Towards Memory Safe Enclave Programming with Rust-SGX

Huibo Wang[†1], Pei Wang[*], Yu Ding[*], Mingshen Sun[*], Yiming Jing[*], Ran Duan[*], Long Li[*],

Yulong Zhang[*], Tao Wei[*], Zhiqiang Lin[‡]

[†]The University of Texas at Dallas
[*]Baidu X-Lab
[‡]The Ohio State University

ABSTRACT

Intel Software Guard eXtension (SGX), a hardware supported trusted execution environment (TEE), is designed to protect security critical applications. However, it does not terminate traditional memory corruption vulnerabilities for the software running inside enclave, since enclave software is still developed with type unsafe languages such as C/C++. This paper presents Rust-SGX, an efficient and layered approach to exterminating memory corruption for software running inside SGX enclaves. The key idea is to enable the development of enclave programs with an efficient memory safe system language Rust with a Rust-SGX SDK by solving the key challenges of how to (1) make the SGX software memory safe and (2) meanwhile run as efficiently as with the SDK provided by Intel. We therefore propose to build Rust-SGX atop Intel SGX SDK, and tame unsafe components with formally proven memory safety. We have implemented Rust-SGX and tested with a series of benchmark programs. Our evaluation results show that Rust-SGX imposes little extra overhead (less than 5% with respect to the SGX specific features and services compared to software developed by Intel SGX SDK), and meanwhile have stronger memory safety.

CCS CONCEPTS

• **Security and privacy → Formal methods and theory of security**; **Systems security**;

KEYWORDS

SGX, Rust Programming Language, Memory Safety, Type System Soundness

ACM Reference Format:
Huibo Wang[†1], Pei Wang[*], Yu Ding[*], Mingshen Sun[*], Yiming Jing[*], Ran Duan[*], Long Li[*], Yulong Zhang[*], Tao Wei[*], Zhiqiang Lin[‡]. 2019. Towards Memory Safe Enclave Programming with Rust-SGX. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3319535.3354241

[1]The bulk of this work was done while the first author was interning at Baidu X-Lab.

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3354241

1 INTRODUCTION

Software developed with type unsafe languages such as C/C++ is subject to memory corruption, such as buffer overflow, integer overflow, double free, and use after free. Memory corruption attacks have been one of the most severe cyber threats for over 40 years. Numerous such attacks have occurred, including Morris Worm [1], stack smashing [26], return-into-libc [35], return oriented programming (ROP) [29] (and its variants such as BROP [10] and JIT-ROP [31]), jump oriented programming (JOP) [11], call oriented programming (COP) [27], and even data-oriented programming (DOP) [16]. It is likely that these attacks will continue to remain a major cyber threat for years to come.

Several years ago, Intel introduced Software Guard eXtensions (SGX) in its Skylake CPU, which provides application programmers the capability to execute code in a secure enclave, namely an isolated trusted execution environment (TEE) [24]. In particular, SGX isolates sensitive code and data from the operating systems, hypervisors, BIOS, and other applications. It guarantees confidentiality and integrity of enclave programs even when the systems software such as operating systems, hypervisors, and BIOS are compromised. However, SGX hardware does not guarantee any memory safety for the software running inside enclave, since they are still developed with memory unsafe languages such as C/C++ or assembly today.

As a result, SGX programs still face the traditional memory corruption vulnerabilities as does traditional software. This can seriously undermine the security guarantee provided by SGX, and allow attackers to violate the integrity and confidentiality of enclave programs. For instance, when the enclave code is implemented with memory corruption vulnerabilities, it has been shown that attackers can leverage return-oriented programming (ROP) [29] to perform memory hijacking to leak secrets [9, 21]. Although randomization-based approaches [28] have been proposed to mitigate such attacks, Biondo et al. [9] have also pointed out that SGX runtime inherently contains memory regions whose addresses are fixed, and it is therefore difficult to completely eradicate such threats.

Over the past decades, a large number of defense mechanisms have been proposed to defeat memory corruption attacks, such as Stack Canaries [13], Data Execution Prevention (DEP) [34], Address Space Layout Randomization (ASLR) [18], Control Flow Integrity (CFI) [6], so on and so forth. However, they all are imperfect for

various reasons such as the performance cost outweighing the potential protection, incompatibility with legacy systems, relying on changes in the compiler tool chain, or requiring the access of program source-code. It is thus imperative to look for other alternatives to secure SGX enclaves programs.

