An Evaluation of Methods to Port Legacy Code to SGX Enclaves

Kripa Shanker
Indian Institute of Science
Bangalore, India
kripashanker@iisc.ac.in

Arun Joseph
Indian Institute of Science
Bangalore, India
arunj@iisc.ac.in

Vinod Ganapathy
Indian Institute of Science
Bangalore, India
vg@iisc.ac.in

ABSTRACT

The Intel Security Guard Extensions (SGX) architecture enables the abstraction of enclaved execution, using which an application can protect its code and data from powerful adversaries, including system software that executes with the highest processor privilege. While the Intel SGX architecture exports an ISA with low-level instructions that enable applications to create enclaves, the task of writing applications using this ISA has been left to the software community.

We consider the problem of porting legacy applications to SGX enclaves. In the approximately four years to date since the Intel SGX became commercially available, the community has developed three different models to port applications to enclaves—the library OS, the library wrapper, and the instruction wrapper models.

In this paper, we conduct an empirical evaluation of the merits and costs of each model. We report on our attempt to port a handful of real-world application benchmarks (including OpenSSL, Memcached, a Web server and a Python interpreter) to SGX enclaves using prototypes that embody each of the above models. Our evaluation focuses on the merits and costs of each of these models from the perspective of the effort required to port code under each of these models, the effort to re-engineer an application to work with enclaves, the security offered by each model, and the runtime performance of the applications under these models.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

SGX, enclaves, porting, legacy code

ACM Reference Format:
Kripa Shanker, Arun Joseph, and Vinod Ganapathy. 2020. An Evaluation of Methods to Port Legacy Code to SGX Enclaves. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3409726

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3409726

1 INTRODUCTION

Intel's Software Guard Extensions (SGX) [10, 14] technology has now been commercially available since microprocessors using the Skylake micro-architecture were launched in August 2016. Using the facilities of the SGX, user-level applications can create *enclaves* within which they can place their sensitive code and data. Enclaves are cryptographically secured by the hardware so that an adversary cannot observe the data or the computations with in the enclave. SGX's threat model accommodates a powerful set of adversaries, including the most privileged software running on the system, *i.e.,* the operating system or the hypervisor.

This powerful threat model makes SGX attractive for use in public cloud computing platforms. On such platforms, the cloud provider controls the system software. An adversarial cloud provider (or a benign one acting under government subpoena) can leverage this control to completely subvert the confidentiality and integrity of a cloud client. The cloud provider can peek into and arbitrarily modify the state of a client's virtual machines or containers. This makes public cloud computing environments unattractive to clients in several domains that handle sensitive data, such as healthcare and banking. To accommodate clients with such sensitive computing needs, a number of public cloud providers have begun to deploy SGX hardware in their data centres, and offer solutions that allow clients to leverage the capabilities of the SGX to build applications.

A client that wishes to leverage SGX must write its applications to be SGX-aware. An SGX-aware application will place its sensitive data in enclaves, and ensure that the code that operates on this data is also placed in the enclave. The SGX hardware places certain restrictions on the kinds of instructions that can execute within enclaves, *e.g.,* system calls cannot be executed within an enclave. Enclave code must be written to respect these restrictions, *e.g.,* by having the application that created the enclave make the system call on behalf of the enclave. The enclave code must also take care to ensure that it does not inadvertently leak sensitive data outside the enclave, and that any sensitive data written outside the enclave is cryptographically-protected using keys stored within the enclave. Thus, while the SGX hardware offers powerful primitives, much of the responsibility of ensuring the confidentiality and integrity of enclave data falls on the application authors.

A number of techniques have been proposed in the literature to allow application authors build secure enclave applications. These techniques range from those that statically verify the absence of information leaks from enclave applications [27], programming-aids and libraries to allow enclave applications to be written easily with encryption of any egress data being handled by the library [26], and techniques that use programmer annotations on sensitive data structures to automatically split applications into enclave/non-enclave portions [13]. The focus of these techniques is to aid authors of enclave applications, writing *new* code tailored to use SGX.

The focus of this paper is on frameworks that have been developed to allow *legacy* code (*i.e.,* code that has already been written) to execute within enclaves. While several applications have been tailor-built for enclaves (*e.g.,* [20, 24]), this is a resource-intensive process, and application developers may wish to enjoy the benefits of the SGX without the upfront investment needed to build enclave code from scratch. These frameworks provide the necessary in-enclave support to allow legacy code to operate within the constraints imposed by the enclave programming model, *e.g.,* inability to perform certain operations such as system calls within the enclave. These frameworks broadly follow three different models:

① ***Library OS model.*** In this model, an entire library OS executes within the enclave. To port an application within the enclave, the application developer loads the application binary together with any libraries that it uses, and executes the resulting binary within the enclave. The library OS itself implements much of the functionality of a traditional OS, so the enclave interface to the rest of the system is simple and low-level. Examples frameworks implementing this model are Haven [5], Graphene-SGX [31], and SGX-LKL [21].

② ***Library wrapper model.*** This model, which is implemented by Panoply [25], assumes that applications invoke system services via libraries such as the standard C library (`libc`). Normally, these libraries contain the low-level system calls and other sensitive instructions that cannot be executed within the enclave. Panoply provides library wrappers that enclave-based applications can link against. Panoply's library wrappers then ensure that the library code is invoked outside the enclave. The enclave interface is that of the standard C library.

③ ***Instruction wrapper model.*** In this model, wrappers are provided for low-level instructions (such as `syscall`, `inb`, `outb`) that are not permitted within enclaves. The wrappers contain the machinery to cross the enclave boundary, and take care of data protection. They automatically encrypt all data leaving the enclave boundary and decrypt and ciphertext data received by the enclave. In this case, the enclave interface is the set of instructions that are forbidden by Intel SGX (*e.g.,* `syscall`, `inb`, `outb`), which the instruction wrapper implementation intercepts, and executes outside the enclave. Because applications rarely use the low-level instructions in their raw form, and rely on libraries to perform system calls, the instruction wrapper can itself be implemented by modifying a standard C library implementation. SCONE [4] and lxcsgx [28] use this model.

We present these models in more detail in Section 3. While the example frameworks discussed above have been developed with legacy applications in mind, they implement the heavy-lifting needed to get applications to conform to the constraints imposed by enclave programming. The same frameworks can also be leveraged as supporting infrastructure by new enclave-based applications.

The primary contribution of this paper is to evaluate the relative merits of the three methods above in porting legacy code to SGX enclaves from a software engineering perspective. The criteria on which we wish to evaluate the methods are:

- How much effort is required to port application code to enclaves in each of these methods? This criterion measures the amount of effort that it takes a software developer to deploy a first-cut of an application within the enclave.
- How much flexibility does each method offer the application developer in engineering the enclave? For example, suppose that the application developer decides to execute some code outside the enclave for performance reasons, how much effort does the developer need to invest to port the code in that way using each of the methods?
- How much trusted code (in addition to the application's own code) must execute within the enclave?
- What are the performance overheads imposed by each approach?

We discuss these questions in detail in Section 4. To study these questions, we ported a number of popular applications (including OpenSSL, Memcached, a Python interpreter, and a Web server) to SGX enclaves using representative frameworks that implement each of the models described above: Graphene-SGX [31] representing the library OS model, Panoply [25] representing the library wrapper model, and Porpoise, our in-house reimplementation of SCONE [4], representing the instruction wrapper model. We studied the benefits/costs of these methods to port legacy applications.

2 BACKGROUND ON SGX

SGX enables confidentiality and integrity-protected execution of trusted code in untrusted software environments [10, 14]. The primary end-user-visible artifact in an SGX system is the concept of an *enclave*. An enclave is a linear region of a process's virtual address space, the contents of which are protected by SGX from even the most privileged software running on that hardware platform.

