nektarios Georgios Tsoutsos, and Michail Maniatakos. Sgxcrypter: Ip protection for portable executables using intel's sgx technology. In *Proceedings of the 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 354–359, 2017.

[55] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. Erim: secure, efficient in-process isolation with protection keys (mpk). In *Proceedings of the 28th USENIX Conference on Security Symposium*, pages 1221–1238, 2019.

[56] A Yahyavi, K Huguenin, J Gascon-Samson, J Kienzle, and B Kemme. Watchmen: Scalable cheat-resistant support for distributed multi-player online games. In *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS)*, pages 134–144. IEEE, 2013.

[57] Ferad Zyulkyarov, Vladimir Gajinov, Osman S Unsal, Adrián Cristal, Eduard Ayguadé, Tim Harris, and Mateo Valero. Atomic quake: using transactional memory in an interactive multiplayer game server. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 25–34, 2009.