Since SGX is a brand new platform, where not as many legacy applications need support, it provides a perfect opportunity of using efficient memory safe system languages to develop enclave programs instead of relying on traditional defenses such as ASLR or CFI. Today, there are a variety of programming languages with memory safe features including Rust, Go, Swift, JavaScript, Java, Python, and Solidity. Enabling the use of these programming languages for SGX program development would be an appealing approach to terminate the memory corruptions inside enclave.

However, there are still enormous challenges when enabling a memory safe language for enclave software development. In particular, memory safe programming languages often contain memory-unsafe components, such as Java's JVM, Python's C libraries, Swift's Object-C runtime, JavaScript's garbage collection (GC) engine, Go's assembly code, and Rust's unsafe features. Second, not all components inside the enclave can be developed with memory safe languages, such as the SGX feature specific instructions (e.g., remote attestation) and also the performance critical cryptography operations, as they may have to be written in assembly languages.

Therefore, it is impractical to have every component inside SGX enclave to be memory safe. Practical memory safety inside enclave needs to be contained and layered. Rust-SGX, a new memory safe SGX SDK proposed in this work, is designed based on this observation. More specifically, we leverage Intel SGX SDK as the foundation that provides a set of full fledged SGX development APIs, and add a Rust layer on top of it. With Rust-SGX, enclave programmers can develop their programs in pure Rust, and Rust-SGX will bridge the gap between the Rust world and Intel SGX interfaces. Compared to the enclave programs developed by Intel SGX SDK, the enclave programs developed with Rust-SGX are significantly more secure at the application layer thanks to the memory-safe Rust language. Whereas at the library layer, they have equivalent security properties due to the same dependency on Intel SGX SDK.

Enabling Rust atop Intel SGX SDK is non-trivial. A key challenge we must solve is to make sure the foreign function interface (FFI) between memory-safe and memory-unsafe languages (i.e., Rust and C/C++) is secure. Rust-SGX addresses this issue via formalized modeling and proof of the memory safety of the FFI. At a high level, our formalization is inspired by both CCured [25] and the safe Java Native Interfaces (JNI) by Tan el al. [32], and we extend them to secure the critical boundary code bridging the safe Rust world and unsafe C/C++ world.

We have implemented Rust-SGX, and tested it with a number of benchmark programs. Our evaluation results show that Rust-SGX SDK only imposes imperceptible or modest overhead (less than 5% with respect to the SGX services compared to software developed by Intel SGX SDK). Rust-SGX has been released as an open source project. It has been widely employed by the community to develop memory-safe SGX enclaves since 2017. The community adoption with Rust-SGX has indicated that programming with Rust-SGX is productive, efficient, and reliable.

Contributions. To summarize, we make the following contributions in this paper:

- We present Rust-SGX, a practical and layered approach to exterminating memory corruptions for SGX enclave programs.
- We propose the use of type safety and formal proof to handle the unsafe components when dealing with a layered memory safety model.
- We have implemented Rust-SGX and evaluated it with a number of benchmark programs. Our evaluation results show that Rust-SGX fully preserves the SGX functionality and meanwhile does not impose any significant performance overhead.

Roadmap. The rest of this paper is organized as follows. In §2, we provide necessary background related to Intel SGX and Rust programming language. §3 describes the objectives, threat model and scope, challenges and architecture of Rust-SGX. Next, we present how we perform secure binding between Rust and C/C++ (§4), formalize and prove the memory safety (§5), with implementation (§6), evaluation (§7), and applications (§8) of Rust-SGX, respectively. In §9, we discuss the limitations and future work, followed by related work in §10. Finally, §11 concludes the paper.

2 BACKGROUND

2.1 Intel SGX

Intel SGX is designed to provide applications the capability of executing code in a secure enclave while protecting secrets with their own execution environment [3, 24]. With SGX, application programmers can directly control the security of their applications without relying on any underlying system software such as the OS or hypervisor. Such a design significantly reduces the trusted computing base (TCB) to the smallest possible code (i.e., only the code executed inside the enclave is trusted), and prevents various software attacks even when the system software is compromised.

To use SGX, the applications typically need to be implemented with two components: a trusted component and an untrusted component. The trusted component is executed inside the enclave, whereas the untrusted component is executed outside. When data needs to be passed between trusted and untrusted components, it has to be copied from and to the enclave because enclave memory cannot be read directly outside of the enclave. Intel provides a mechanism to create bridge functions by using its corresponding SGX SDK. A bridge function at the enclave entry point dispatches calls to the corresponding functions inside the enclave. This allows an enclave to run only certain functions specified by the developers. These functions are called `ECALL`s, and they are called from the untrusted component. There are also corresponding functions that reside in the untrusted component called `OCALL`s, which are invoked inside the enclave to request services from the outside world.