A process creates and initializes an enclave via a set of instructions exported by the SGX ISA. To create an enclave, the process provides a pointer to a binary executable and instructs the hardware to initialize the enclave with this binary (which contains the code and data which the enclave must start executing). The hardware reserves a region of the virtual address space for the enclave, and loads up the enclave with this binary. The pages for this portion of the address space are drawn from a reserved region of physical memory (called the encrypted page cache). The hardware then obtains a measurement of the enclave (for attesting it to the entity that started the enclave) and seals the enclave so that any further modifications are not possible [2].

SGX introduces a new *enclave-mode* in which the hardware can execute when executing the code of the enclave. The SGX hardware ensures that the contents of the enclave are visible in the clear only when the processor is in enclave mode, and the control is within that enclave. When the processor is in kernel-mode or in user-mode, any code that attempts to access the enclave will be unable to access the cleartext contents of the enclave. It accomplishes this protection by encrypting the contents of the enclave with hardware-generated keys, and ensuring that decryption only happens when the there is a memory access from code executing within the enclave.

The process enters enclave mode by executing an `EENTER` instruction exported by the SGX ISA. Once the processor is in enclave-mode, the enclave code can freely access data stored both within the enclave, as well as other user-space memory within the process's address space.

Table 1: Set of instructions forbidden for use within the enclave. Adapted from the Intel SGX manual [11].

Instruction	Comment
cpuid, getsec, rdpmc, sgdt, sidt, sldt, str, vmcall, vmfunc	Might cause VM exit
in, ins/insb/insw/insd, out, outs/outsb/outsw/outsd	I/O fault may not safely recover. May require emulation.
Far call, Far jump, Far Ret, int n/into, iret, lds/les/lfs/lgs/lss, mov to DS/ES/SS/FS/GS, pop of DS/ES/SS/FS/GS, syscall, sysenter	Accessing the segment register could change privilege level.
lar, verr, verw	Might provide access to kernel information.
enclu[eenter], enclu[eresume]	Cannot enter an enclave from within an enclave.

However, SGX places several restrictions on the instructions that can execute when the processor is in enclave mode. Table 1 lists the set of instructions that are forbidden in enclave mode. These instructions can either be executed when the process is in user-mode (*e.g.,* `syscall`, `enclu`), or by the privileged system software (*e.g.,* the OS or the hypervisor) on behalf of the process (*e.g.,* `encls`, modifications to the registers `DS/ES/SS/FS/GS`). Applications executing within the enclave must not include these instructions.

Legacy applications are not written with enclaves in mind, and may include many of these instructions, *e.g.,* instructions such as `syscall`, `sysenter` and `int` are routinely used within user-space applications to invoke kernel services. While these instructions are permitted for execution in the processor's user-mode, they are not permitted when the processor is in enclave-mode. The main goal of the enclave-execution frameworks is therefore to enable user-space applications that use these instructions to execute within the enclave by suitably wrapping the forbidden instructions and forwarding them for execution outside the enclave.

Once the enclave has completed execution, it exits using the `EEXIT` instruction, transferring control back to the user process that entered the enclave. Enclave exits can also happen asynchronously (called an `AEX` in SGX). In both cases, the hardware saves the state of the enclave, scrubs registers, and returns to user-mode.

Threat Model. As is standard with SGX, we assume that the enclave contains sensitive code and data that must be protected from adversaries. SGX admits a powerful adversary model in which even the code of the user-space process that launches the enclave and the privileged system software (*e.g.,* OS or hypervisor) are untrusted. SGX protects against these adversaries by encrypting the enclave contents with hardware-managed keys, and decrypting the contents only when the access is from within the enclave. Decryption is done within the cache-hierarchy to prevent cold-boot and bus-snooping attacks on the contents of the enclave.

The attacker can attempt to attack the enclave in a variety of ways to compromise confidentiality and integrity. SGX provides confidentiality and integrity protection against such adversaries using standard cryptographic techniques (albeit implemented in hardware). The adversary can also attempt to subvert the execution of the enclave by feeding it malformed input, *e.g.,* to exploit a memory error in the enclave code itself, or by suitably modifying return values when enclave code interacts with non-enclave code (IAGO-like attacks [6]). Such attacks are a realistic threat, especially the enclave contains a lot of low-level trusted code. Systems such as Haven [5] attempt to protect against some such attacks (in particular, IAGO attacks) by implementing a shim layer that checks the values that cross the enclave boundary. Nevertheless, it is important to minimize the amount of trusted code running within the enclave. Indeed, this is one of the metrics on which we evaluate the various enclave-execution frameworks that we consider.

For the purposes of this paper, however, we do not consider within our threat model recent work on hardware-based side channels to subvert SGX (*e.g.,* ForeShadow [34]). While these attacks constitute a serious threat to SGX, we consider them out of scope for this work, whose main goal is to evaluate the merits and costs of various approaches to enclave code development.

3 ENCLAVE-EXECUTION MODELS

In this section, we present the technical details of the three models that have been proposed to date in the research community to support enclave-based applications. All the three models referenced in this section are illustrated in Figure 1. Fundamentally, the three models differ in how much processing they perform within the enclave before crossing the enclave boundary.

3.1 Library OS Model

In this model, used in Haven [5], Graphene-SGX [31] and LKL-SGX [21], the enclave consists of the application to be protected linked with a library OS. Library OSes (*e.g.,* Drawbridge [19], the Linux kernel library (LKL) [22], Graphene [33]) offer a user-space implementation of much of the functionality that traditional OSes implement in kernel-model. Privileged operations must still be executed in the processor's supervisor mode (*e.g.,* operations related to protection and isolation, such as switching page tables upon a context switch). Thus, the library OS interfaces with a small privileged software layer that implements these privileged operations.

Because library OSes often implement a lot of the functionality of a traditional OS (*e.g.,* they may have a file system and a network stack implementation), the interface between the library OS and the privileged software layer is typically narrower than the system call layer exposed by the OS to applications, *e.g.,* 38 distinct operations in the interface of Graphene-SGX, 24 in Haven, and 7 in LKL-SGX [21]. The library OS redirects control to the privileged software layer when it performs operations that require instructions from Table 1.

An application developer specifies the binary executable that must be executed within the enclave. The application developer specifies the list of all libraries that the application may potentially use (together with their code). The enclave is initialized with the code of the library OS, the code of the application, as well as all libraries that the application may use. Because the library OS contains all the supporting code needed by the application, it can link dynamically against any libraries it uses (that are already pre-loaded into the enclave).

From an end-user's standpoint, the execution of the application proceeds in a manner very similar to executing the application on a traditional desktop (*i.e.,* without enclaves). It is important to note that no new code is loaded into the enclave after initialization (SGX's attestation model disallows this) even though the end-user gets the illusion of dynamic linking. This is because all the code is loaded into the enclave prior to attestation, and the dynamic linker

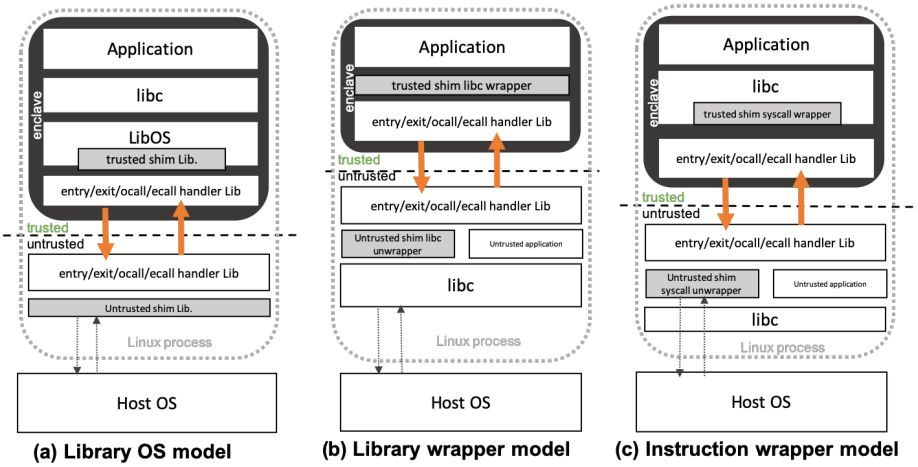

Figure 1: Various models proposed in the literature to support the execution of applications within SGX enclaves.

simply patches up symbol tables as the application references the libraries that it needs at runtime.