2.2 The Rust Programming Language

Rust [23] is a systems programming language, allowing developers to have efficient implementation for systems software. It supports

both functional and imperative-procedural programming. Rust is designed to be memory safe with zero-cost abstraction. Its unique type system can statically guarantee the absence of uninitialized variables, dangling pointers, race conditions, and other memory safety bugs. Meanwhile, Rust does not rely on any automated garbage collection to reclaim memory resources, and instead most Rust objects have statically known lifetime and the rest are managed through smart pointers. Rust does not allow null pointers, and the data structures have to be acyclic.

An important concept in Rust to gain statically verifiable memory safety is the memory ownership [23]. The basic rules of memory ownership dictate that each memory object has a single owner variable at each program point. When the owner variable goes out of scope, the value is automatically deallocated. Other variables can temporarily "borrow" the object from its owner, but they have to return it before the owner reaches the end of its lifetime. An object can be borrowed unlimited number of times when it is in an immutable state, whereas mutable borrows are exclusive. In this way, Rust ensures that no memory objects are both mutable and aliased at the same time, eliminating memory errors.

However, Rust does provide a mechanism to allow programmers to write code that bypasses its memory safety checks. This is a necessary compromise when Rust needs to interact with other languages or manipulate the hardware directly, since the type safety of Rust is not applicable to code across these irregular programming boundary. An "`unsafe`" keyword is introduced to mark unchecked code for the convenience of additional security auditing.

3 OVERVIEW

3.1 Objectives, Threat model, and Scope

The key objective of Rust-SGX is to remove memory corruption vulnerabilities inside SGX enclave by enabling the enclave software development with memory safe languages. Not all memory safe languages (e.g., Python) are of our interest, and instead we have particular interest in the more efficient memory safe system languages. That is why we focus on Rust. However, we must ensure that our design and implementation will impose no significant performance overhead when enabling Rust with SGX enclave programming.

Rust-SGX shares the same threat model as does Intel SGX; namely, only the software running inside the enclave is trusted, and the rest (e.g., operating systems and hypervisors) is untrusted. The particular attacks Rust-SGX aims to defeat are those memory corruptions that exploit the insecure memory operations inside enclave programs. We do not provide any additional mechanisms to defeat various side channel attacks (e.g., page [36], cache [15]) against enclave programs, which is orthogonal to the memory safety problem we aim to solve.

In this work, we only focus on enabling application layer memory safety. The memory safety of the lower layer software such as the core SGX libraries (e.g., the cryptographic function implementations and SGX core service routines) are not within our scope. Note that Rust-SGX only provides enclave developers with the support for building memory safe SGX applications; however, it does not enforce strict adherence to all rules of memory safe programming. For example, application developers can still abuse the "`unsafe`" feature mentioned in §2.2 to produce Rust programs with exploitable memory bugs. To have a completely secure development process, the enclave code should be audited (manually or automatically with machine checks) before being deployed. Rust programs, by their nature, are significantly easier to audit than those written in memory unsafe languages, even with the presence of "`unsafe`" code blocks. Nevertheless, how to audit the enclave code is also not within the scope of this work.

3.2 Challenges

At a high level, there are two directions to build memory safe Rust-SGX. The first one is to build it from scratch without relying on anything, and the second is a layered approach in which Rust-SGX is built atop an existing full-fledged SDK (e.g., Intel SGX SDK).

To build Rust-SGX from scratch sounds very appealing. It unfortunately faces erroneous challenges. First, SGX enclave programs heavily involve cryptographic computations (abstracted as APIs), e.g., when communicating with outside enclave, the messages needed to be encrypted or signed. It is challenging to implement these cryptography APIs directly using Rust without using assembly code, especially for performance reasons. In general, cryptography algorithms get much better speedups from being written in assembly than most other programming languages. One notable example is the Intel's native AES-NI hardware implementation of the cipher, which can be directly invoked by the assembly code.

Second, now that Rust itself contains unsafe Rust and also Rust-SGX inevitably has to include assembly code for the efficient cryptographic API implementations, we will not be able to develop Rust-SGX purely with Rust. On the other hand, there is already full-fledged Intel SGX SDK available that contains not only performance efficient implementations for cryptography APIs, but also strong support of SGX functionality such as sealing and remote attestation. There is no need to re-implement them in another language if we can still achieve the same performance. Therefore, we decide to build Rust-SGX atop Intel SGX SDK. This also leads to the second challenge we must solve: namely how to guarantee programs developed with Rust-SGX still has efficient performance, compared to the enclave programs developed with traditional SGX SDKs.

Third, with the integration of Intel SGX SDK and also the original unsafe Rust, we must provide a safety guarantee in spite of these unsafe components. With a layered approach, a practical way is to ensure the interfaces between layers are secure. For instance, when providing APIs for Rust SGX programmers, we must ensure that the foreign function interface (FFI) is secure. That is, all the arguments and side effects of FFI function must be interposed and verified.

Finally, there are also a number of engineering challenges of implementing Rust-SGX, especially due to the discrepancies between the program execution model in enclave programs and traditional Rust programs. For instance, static data can be easily initialized in Rust, but there is no such mechanisms in SGX enclave. Also, unlike Rust thread, SGX thread does not have constructors nor destructors. In addition, Rust mutex is different compared to SGX pthread mutex, and we must properly implement Rust mutex with SGX enclaves.

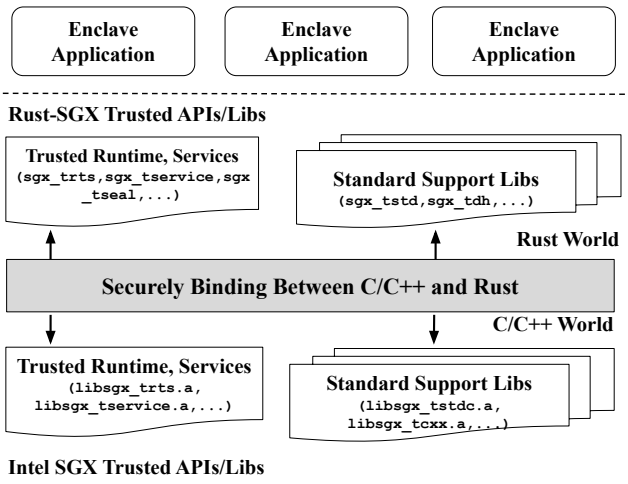

Figure 1: Overview of Rust-SGX.

3.3 Architecture

An overview of Rust-SGX is presented in Figure 1. There are three layers inside Rust-SGX: (1) the bottom layer that is the Intel SGX SDK (which is implemented in C/C++ and also assembly), (2) the middle layer that is the Rust and Foreign function Interfaces (FFI) for C/C++, and (3) the top layer that is the Rust SGX SDK. Rust-SGX provides memory safety by using Rust's memory safe properties, and meanwhile ensuring the interface of interacting with the lower Intel SGX SDK is memory safe.

Note that we cannot prove nor verify the memory safety of Intel SGX SDK due to its large code base and heavy use of C/C++, but if we can ensure the interaction between Rust world and C/C++ world is secure and prove the Rust world can not be escaped, then we can still ensure the memory safety of the Rust world. This is similar to the issues in which type safe language such as Java interacts with the Java native interface (JNI). However, there are still substantial differences, e.g., Rust and Java are totally independent from language aspect, and meanwhile SGX SDK itself has a variety of data types. We must build a proper mapping between the Rust world and SGX world. The details of how we achieve this mapping is presented in the next section.

4 SECURE BINDING BETWEEN RUST AND C/C++

As our layered design of Rust-SGX consists of code written in both C/C++ and Rust, it naturally raises the question of how we can securely bind the two worlds together. Since the memory safety of C/C++ and Rust relies on different security invariants, an arbitrary combination of the two worlds cannot be guaranteed to be memory safe. Therefore, we must add additional enforcement of how the two worlds interact with each other to assure that the two different language constructs are securely bound.

Note that previous research has investigated how to achieve memory safety in the context of the manner in which a safe language interacts with an unsafe one. In particular, Tan et al. [32] developed a scheme to safely call C functions inside a Java runtime environment via Java Native Interfaces (JNI). We face a similar situation in the design of Rust-SGX. Although the safety of Java and Rust is enforced via different mechanisms, the details are independent of the approach of Tan et al. Therefore, we have to modify and extend their techniques to securely binding the Rust language with the underlying Intel SGX SDK through carefully designed foreign function interfaces (FFI).