The primary benefit of the library OS model is that of a simpler enclave interface. Because the library OS implements functionality that is traditionally relegated to the OS, the enclave interface is greatly simplified. For example, the library OS may implement significant pieces of the file system within the enclave itself, and by the time control reaches the enclave interface, the operations can be specified using reads or writes at the disk block level. Many system calls such as `getpwd`, `fcntl`, `dup` and `brk` that require a user/kernel domain crossing on a traditional system don't even hit the enclave interface. The operations represented by these system calls can simply be implemented as modifications to data structures to the file system implementation within the library OS. These calls do not modify security-sensitive state of any other applications and do not need to be executed by privileged system software.

For some system calls, it is important *not* to cross over from the enclave. For example, applications typically rely on the OS to provide a source of randomness. Since the OS is untrusted, the application can no longer simply trust the results of a `getrandom` system call executed by the OS; the OS could simply cheat by providing a poor set of random values, weakening any cryptographic keys that the application may then generate using these random values. Instead, the underlying enclave execution framework must leverage other mechanisms to obtain randomness, *e.g.,* the `rdrand` x86-64 instruction, which sources randomness from the hardware.

On the SGX, all code running outside the enclave is untrusted. This includes the user-space application within which the enclave is initiated, as well as the privileged system software layer. As a result, enclaves must typically guard against IAGO-like attacks [6], in which the untrusted code attempts to compromise the security and privacy of enclave code by passing malicious return values to calls that cross the enclave boundary.

In the library OS model, the interface to the enclave is conceptually simpler than even the system call interface on traditional OSes. For example, as discussed previously, among the three library OS execution models discussed in the literature, Graphene-SGX has the widest interface, and even that interface consists of only 38 interfaces, as opposed to a few hundred in a typical system call interface. As a result, on systems that use the library OS model, it is easier to design shim layers to protect against IAGO-like attacks.

The main disadvantage of the library OS model is that the entire library OS runs within the enclave, amounting to code that is a few hundreds of thousands of lines within the trusted-computing base (TCB); we provide concrete numbers in Section 4. As a result, an attacker who targets the enclave-based application has a larger attack surface to work with, and any bugs within the library OS or other supporting code become a liability for the enclave.

As proposed and illustrated in the research literature on library OS prototypes, entire applications execute within the enclave. This offers little flexibility to a potential application developer who wishes to execute only part of the application within the enclave. For example, consider an in-enclave Python interpreter—the developer may wish to execute only the core interpreter loop within the enclave, leaving all the other functionality of the interpreter outside the enclave. This may either be for performance reasons, or to reduce the amount of trusted code in the application.

In the library OS model, re-engineering application code is cumbersome at best. The portion of the application code that executes will need to communicate with the rest of the application, but this will involve communicating across the layers of the library OS. This model also complicates the case where two enclave-based applications need to interact with each other. For example, suppose that a Python interpreter, executing within an enclave, needs to communicate with a key-value store, also executing within an enclave. The two are mutually-distrusting, and therefore cannot execute as applications within the same library OS (and hence the same enclave). Thus, they execute in two different enclaves, and each message in a cross-enclave communication must go through the library OS layers in both enclaves. Graphene-SGX implements this idea in the notion of *enclave groups*, which are a group of mutually-distrusting, yet interacting applications, derived from a common parent process using a `fork`.

3.2 Library Wrapper Model

To our knowledge, Panoply [25] is the only system that implements the library wrapper model. In stark contrast to the library OS model, the library wrapper model implements an interface that exits the enclave while implementing little additional functionality within the enclave. In Panoply, the standard C library executes outside the enclave. It provides library wrappers that enclave applications can link against. The wrappers simply marshal data and pass it to the library that runs outside the enclave.

Because the in-enclave library wrappers do little more than marshaling/demarshaling data, Panoply adds the least amount of additional code to the enclave (thereby reducing the TCB size within the enclave). The application developer has the flexibility to decide which portion of the application executes within the enclave. The task of splitting an application consists of executing all the enclave code as a separate module, and making a cross-module call to a function within that code. The enclave code itself is easy to produce, by simply linking the module against the Panoply library wrappers.

The flip-side of these benefits is that the library code that executes outside the enclave is untrusted, and can be used by the attacker to subvert the enclave application, *e.g.,* via IAGO-like attacks. Because the library wrapper is much larger than the library OS interface, it becomes more challenging to defend against attacks.

The standard C library has an interface spanning several thousand functions. Moreover, this interface has also been changing as the library implementation evolves. For example, we studied the `glibc` repository and observed the number of API calls across ten versions of the library, and the number of APIs added or removed from each version. Table 2 presents our results. As this table shows, the library wrapper interface is over two orders of magnitude larger than the library OS enclave interface, and evolves significantly over the ten generations that we studied. Consequently, a library wrapper implementation such as Panoply must also be modified and tailored for each version of the library.

It is also important to note that standards such as POSIX and ISO only specify the library's function-call interface, but leave unspecified the definitions of certain data structures. These data structures can change from one library version to another even if the interface remains POSIX or ISO-compliant. Changes to these data structures will require modifications to the library wrappers to suitably marshal/unmarshal the data as it crosses the enclave boundary. As an example, Panoply does not support the `FILE` structure. Any application code that uses the `FILE` structure must be rewritten to use a placeholder variable of type `int`, which is then translated to the corresponding `FILE` structure in the library (outside the enclave) via a table lookup (see Section 4-RQ1).

In addition to a large interface, the Panoply prototype requires invasive changes to applications to be modified to leverage its library wrappers. All calls to the library within the application must be modified by calls to the Panoply library instead. The Panoply authors report modifications on the order of about a 1000 lines for the benchmarks that they report in their paper.

Although modifying applications does impose a burden on a software developer wishing to quickly prototype an enclave-protected version of the application, the resulting engineering effort may sometimes be used to improve application performance as well.

Table 2: The evolution of the standard C library (glibc) interface across several versions. This table shows the number of API calls in each version, and the number of API calls added (+) or removed (-) from the prior version.

#Version	# API size	# added/removed
2.20	2021	+1/-2
2.21	2023	+2/-0
2.22	2032	+9/-0
2.23	2043	+12/-1
2.24	2046	+3/-0
2.25	2058	+13/-1
2.26	2073	+32/-17
2.27	2110	+37/-0
2.28	2126	+18/-2
2.29	2128	+2/-0

For example, we observed that the authors of Panoply had significantly modified the code of `openssl` in the process of porting it to work within the enclave. For example, aside from the modifications to use Panoply library wrappers, the Panoply version of `openssl` also replaces the random number generation code with calls to the Intel SDK's random number generator, which in turn calls the `rdrand` x86-64 instruction to obtain random numbers. This prevents a domain crossing, and is also more secure than relying on the underlying untrusted OS to provide random numbers.

3.3 Instruction Wrapper Model

The instruction wrapper model takes a middle-ground between the library OS and library wrapper approaches in providing an enclave interface. It provides wrappers for instructions that are forbidden for use within the enclave, *i.e.,* the instructions in Table 1. The wrappers perform marshaling of arguments, and forward them to supporting code outside the enclave, which executes the instructions on behalf of the enclave. In contrast to the library wrapper model, the marshaling happens at a much lower level of abstraction, *i.e.,* at the level of registers and memory.

In theory, this approach can be made to work on arbitrary binaries by replacing all occurrences of the instruction in the enclave code with the wrapper. However, practical implementations of the instruction wrapper model, (*e.g.,* SCONE [4]) make the observation that applications rarely use these instructions in their raw form. Rather, the applications are programmed to use libraries, which in turn execute these low-level instructions on their behalf. Thus, they wrap the occurrences of these instructions within the library. Applications simply link against these libraries to leverage the instruction wrappers.

By implementing wrappers at a much lower level (*i.e.,* wrapping instructions rather than providing library call wrappers), it works on a much slower changing interface. To take an example, we consider the `syscall` instruction, which is used to implement system calls. Naturally, the wrappers for the `syscall` instruction depend on which system call is being invoked (*e.g.,* because the number of arguments to each system call are different). As Table 3 shows, the system call interface on Linux is both much narrower and much stabler over kernel versions as compared to the glibc interface (which was shown in Table 2).

The instruction wrapper approach shares some of the benefits of the library wrapper model. The main difference between this model and the library wrapper model is that the standard C library executes within the enclave. As a result, instruction wrappers have

Table 3: Evolution of the system call interface across versions of the Linux kernel.

#version	# system calls	# system calls added
v4.15	333	0
v4.16	333	0
v4.17	333	0
v4.18	335	2
v4.19	335	0
v4.20	335	0
v5.0	335	0
v5.1	339	4
v5.2	345	6

Table 4: Applications used in our evaluation.

Application	Description
`bzip2-1.0.6`	File compression utility
`memcached-1.5.20`	Key-value store
`openssl-1.0.1m`	OpenSSL cryptography library
`h2o-2.0.0`	HTTP Web server
`cpython-3.7`	Python interpreter in C

a somewhat larger TCB than Panoply. Instruction wrappers must still implement a shim to protect against IAGO-like attacks on the (much narrower) instruction return interface. This model does not require invasive application-level modifications for rapidly creating an in-enclave prototype. However, application-level modifications may be necessary to optimize its performance, and as our evaluation shows, the instruction-wrapper model also lends itself well to any future re-engineering of the application.

The primary costs of the model are that it has a somewhat larger TCB than Panoply (although a much smaller TCB than the library OS model). Applications also need to be re-linked to use libraries in which the low-level instructions forbidden within SGX enclaves are wrapped.

SCONE [4] uses instruction wrapping. However, its implementation is not publicly available (SCONE is the basis for a commercial offering from `scontain.com`). For our study, we therefore built an in-house replica of SCONE (which we will refer to as Porpoise). Like SCONE, Porpoise is implemented by wrapping instructions from Table 1 in the `musl-libc` [17] version 1.1.9 implementation of the standard C library. However, unlike SCONE, the version of Porpoise used for this paper does not incorporate asynchronous system calls and does not integrate with Docker. Porpoise supports threads within the enclave using a `pthreads`-like API. The developer pre-specifies the number of threads used by the enclave, and these are mapped 1:1 to threads outside the enclave. This differs somewhat from the M:N threading model proposed in SCONE. We have archived the version of Porpoise used for the experiments reported in this paper (DOI:`10.5281/zenodo.3895761`).

4 EVALUATION

We now quantitatively evaluate the costs and benefits of the three models discussed in the previous section. Our methodology is as follows: We consider one concrete prototype as a representative of each model—Graphene-SGX for the library OS model, Panoply for the library wrapper model, and Porpoise for the instruction wrapper model, and port a suite of applications (see Table 4) to enclaves using each of these models (not all applications work in all settings, as we will see) to answer the following research questions:

Table 5: Evaluating the ability to port applications to enclaves using each framework (RQ1).

Application	Graphene-SGX	Panoply	Porpoise
`bzip2`	✔	✔	✔
`memcached`	✔	✖	✔
`openssl`	✔	✔	✔
`h2o`	✔	✔	✔
`cpython`	✔	✖	✔

(RQ1) Porting effort. From an application developer's point of view, what is the effort required to port the application and get it running within the enclave?

(RQ2) Application re-engineering effort. Suppose that an application developer wishes to re-engineer the application by deciding that he only wants to run a portion of the application within the enclave. After the application developer has decided what code to run within the enclave, what is the amount that he needs to invest to get the code running within the enclave?

(RQ3) Security. How much trusted code runs within the enclave, in addition to the application's own enclave code?

(RQ4) Runtime performance. What is the runtime performance overhead of each of these approaches, and how do they compare to native execution (*i.e.,* executing the code without enclaves)?

4.1 RQ1: Porting Effort

To answer RQ1, we attempted to port the applications from Table 4 using each of the three methods. For this research question, we ported the *entire* application to run within the enclave. The application that starts the enclave is simply a dummy `main` function, that starts the application, but then has no further role to play in the execution of the application. Other than this `main` function, only the untrusted portion of the Intel SGX SDK, and any other code required by the framework (*e.g.,* the untrusted shim of Porpoise) run in user-space. Table 5 presents the results of our experiments.

Of the three methods, Graphene-SGX provides the least-effort porting experience. We simply wrote a manifest file that describes (among other things) the set of libraries used by the benchmark, and Graphene-SGX is able to execute the applications. The effort to port these applications to run with Porpoise is comparable to that of Graphene. We compiled the application with Porpoise's version of `musl-libc`, and built it as a statically-linked, position-independent binary. We ported all five applications to Porpoise.

Panoply is the most cumbersome of the three approaches to which to port applications. In fact, the authors of Panoply themselves ported `openssl` (version 1.0.1) and `h2o` (version 2.0.0), and reported having to modify 307 SLOC and 154 SLOC in these applications, respectively [25]. In our experiments, we used the same codebase for `openssl` and `h2o` as provided by the authors of Panoply. As a result, we chose `openssl` version 1.0.1 and `h2o` version 2.0.0 for our experiments with Graphene-SGX and Porpoise (so that we could compare them across the same versions for our research questions), even though both Graphene-SGX and Porpoise can execute the latest versions of both `openssl` and `h2o`.

To understand the complexity of porting an application afresh to Panoply, we attempted to port `bzip2` and `memcached` from scratch.

While we were successful in our attempt with bzip2, we were unable to get memcached executing successfully in Panoply. We found that bzip2 uses 10 API calls from the standard C library that were not wrapped in the Panoply prototype available to us (Panoply currently has about 250 wrappers implemented out of about 2000 functions in the libc interface); we therefore implemented these 10 wrappers. memcached similarly invokes a libevent library that consists of 69 interfaces for which we had to write wrappers.

In addition to requiring more wrappers, we found that bzip2 uses the FILE structure in its code. The FILE structure includes several pointers. Panoply does not currently support such structures that have pointers in them as part of its library wrapper interface. Instead, it maps each FILE structure to an index (of type int), and uses the index as part of the library wrapper. The Panoply code outside the enclave uses the index to look up the FILE structure, now maintained outside the enclave, and performs the operation. We therefore had to modify bzip2 to replace all occurrences of the FILE structure with an index instead. In all, this and other changes required 149 SLOC of modifications to bzip2. We found that memcached also has similar deeply nested structures (*e.g.,* as part of the libevent library's interface) that are difficult to port easily using Panoply. We were unable to port memcached despite adding wrappers for 30 additional libc interfaces. We note that these challenges expose a foundational difficulty in the library wrapper method—because the library interface is large, and supports complex data structures, writing library wrappers is fundamentally a difficult task. This task is further complicated when the library wrappers must evolve as the library interface itself evolves (see Table 2). Moreover, wrappers must be provided for every additional library (*e.g.,* libevent) that the application may use.