The core idea is to regulate the behavior of unsafe C/C++ code via a safe memory management scheme and an advanced type system (with certain run-time checks) that supports safe pointer operations. In general, such a memory management scheme and type system can restrict the expressiveness of the unsafe language and affect programmer's productivity. For Rust-SGX, however, this is much less of a concern, since we only need to safely wrap a fixed set of C/C++ APIs that are guaranteed to be stable by Intel. After a thorough review of Intel SGX SDK, we found that the semantics of the C/C++ APIs are very suitable to the aforementioned safety regulation. In the rest of this section, we show that it is feasible to design a memory safe interpolation scheme between the Rust language and Intel SGX SDK, with the safety properties formally proved.

It is worth mentioning that our purpose is not to provide a general memory safe Rust foreign function interface to C/C++, but to securely connect Rust with the C/C++ interface of Intel SGX SDK. Therefore, we are allowed to exclude a considerable proportion of C/C++ semantics from our consideration. Many problems that are difficult to address for a general solution can be solved through enumeration and a reasonable amount of manual checks, since the Intel SGX SDK only provides a finite set of C/C++ data structures and APIs that are not extensible by enclave developers.

4.1 Safe Memory Management

By manually auditing the semantics of all Intel SGX APIs, which are clearly documented by an Intel-provided manual, we found that these C/C++ functions are able to ensure high-degree of segregation between the heaps managed by the Rust world and the C/C++ world. In particular, the C/C++ APIs never free, or hold references to, the memory objects managed by callers. Therefore, calling Intel APIs preserves the safety invariant of the Rust heap without the need of additional regulation.

However, the Intel APIs do expose some objects allocated on their private heap to the outside Rust world. One example is `sgx_sha256_init`. This C/C++ API returns an allocated and initialized SHA context state, which can later be reclaimed by the Intel SDK through a pairing API, i.e., `sgx_sha256_close`. This opens a loophole that allows the Rust code to corrupt the C/C++ heap, which thus must be mitigated in the design of Rust-SGX.

There are two potential problems when accessing C/C++ heap via Rust code if the binding is inappropriately designed. The first is that the Rust code may be able to manipulate the internal states of these objects which are supposed to be hidden. We demonstrate that this concern can be resolved by introducing type safety enforcement, which will be explained in §4.2. The second problem is directly related to the memory management, i.e., the Rust code may continue to operate on a C/C++ heap object after it has been freed, leading to memory errors such as use after free or double free. To deal

```
extern {
    // Raw Intel SGX SDK APIs
    fn sgx_sha256_init(...);
    fn sgx_sha256_close(...);
}

pub struct SgxSha256 { handle: sgx_sha256_handle_t, ... }

impl SgxSha256 {
    ...
    // Initialization, call Intel SGX API to allocate and
    // object on the C/C++ heap and get a handle of it
    pub fn new() -> Self {
        ...
        unsafe { sgx_sha256_init(&mut handle, ...); }
        ...
    }
}

impl Drop for SgxSha256 {
    // When the object exits its scope, 'drop' is automatically
    // called to free the object via the pairing Intel SGX API
    pub fn drop(&mut self) {
        ...
        unsafe { sgx_sha256_close(&mut self.handle); }
        ...
    }
}
```

Figure 2: RAII-style C/C++ heap management in Rust

with these errors, we implement high-level wrappers for all C/C++ SGX data structures allocated on the heap. These wrappers provide the resource-acquisition-is-initialization (RAII) style of memory management. In this way, the task of releasing C/C++ heap objects is completely delegated to the lifetime semantics of the Rust language. The security and safety of the C/C++ heap is therefore guaranteed by the soundness of Rust's memory management scheme. Figure 2 displays an example of the conceptual design of RAII memory management in Rust. Users of type `SgxSha256` are only allowed to construct its object through the static `new` function. When the object reaches the end of its lifespan, the `drop` method is automatically called to securely release the internal reference to data on the C/C++ heap.

Note that we did not invent any automated tools to extract formalized semantics of the C/C++ APIs from the Intel reference. All inspections and reasoning are manually performed by experienced security researchers and professionals. Since there is only a finite number of Intel SGX APIs and they are reasonably stable, this manual effort is manageable and mostly a one-time cost.