In addition to these challenges, we were also hampered by some implementation-related quirks of Panoply. In particular, Panoply itself relies on the Intel SGX SDK to support enclaved execution. The Intel SGX SDK provides support for a subset of BSD libc. Since this subset is not sufficient to support the applications considered in the Panoply paper, the authors added support based on the applications needs. However, they chose to use GNU libc for this purpose. This mismatch creates a number of complications because the GNU and BSD versions differ in the set of headers required during compilation. For example, the definition of fields in struct timespec in BSD and GNU libc usea different set of macros in their type definitions, and these macros are expanded in different header files in the BSD and GNU libc libraries. When the application developer encounters such a situation, he must manually identify which header file (BSD's or GNU's) to include. Because the same set of definitions do not appear in the same set of files across these library versions, blindly including a header file may also lead to name clashes that must be resolved manually. Given this rather time-consuming experience attempting to port bzip2 and memcached to Panoply, and the number of invasive source code changes needed, we did not attempt to port cpython to Panoply, because it uses many more interfaces from the standard C library.

Summary (RQ1)—The library OS and instruction-wrapping approaches provide a seamless enclave-porting experience. The library-wrapping approach, as implemented by Panoply, requires modifying a few hundred lines of code within the application. Additional libraries used by the application will require developing new wrappers.

Table 6: Number of new interfaces required and code added for application re-engineering with Porpoise (RQ2).

Application	#Interfaces	SLOC added
bzip2	3	29
openssl	1	8
cpython	24	277

4.2 RQ2: Application Re-engineering Effort

While RQ1 concerned the effort to port an application in its entirety into the enclave, RQ2 concerns the effort that an application developer would invest to port an application after deciding to re-engineer it. For example, an application developer may decide that it is not necessary to execute the entire application inside the enclave, and that it suffices to execute just certain security-critical parts of it within the enclave. RQ2 asks the following question: how different is the experience in building enclave code with each of these frameworks after such application re-engineering?

The research literature does contain examples of tools that assist with such porting, notably Glamdring [13], Civet [32], and the gcc-based tool described by the authors of lxcsgx [28]. These tools assist the programmer with the core task of re-engineering the application. Given a specification of sensitive data that must be protected (*e.g.,* in the form of annotations), and hence execute within the enclave, these techniques use static taint analysis to identify the other dependent code that must also execute within the enclave. Note that these tools assume the existence of an enclave execution framework.

Our goal in RQ2 is *not* to identify the sensitive data that must execute within the enclave or assess the difficulty of splitting the application. Rather, assuming that a suitable split has been identified, we wish to determine how much effort it is to re-engineer the application after such a split has been identified. We assume that the split is identified at the function-level of granularity, *i.e.,* certain functions have been identified to execute within the enclave, while the rest execute within the untrusted code. Because we manually analyzed the applications to identify this split for RQ2, we restricted ourselves to three of the application benchmarks, *viz.,* bzip2, openssl, and cpython, as described below.

- bzip2. We split bzip2 so that only the main file compression algorithm executes inside the enclave. This results in an enclave interface of three functions, BZ_compressInit, BZ2_compress and BZ2_compressEnd, with which the the re-engineered bzip2 application interacts with the compression algorithm
- openssl. We moved the functionality that generates RSA keys to the enclave. The enclave interface to do so consists of just one function, (genrsa_main).
- cpython. The cpython application consists of code to parse, compile and then interpret an input python program. In real-world settings, the interpreter is responsible for running the python code on sensitive data. Therefore, we decided to port only the interpreter to the enclave. The interpreter's enclave interface has 24 functions.

We only re-engineer our benchmarks to execute atop Porpoise and Panoply (only atop Porpoise for cpython), where the effort to build the enclave part of the code after splitting is comparable. This

```
filename: enclave.edl
public ecall_PyArena* PyArena_New(void);

filename: pythonrun.c
PyArena * ecall_PyArena_New(void);
#define PyArena_New ecall_PyArena_New

filename: function_wrapper.c
PyArena* ecall_PyArena_New(void){
    PyArena *ret = NULL;
    sgx_status_t status = SGX_SUCCESS;
    status = ecall_PyArena_New(enclave_id, &ret);
    // ret will point to a buffer that stores
    // the return value from the enclave
    assert(status == SGX_SUCCESS);
    return ret;
}
```

Figure 2: Example of new code required to introduce an enclave interface in `cpython` with Porpoise (RQ2).

is because both Porpoise and Panoply have a similar enclave/non-enclave interaction interface.

Figure 2 shows the new code that we write to create a new enclave interface in Porpoise for the `cpython` application. As this code shows, an application that wishes to invoke a function in the enclave interface must simply perform an `ecall` to that function together with its arguments, and the enclave ID. A new entry for this interface is also included as part of the enclave's interface definition file. Table 6 shows the number of lines of such code that we had to write to create enclaves for each re-engineered application.

We did not attempt to re-engineer our benchmarks to run atop Graphene-SGX. This is because Graphene-SGX was originally designed to run applications in their entirety within the enclave. This is reflected in the design of their enclave interface, which is a low-level interface that communicates with a platform-specific adaptation layer (called the Graphene-PAL). By design, the Graphene-PAL invokes a fixed entrypoint inside the enclave, typically the equivalent of the `_start` function in a traditional application. As a result, re-engineering an application to work atop Graphene-SGX, with part of its code running in the enclave, would require invasive changes to the Graphene-SGX platform itself—an activity that we did not wish to undertake, since our goal is to understand the platforms as-is.

It would be possible to re-engineer an application atop Graphene-SGX, so that the sensitive portion runs in its own process (with its own enclave), and interacts over IPC with the remaining parts of the application. However, this would require a fundamental rewrite of the application to make it a distributed client/server system. We view this change as being rather invasive to the application's code base, and therefore do not evaluate this method.

Summary (RQ2)—The effort required to re-engineer applications with the library-wrapping and instruction-wrapping models is similar. The enclave interface exposed by library OSes does not facilitate easy re-engineering of the application into an enclave and non-enclave portion.

4.3 RQ3: Security Evaluation

We evaluated the amount of trusted code that must execute within the enclave for each of the three frameworks. For RQ3, we did not consider the trusted code of the application itself (*i.e.,* its enclave code), because that number would depend on the application itself, and how the developer has decided to engineer the enclave. Rather, we only consider the code that is core to the framework itself. In addition, both Panoply and Porpoise use the Intel SGX SDK to bootstrap basic enclave functionality (Graphene-SGX does not), and this code is therefore part of their trusted code base. SCONE [4], an instruction-wrapping framework, also does not use the Intel SGX SDK within the enclave, relying instead on a home-grown library for basic enclave functionality. The source code for SCONE is not publicly available, but their paper reports a TCB of size of approximately 187,000 lines of code for the version of SCONE that implements shielding against IAGO-style attacks.

Table 7: Amount of trusted (and untrusted) code that executes within each of the frameworks (RQ3). `glibc` refers to `glibc-2.27`, while `musl` refers to `musl-1.1.9`.

Graphene		Panoply		Porpoise	
Comp.	**SLOC**	**Comp.**	**SLOC**	**Comp.**	**SLOC**
Trusted Code					
LibOS	31,742	Shim	14,506	Shim	1,934
glibc	1,222,912	-		musl	82,978
-		Intel SDK	119,545	Intel SDK	119,545
Untrusted Code					
PAL	40,493	Panoply shim	3,004	Porpoise shim	1,209

Table 7 presents the results of our evaluation. It shows the number of lines of trusted code that executes in each of these frameworks (measured using the `sloccount` utility). It also shows the amount of untrusted support code provided by the infrastructure (*i.e.,* the code that executes outside the enclave, and interacts with the enclave).