4.2 Safe Memory Access of C/C++ Objects

The most important security property to enforce in our design of Rust-SGX is type safety. This problem is rooted in the mismatch between Rust and C/C++ types. For primitive types shared by both languages, there are well defined conversion semantics. For example, the Rust `std::libc` crate[1] provides the definitions of platform-dependent C/C++ numerical types, along with the routines to safely cast them from and to the corresponding Rust types. However, for complex types defined by the Intel SGX SDK, there are no built-in Rust representations for them. A naive solution is to refer to all complex C/C++ types with `* c_void` in Rust. Apparently, this abstraction is unsound and can easily cause type confusion errors.

To prevent conversions among incompatible C/C++ types in Rust, we introduce a *handle* type in Rust. For any C/C++ type τ, we define a corresponding Rust type $\texttt{Handle}_\tau$, which is essentially a pointer referring to a memory chunk which can be interpreted as a legal object of type τ in the C/C++ world. In the Rust world, $\texttt{Handle}_\tau$ can be used for testing equality and be used as function arguments. It also carries an `init` flag to indicate if the *handle* has been initialized. No other pointer operations are supported.

For any C/C++ defined type τ with SGX-specific semantics, there are only two ways to obtain $\texttt{Handle}_\tau$ objects in Rust. One is through the initialization from a `null` pointer. The other is to call the Intel SGX SDK API that initializes a Rust-allocated memory chunk as a τ object. In this way, we achieve type safety for memory accessed by both Rust and C/C++.

4.3 Safe Memory Access of Raw-Byte

Many Intel SGX APIs need to read or write to memory in a "type-less" manner. In other words, these APIs treat memory chunks as raw bytes, ignoring any semantics the memory chunk may carry. These APIs are mostly for cryptography tasks, which convert in-memory data structures to type-less byte arrays. Type-less memory is often with variable-length and its access could be out of bound.

To address this problem, we again referred to the Intel SGX SDK manual and found that Intel APIs always perform bounded type-less memory access by requiring callers to specify the length of the memory through additional parameters. Figure 3 shows such an API. In particular, `sgx_seal_data` is the function that encrypts a piece of type-less data with AES-GCM, using a key derived from secrets unique to the CPU chip. The function takes six parameters that can be divided into three pairs, each of which contains an unsigned integer (the `*_length` parameters) and a byte pointer (the `ptr_*` parameters). To safely use this API, the caller should ensure that, for each parameter pair, the integer value correctly bounds the length of the memory referred to by the pointer parameter. Abstractly, each of these parameter pair can be viewed as a fat pointer carrying bound information, which can be formalized by the `SEQ` (sequence) pointer defined by CCured [25]. We use the `Bytes` notation in Rust-SGX. Different from `SEQ` in CCured, `Bytes` is not a generic type constructor but a concrete type that only models arrays of bytes.

To regulate the usage of `Bytes` pointers, we define a trait `ContiguousMemory` to mark data structures with a contiguous memory layout. Types with this trait are allowed to derive a `Bytes` pointer, providing a type-less view of the object. Whoever implements `ContiguousMemory` is responsible for the safety of the conversion. We make `ContiguousMemory` an unsafe trait to prevent enclave developers of Rust-SGX from abusing `Bytes` pointers. Note that the `unsafe` keyword here is purely syntactical and does not necessarily indicate any actual lack of safety. Indeed, there is no

[1] A Rust crate is similar to a library in other languages.

```
sgx_status_t sgx_seal_data(
    // For read-only type-less memory access
    const uint32_t additional_MACtext_length,
    const uint8_t * ptr_additional_MACtext,
    // For read-only type-less memory access
    const uint32_t text2encrypt_length,
    const uint8_t * ptr_text2encrypt,
    // For writable type-less memory access
    const uint32_t sealed_data_size,
    sgx_sealed_data_t * ptr_sealed_data
);
```

Figure 3: Intel SGX API with Bounds Checks for Type-less Memory Access.

Table 1: Types Used by our Rust and C/C++ Interpolation.

Notation	Supported Operations
`Bytes`	pointer arithmetic; pass to C APIs; copy content to non-overlapping `Bytes`-described memory location
`ContiguousMemory`	convert to `Bytes`
`Sanitizable[T]`	construct `T` from `Bytes`
$\texttt{Handle}_\tau$	equality test; validity test

guarantee that enclave developers will securely implement this trait, but the unsafe marking makes the code easy to audit by security professionals. Similarly, we define an unsafe generic trait `Sanitizable[T]`, which provides a static function `from_bytes` that can safely construct a `T` object from a `Bytes` pointer. The implementation of `Sanitizable[T]` is obligated for sanitizing the raw byte array and ensuring that the memory represents a valid object of type `T`. A summary of the types used by Rust-SGX for securely wrapping of Intel SGX APIs inside the Rust language is presented in Table 1.

5 FORMALIZATION AND PROOF

In this section, we present a formal proof of the memory safety of our Intel SGX SDK binding in Rust-SGX, using a formal language $\mathcal{L}_R$ to model the subset of Rust that is relevant to our secure Intel SGX SDK binding scheme. Part of our formalization resembles the systems in CCured [25] and the work on safe Java Native Interfaces (JNI) by Tan el al. [32].

There has been a considerable amount of work trying to formally model and verify certain security properties of Rust programs or the language itself. To clarify the scope of our work, we emphasize that the to-be-presented formalization and proof aim to demonstrate the memory safety of the *design* of our foreign language binding. We do not intend to verify the security properties of our implementation or any "unsafe" Rust code inherited from the official Rust standard library code. The fulfillment of these objectives requires more powerful theories and significantly more human labor, which is out of the scope of this work.

$$
\begin{array}{rrcl}
\text{C Types} & \delta & ::= & \texttt{ctype}_1 \mid \texttt{ctype}_2 \mid \cdots \mid \texttt{ctype}_M \\
\text{Intel SGX APIs} & F & ::= & f_1 \mid f_2 \mid \cdots \mid f_N \\
\text{Type Constructors} & T & ::= & \texttt{Sanitizable} \mid \texttt{Ref} \\
\mathcal{L}_R \text{ Types} & \tau & ::= & \texttt{Bool} \mid \texttt{Int} \mid \texttt{SgxStatus} \\
 & & \mid & \texttt{ContiguousMemory} \\
 & & \mid & \texttt{Bytes} \mid \texttt{Handle}_\delta \\
 & & \mid & T[\tau] \\
 & & \mid & (\tau_1, \cdots, \tau_k) \rightarrow \tau_{k+1} \\
\text{Values} & v & ::= & n \mid \mathit{null} \mid \mathit{true} \mid \mathit{false} \\
 & & \mid & \mathit{sgx\_success} \mid \mathit{sgx\_error} \\
 & & \mid & F \\
 & & \mid & \mathit{ref}(n_{\text{addr}}) \mid \mathit{handle}(n_{\text{addr}}) \\
 & & \mid & \mathit{bytes}(n_{\text{addr}}, n_{\text{start}}, n_{\text{end}}) \\
 & & \mid & \mathit{failsafe} \\
\text{Expressions} & e & ::= & x \mid n \\
 & & \mid & e_1 + e_2 \\
 & & \mid & !e \mid e_1 = e_2 \\
 & & \mid & \&e \\
 & & \mid & \texttt{if\_init}\ e \\
\text{Statements} & s & ::= & e \\
 & & \mid & \textbf{let}\ x := s \\
 & & \mid & x := s \\
 & & \mid & \textbf{alloc\_bytes}\ e \\
 & & \mid & \textbf{to\_bytes}\ e \\
 & & \mid & \textbf{from\_bytes}[\tau]\ e \\
 & & \mid & F(e_1, \cdots, e_k) \\
 & & \mid & \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \\
 & & \mid & \textbf{while}\ e\ \textbf{do}\ s \\
 & & \mid & s_1; s_2
\end{array}
$$

Figure 4: Language Syntax of $\mathcal{L}_R$.

Figure 4 shows the syntax of our $\mathcal{L}_R$. For simplicity, $\mathcal{L}_R$ has no general function declarations and definitions. Instead, we treat functions important to our memory safety property as built-in statements. For example, the statement "**`alloc_bytes`** n" means allocating a byte array of length n. In real Rust programming, the statement is concretized as creating a variable of type `[u8;n]`. Common control flows like branch and loop are supported. Scoping is not modeled in $\mathcal{L}_R$, since we assume that all temporal safety properties are enforced by the Rust compiler.

There are also several special values defined in the language. In particular:

- Intel SGX APIs are modeled as callable built-in values, denoted by $f_1, \cdots, f_M$, where M is the total number of APIs used by Rust-SGX.
- *sgx_success* and *sgx_error* are to indicate whether a call to the corresponding Intel SGX API is successful or not.