We see that Panoply emerges as the framework that reduces the amount of code that executes within the TCB. However, they do this at the cost of implementing wrappers at the library level of abstraction, which means that many more wrappers have to be written and that these wrappers have to evolve as the libraries evolve. Recall from Table 2 and Table 3 that the library API evolves much more than the relatively-stable system-call API. Thus, while Porpoise has more trusted code (in particular, the `musl-libc` library, which it modifies), its interface is more stable and requires fewer changes as the code evolves. Graphene-SGX requires the most trusted code, because it includes the library OS in the enclave.

The enclave-execution framework is an application's interface the untrusted environment outside the enclave. It must shield the enclave by providing security. Shields proposed in prior work implement, encryption by default for file system and network traffic [4] and check return values [5, 25, 31] to protect the enclave from IAGO attacks [6]. For example, the file system shield of Graphene-SGX adds a message-authentication code (MAC) to files stored outside the enclave. When it needs to open a file, it memory maps the file, and checks the MAC, and then copies the file into in-enclave memory. All further file operations are directed to this copy, and the file is copied back from the enclave to the memory mapped region when it is closed. SGX-LKL goes a step further, and implements an in-enclave file system that encrypts, shuffles, and adds a MAC to each disk block when it sends that block outside the enclave for storage. Disk blocks retrieved from outside are checked and decrypted within the file system.

It is difficult to quantitatively compare the implementation of such shields head-to-head across frameworks. This is because the checks are often included as part of the core functionality of the

Table 8: Workloads used to run applications (RQ4).

Application	Workload
bzip2	zipping and unzipping files of various sizes
memcached	memtier_benchmark [15]
openssl	HMAC (md5), DES-CBC, AES-256-CBC, SHA-256, MD5, RSA-2048-sign and RSA-2048-verify from openssl speed benchmark
h2o	wrk2 http workload generator [36]
cpython	benchmarks from pyperformance using timeit

framework itself (*e.g.,* an encrypted file system in SGX-LKL). This functionality itself adds to the size of the TCB executing in the enclave. Nevertheless, if we were to use the width of the enclave interface as a metric (because it determines the number of distinct functions that require shielding), the library OS model provides the thinnest interface to be protected (*e.g.,* 38 in Graphene-SGX, and only 7 in SGX-LKL [21]). The checks are also logically simple, *e.g.,* for the file system, checking MAC values after decrypting disk blocks that are read, or affixing MAC values to and encrypting the disk blocks that are written. The instruction wrapper interface is somewhat bigger, and SCONE implements checks for different system calls related to the file system and network. The checks here work at the system call interface, and therefore work atop those abstractions, *e.g.,* files rather than disk blocks as in the library OS model. The library wrapper model has the widest enclave interface and would likely require shields to extensively cover the library interface. However, we could not directly verify this fact. While the Panoply paper claims that they implemented shields in the library interface, the publicly-available prototype implementation does not include these checks.

Summary (RQ3)—Library-wrappers, as implemented in Panoply, require the least amount of support code within the enclave. However, this code must also evolve to support changes to the library API. The instruction-wrapper approach requires more code within the enclave, but is likely to be stabler with respect to code evolution. The library OS approach requires the largest amount of trusted code within the enclave.

4.4 RQ4: Runtime Performance

We conducted experiments to understand the runtime performance implications of the three models. We studied both the overall performance impact on the applications that we ported, as well as microbenchmarks to stress the enclave/non-enclave interface.

We conducted all our experiments on an Intel(R) Core(TM) i7-7700 CPU (3.60GHz) with 4 cores and 2 threads per core (8 hypercore) and an 8192KB cache and 16GB of RAM. We used Ubuntu 16.04 LTS (Linux 4.4.0-169) as the underlying OS for Graphene-SGX, Panoply and Porpoise. (Panoply works only atop Linux 4.4.0-169; we therefore used it for our evaluation. However, Porpoise works even on the latest version of the Linux kernel (5.3.8).) For experiments with our application benchmarks, we run the entire application within the enclave under each of the frameworks. The machine runs only the application under test, and all the reported numbers are an average of five runs. However, we are unable to report performance numbers for each application on all frameworks, because we were unable to port all the applications to Panoply. Table 8 shows the workloads with which we ran the applications, while Table 9 reports the results of our experiments

Table 9 shows the performance of each of the applications, running the benchmarks from Table 8 on Graphene-SGX, Panoply (where applicable), and Porpoise. It also shows the native performance of the application, *i.e.,* when run outside the enclave. We find that across the benchmarks, neither Graphene-SGX, Panoply nor Porpoise consistently outperforms the other. For example, on memcached and h2o, Porpoise provides higher throughput and lower latency than Graphene-SGX. However, Graphene-SGX outperforms Porpoise on bzip2 and python.

The bzip2 benchmark reads files in fixed-size chunks as it passes the file contents to the enclave. Thus, as the file size increases, the number of enclave/non-enclave interactions increases. However, Graphene-SGX, being a library OS implements file caching techniques, which fulfil some of the read requests, thereby ameliorating the number of domain crossings required. For the bzip2 benchmark, Panoply offers performance roughly comparable to Graphene-SGX, both outperforming Porpoise, especially as file sizes increase. This is because Porpoise executes the standard C library inside the enclave, in contrast to Panoply, which executes it outside the enclave. A number of library calls such as fopen, read, and so on, result in multiple domain crossings for Porpoise, and the number of such domain crossings increases with file size. In contrast, the number of domain crossings does not grow proportionally with the file size in the case of Panoply, thus explaining the performance difference.

For openssl, we found that Panoply outperforms both Graphene-SGX and Porpoise. This is because the version of openssl ported to Panoply makes extensive changes to optimize the performance. For example, as explained earlier this version replaces the random number generation code included in the openssl release with calls to sgx_read_rand, a function provided by Intel SGX SDK, which uses the rdrand hardware instruction to source randomness.

Both memcached and h2o run I/O intensive workloads, making them both network-bound. For both these benchmarks, we find that Graphene-SGX is significantly slower than Porpoise in terms of both throughput and latency. In case of h2o, the version running atop Graphene-SGX saturates at around 10,000 requests per second, with all requests exceeding that threshold being dropped by the Web server. The latency is also rather large, at 132ms per request. Both the versions running atop Panoply and the Porpoise are able to sustain a larger number of requests per second, however, the Panoply version saturates at around 33,000 requests per second, offering a latency of more than 1 second per request. The version running on Porpoise saturates above 40,000 requests per second. With memcached, the version on Porpoise outperforms the version on Graphene-SGX even as we vary the number of server threads.

Evaluation with Microbenchmarks. To understand the raw overheads of enclave crossings imposed by each of the frameworks, we also evaluated them with microbenchmarks. Our microbenchmarks consist of enclave code that cause an enclave crossing by requesting the execution of a system call a million times. We considered a set of system calls shown in Table 10.

We find that Graphene-SGX offers the best performance for sequential read operations, comparing favourably even with native performance. This is because Graphene-SGX, being a library OS, implements several file caching techniques that avoid costly domain crossings. However, the benefit of these optimizations does not apply when the read operations seek to random locations in the

Table 9: Comparing the performance of various application benchmarks running atop Graphene-SGX, Panoply and Porpoise (RQ4). `memcached` and `cpython` are not available atop Panoply (see Table 5).

Application	Workload	Graphene-SGX	Panoply	Porpoise	Native (no enclaves)
`bzip2`	File size (MB)	Time (s)	Time (s)	Time (s)	Time (s)
	16	4.134	9.410	5.585	1.979
	32	7.212	12.738	9.592	3.971
	64	11.732	17.688	17.850	7.957
	128	21.157	27.690	36.438	17.373
	256	44.347	49.921	67.302	32.578
`memcached`	memtier params	Throughput (MBps), Latency (ms)	-	Throughput (MBps), Latency (ms)	Throughput (MBps), Latency (ms)
		#Threads | Throughput | Latency		#Threads | Throughput | Latency	#Threads | Throughput | Latency
	4 threads, 50 connections/thread, 10,000 req/-client	1 | 0.4 | 9.77 2 | 1.15 | 3.83 3 | 1.67 | 2.52 4 | 2.37 | 1.81 5 | 2.42 | 1.86	N/A	1 | 2.11 | 2.73 2 | 4.24 | 1.70 3 | 6.46 | 1.17 4 | 8.23 | 0.79 5 | 7.78 | 0.78	1 | 6.46 | 0.95 2 | 12.38 | 0.51 3 | 17.83 | 0.39 4 | 16.34 | 0.41 5 | 11.32 | 0.49
`openssl`	Workload	Throughput (KBps)	Throughput (KBps)	Throughput (KBps)	Throughput (KBps)
	HMAC (md5)	390,626	416,091	381,300	455,747
	DES-CBC	103,768	97,886	103,723	103,883
	AES256-CBC	250,578	135,932	250,406	259,107
	SHA-256	188,996	238,508	183,482	186,413
	MD5	242,736	405,965	242,270	373,865
	RSA-2048-sign	303.8 ops/sec	1284.7 ops/sec	305.6 ops/sec	327.0 ops/sec
	RSA-2048-verify	14,540.5 ops/sec	37,678.7 ops/sec	14,562.2 ops/sec	14,893.2ops/sec
`h2o`	Requests/sec	Latency (ms)	Latency (ms)	Latency (ms)	Latency (ms)
	10,000	132	1.61	1.28	1.29
	20,000	*	1.61	2.30	1.32
	30,000	*	2.65	1.93	1.36
	40,000	*	> 1sec	18.21	1.77
	50,000	*	> 1sec	> 1sec	2.13
`cpython`	`pyperformance`	Time (ms)	Time (ms)	Time (ms)	Time (ms)
	`html5lib`	260		269	106
	`pyres`	1940		1980	890
	`json_load`	0.348	N/A	1.030	0.171
	`float`	432		396	122
	`fannkuch`	1805		1806	566

Table 10: Measuring the performance of the frameworks using microbenchmarks. The time reported (in seconds) is for 1 million executions of the system calls (RQ4).

syscall	Graphene-SGX	Panoply	Porpoise	Native
read (sequential)	0.405	3.133	4.295	0.209
write	13.268	3.471	4.742	0.609
open+close	24.946	6.973	9.192	1.103
lseek+read	3489.270	6.186	8.637	0.716
gettimeofday	4.134	2.479	3.668	0.019
getpid	0.0707	2.549	3.9308	0.0022

file, as shown in Table 10. In this case, both Porpoise and Panoply outperform Graphene-SGX. Graphene-SGX also offers poorer performance for `write` and `open+close`, likely due to the additional operations within its shield. The performance of Panoply and Porpoise is roughly comparable.

Summary (RQ4)—No one model appears as the clear favourite with respect to runtime performance. Using a library OS can provide the benefits of file caching for some applications. Both the library-wrapping and instruction-wrapping models perform better for network-bound applications than the library OS model.

4.5 Threats to Validity

The primary threat to validity of our reported results comes from the fact that we have only evaluated one representative prototype for each of the enclave execution models. While the community has invested effort in building frameworks to allow new applications to be written (*e.g.,* [3, 9, 18, 23, 35]), these frameworks are not readily suited to run legacy code in enclaves, which is the primary focus of this paper. For the enclave execution models that we considered, the number of open-source frameworks available is limited—only Panoply for the library wrapper model. SCONE was the only the instruction wrapper prototype available when we started the work (which we reimplemented as Porpoise), and although we became aware of lxcsgx [28] after we built Porpoise, its source code is not yet available. However, Porpoise and lxcsgx were substantially similar, and we decided to use Porpoise for our evaluation. SGX-LKL [21] is a recent implementation of the library OS model that offers a substantially smaller enclave boundary than Graphene-SGX. However, the application developer's experience to port applications with both Graphene-SGX and SGX-LKL (RQ1 and RQ2) will be similar, and those results will also hold for SGX-LKL.

A second threat to validity is that we have only considered five applications for our study. Although we believe that these are representative of applications used in real-world, the results may differ if another set of applications is used. Moreover, even for the applications that we considered, we have not attempted to verify whether the functionality of the version ported to the enclave is equivalent to the native application. In particular, our experience with each ported application was restricted to the workloads on which the application was executed.

5 RELATED WORK

Since we have already discussed various frameworks to port legacy applications [4, 5, 21, 25, 28, 31], we will focus our discussion in this section on other related work.

Porting applications to enclaves. Prior work has ported applications in several specific domains to enclaves. These include frameworks for MapReduce tasks [24], language environments for JavaScipt [8] and Rust [7], BitCoin and Blockchain applications [30, 37], in-memory databases [20], object stores [12], and middleboxes [29]. SCONE [4] and lxcsgx [28] use the instruction-wrapping model to port containers (Docker in case of SCONE and lxc containers in case of lxcsgx) into enclaves.

Each of these projects focused on providing an enclave version of a *specific* application or class of applications. Our focus, in contrast, is on generic frameworks that can be used to port any application to

enclaves. Naturally, because the projects cited above are tailored to individual applications, we expect the resulting enclaves to perform better than applications ported using the generic frameworks. Since they are tailored from scratch for specific application domains, they also are better engineered to just run the sensitive portions of the applications within the enclaves, rather than the entire application, as we did in this paper. Thus, we view the frameworks discussed in this paper as stop-gaps, that can be used to get enclave applications up and running quickly, as developers work on rewriting the applications to tailor them to enclaves.

Finally, there is also work on creating secure in-enclave file systems [1, 16]. The main goal here is to ensure that file accesses that cross the enclave boundary do not reveal any information about the data being accessed inside the enclave. These file systems use techniques inspired from the oblivious-RAM literature to provide such guarantees. The frameworks discussed in this paper do not offer such guarantees, and may leak information (*e.g.,* about the files accessed, the number of bytes read) even if the data is stored encrypted in the files. However, they can be integrated with such ORAM-inspired file systems to provide stronger guarantees.

Tool support to write enclave code. Tools such as Glamdring [13] and Civet [32] offer support to automatically partition a legacy application into an enclave and non-enclave part. An application developer provides a specification of the portions of the code/data that are sensitive. These tools perform static taint analysis of the application to identify data and control dependencies, and identify the enclave boundary. Following this, they automatically partition the code and create the enclave; they rely on one of the models described in this paper as the enclave execution framework.

Researchers have also developed tools to help developers build secure enclaves from scratch. The key consideration for these tools is to ensure that enclaves do not accidentally leak sensitive data to untrusted code. Sinha *et al.* [26] developed a programming framework that ensures information-release confinement, *i.e.,* that cleartext data never leaves the enclave. To do this, they provide a verified, in-enclave trusted library and a simple API consisting of just a few simple calls (`send`, `recv`) to interact with non-enclave code. Provided that an application developer adheres to this simple interface, they can guarantee that the enclave will not accidentally leak data to untrusted code. Moat [27] is a similar analysis tool that analyzes the machine code of enclaves and determines whether there are any unintended information leaks to untrusted code.

6 CONCLUSIONS

No clear consensus has emerged thus far in the community on the right abstractions for enclave programming, especially as concerns porting legacy code to enclaves. We considered the three models that have been proposed in the research literature, namely the library OS, library-wrapping and instruction-wrapping models. Based on our experience porting a number of application benchmarks to Graphene-SGX, Panoply, and Porpoise, we conclude that the choice of the enclave programming model to be used depends on the factors that application developers wish to optimize for:

- *Rapid prototyping.* Developers may wish to quickly prototype an in-enclave version of their application. This can serve as a stop-gap solution that provides the benefits of enclaves as a team develops a version of the application customized for the enclave. The library OS and instruction wrapper models are ideally suited for this setting.
- *Source code availability.* With a legacy application, source code may often be unavailable or recompilation may not be feasible, *e.g.,* due to library compatibility issues. In such settings, only the library OS model allows developers to create enclave versions of the application. Both the instruction wrapper model and the library wrapper model either require access to source code, or require the legacy binary to be linked with suitable wrappers.
- *Flexibility to re-engineer.* With a quick first-cut of their application executing in the enclave, application developers may wish to optimize the execution of the enclave, *e.g.,* by reducing the number of domain crossings, or reducing the amount of code executing in the enclave. It goes without saying that the application's source code is required for such re-engineering. Both the instruction wrapper and library wrapper models are best suited for this setting.
- *Security Concerns.* Application developers may wish to reduce the amount of code executing within the enclave in an effort to reduce the size of the attack surface of their security-critical code. Only the Panoply implementation of the library wrapper model optimizes for this criterion. However, it also entails executing much of the standard C library outside the enclave (Figure 1), and the in-enclave application must take suitable precautions when it makes library calls (*e.g.,* shields for IAGO attacks).
- *Performance.* In our evaluation, no one model emerged as a clear winner with respect to runtime performance, and the developer must choose the enclave programming model that works best for the application at hand. Library OSes can provide good performance for some applications, *e.g.,* by offering caching and avoiding domain crossings, as saw with `bzip2` and `cpython`. However, because enclave execution by itself imposes overheads, and library OSes execute entirely within the enclave, they may also offer poor performance in some cases, as we saw with `memcached` and `h2o`. The instruction wrapper and library wrapper models do offer the potential for better performance if software developers have the flexibility to profile and re-engineer their applications by reducing domain crossings.
- *Support for evolution.* Finally, with respect to code evolution, the library OS and instruction wrapper models are better suited with respect to application code evolution than the library wrapper model. As discussed in the paper, the system call interface evolves much slower than the library call interface, thereby allowing the library OS and instruction wrapper models to provide better support than the instruction wrapper model as the enclave application code evolves to newer versions.

ACKNOWLEDGMENTS

We thank the ESEC/FSE'20 reviewers for their thoughtful feedback. We thank Sujay Yadalam and Arkaprava Basu for early adoption of Porpoise in the SG^{XL} project and providing feedback during its development. This work was funded in part by a grant from the Robert Bosch Centre for Cyber-Physical Systems, IISc Bangalore, an unrestricted research grant from Intel India, and by a Ramanujan Fellowship from the Government of India.

REFERENCES

[1] Adil Ahmad, Kyungtae Kim, Muhammad Sarfaraz, and Byoungyoung Lee. Obliviate: A data oblivious filesystem for intel sgx. In *Networked and Distributed Systems Security Symposium*, 01 2018.

[2] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative technology for CPU based attestation and sealing. In *Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[3] Anjuna–hardware-grade runtime security. https://www.anjuna.io.

[4] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure Linux containers with Intel SGX. In *ACM/USENIX Symposium on Operating System Design and Implementation*, 2016.

[5] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with Haven. *ACM Transactions on Computer Systems*, 33(3), September 2015.

[6] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. In *Proceedings of the 2013 International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[7] Yu Ding, Ran Duan, Long Li, Yueqiang Cheng, Yulong Zhang, Tanghui Chen, Tao Wei, and Huibo Wang. Poster: Rust sgx sdk: Towards memory safety in intel sgx enclave. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2491–2493, 2017.

[8] David Goltzsche, Colin Wulf, Divya Muthukumaran, Konrad Rieck, Peter Pietzuch, and Rüdiger Kapitza. Trustjs: Trusted client-side execution of javascript. In *Proceedings of the 10th European Workshop on Systems Security*, EuroSec'17, pages 7:1–7:6, 2017.

[9] Google asylo–an open and flexible framework for enclave applications. https://asylo.dev.

[10] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[11] Intel. Software guard extensions programming reference, revision 2, 2014. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[12] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer. Pesos: Policy enhanced secure object store. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, 2018.

[13] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. Glamdring: Automatic application partitioning for intel SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 285–298, Santa Clara, CA, July 2017. USENIX Association.

[14] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[15] A high-throughput benchmarking tool for redis and memcached.

[16] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy (SP)*, May 2018.

[17] `musl` - an implementation of the standard library for Linux-based systems. https://git.musl-libc.org/cgit/musl.

[18] Open enclave sdk. https://openenclave.io/sdk.

[19] D. Porter, S. Boyd-Wickizer, J. Powell, R. Olinsky, and G. Hunt. Rethinking the library OS from the top-down. In *ASPLOS*, 2011.

[20] C. Priebe, K. Vaswani, and M. Costa. EnclaveDB: A secure database using SGX. In *IEEE Symposium on Security and Privacy*, 2018.

[21] 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. In *arXiv:1908.11143*, August 2019.

[22] O. Purdila, L. Grinjincu, and N. Tapus. LKL: The linux kernel library. In *RoEduNet IEEE International Conference*, 2010.

[23] Rust SGX SDK of Fortanix. https://fortanix.com/company/news/pr/2019/02/fortanix-launches-open-source-intel-sgx-rust-sdk.

[24] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 38–54, Washington, DC, USA, 2015. IEEE Computer Society.

[25] S. Shinde, D. Le Tien, S. Tople, and P. Saxena. Panoply: Low-TCB Linux applications with SGX enclaves. In *Networked and Distributed Systems Security Symposium*, 2017.

[26] R. Sinha, M. Costa, A. Lal, N. Lopes, S. Rajamani, S. Seshia, and K. Vaswani. A design and verification methodology for secure isolated regions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016.

[27] R. Sinha, S. Seshia, S. Rajamani, and K. Vaswani. Moat: Verifying the confidentiality of enclave programs. In *ACM Conference on Computer and Communications Security*, 2015.

[28] Dave (Jing) Tian, Joseph Choi, Grant Hernandez, Patrick Traynor, and Kevin Butler. A practical intel sgx setting for linux containers in the cloud. In *ACM Conference on Data and Application Security and Privacy*, 2019.

[29] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. Shieldbox: Secure middleboxes using shielded execution. In *Proceedings of the Symposium on SDN Research*, SOSR '18, pages 2:1–2:14, 2018.

[30] M. Tran, L. Luu, M. Kang, I. Bentov, and P. Saxena. Obscuro: A bitcoin mixer using trusted execution environments. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, pages 692–701, 2018.

[31] C. Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *USENIX Annual Technical Conference*, 2017.

[32] C. Tsai, J. Son, B. Jain, J. McAvey, R. Popa, and D. Porter. Civet: An efficient Java partitioning framework for hardware enclaves. In *USENIX Security Symposium*, 2020.

[33] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. Cooperation and security isolation of library oses for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 9:1–9:14, 2014.

[34] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security Symposium*, 2018.

[35] H. Wang, P. Wang, Y. Ding, M. Sun, Y. Jing, R. Duan, L. Li, Y. Zhang, T. Wei, and Z. Lin. Towards memory safe enclave programming with Rust-SGX. In *2019 ACM Conference on Computer and Communications Security*, 2019.

[36] A constant throughput, correct latency recording http benchmarking tool variant of wrk. https://github.com/giltene/wrk2.

[37] Rui Yuan, Yu-Bin Xia, Hai-Bo Chen, Bin-Yu Zang, and Jan Xie. Shadoweth: Private smart contract on public blockchain. *Journal of Computer Science and Technology*, 33(3):542–556, May 2018.