- *ref*(n_{addr}) is the pointer type exclusively used by the parameters of Intel SGX APIs. Rust indeed supports reference types, but they are irrelevant in this formalization.
- *handle*(n_{addr}) denotes values of some $\texttt{Handle}_\delta$ type. Such a value has a single field: n_{addr} is the memory address of the underlying δ object. A hidden b_{init} field can be added to indicate whether the δ object has been correctly initialized.
- *bytes*($n_{\text{addr}}, n_{\text{start}}, n_{\text{end}}$) denotes a chunk of raw-byte memory with bound information.
- A singleton *failsafe* value used to indicate raising a run-time error. We introduce this value to simplify the semantics of the language. It may be implemented as an exception or it can simply indicate the termination of the program.

It is worth noting that the syntax of $\mathcal{L}_R$ contains notations of C/C++ types used to mark the corresponding *handle* types in $\mathcal{L}_R$. There are no semantics bound to these C/C++ type notations and the language does not allow explicit instantiation of the C/C++ objects. For convenience, we use $\mathcal{T}_C$ to denote the universe of C/C++ types appeared in Intel SGX APIs. The universe of valid $\mathcal{L}_R$ types is denoted by $\mathcal{T}_R$. A valid $\mathcal{L}_R$ type is always predicative.

Also, when enclave developers need to handle their own Rust types, they have to implement **to_bytes** and **from_bytes** in $\mathcal{L}_R$. However, in such cases, another security concern arises which is the correctness of data serialization and de-serialization. This problem is difficult to handle at a language level, since enclave developers are allowed to define arbitrarily complicated data structures and some of them may not be serializable by nature. To tackle this problem, we have ported the `serde` Rust library[2] into SGX. Note that `serde` is a toolkit for secure and automated data serialization and deserialization in Rust. It is not a part of Rust-SGX, but enclave developers are encouraged to use it if they need to pass custom data structures to the APIs provided by Intel SGX SDK.

5.1 Type System

Figure 5 defines the typing rules of $\mathcal{L}_R$. Some statements are considered to bear no type information, although there are still type constraints on their substatements. We use `Void` as the type of these statements when they are well formed. A phantom value of `Void` is () which indicates the end of computation.

We use $\sqcup$ and $\sqcap$ to denote type union and intersection, respectively. Formally, the two operators are defined as follows:

$$\Gamma \vdash e : \tau_1 \sqcup \tau_2 \iff \Gamma \vdash e : \tau_1 \vee \Gamma \vdash e : \tau_2$$
$$\Gamma \vdash e : \tau_1 \sqcap \tau_2 \iff \Gamma \vdash e : \tau_1 \wedge \Gamma \vdash e : \tau_2$$

Some of the expression typing rules need more explanation. As indicated by rule HandelNull, the *null* pointer can be viewed as any *handle* type. In this way, every *handle* object can be initialized as null. As aforementioned, the *failsafe* value is considered to be an object of the bottom type in the type lattice and thus can be viewed as an object of any valid $\mathcal{L}_R$ type.

SgxAPI is not a typical type inference rule but a meta constraint we impose on the type of Intel SGX APIs. A native SGX API can only take `Bytes`, `Int`, and reference to $\texttt{Handle}_\delta$ as parameters. Its return value must be an `SgxStatus` to notify the caller if the invocation fails. Although $\mathcal{L}_R$ has reference types, only `Handle` types can be taken reference. Moreover, no dereference operations are allowed in $\mathcal{L}_R$. The only way to modify memory through references are calling Intel SGX APIs with reference values as parameters. This is critical to the type safety of C/C++ memory.

[2] https://serde.rs/

5.2 Operational Semantics

Before defining the semantics of $\mathcal{L}_R$, we need some mathematical constructs to describe program state:

- $\mathcal{X}$ is the universe of all variables in a program.
- A program variable store $\Sigma : \mathcal{X} \to \mathbb{N}$ that maps variables to their memory locations.
- $\mathcal{V}$ is the universe of all possible values in a program.
- A memory configuration $\mathcal{M} : \mathbb{N} \to \mathcal{V}$ is a mapping from natural numbers to values. Note that $\mathcal{M}$ does not model the private memory of Intel SGX SDK.

Throughout program execution, the invariant $\text{rng}(\Sigma) \subseteq \text{dom}(\mathcal{M})$ holds, since every variable should be allocated in a chunk of memory.

We express the operational semantics in the forms of two judgments:

- expression evaluation: $\Sigma, \mathcal{M} \vdash e \Downarrow v$
- statement evaluation: $\Sigma, \mathcal{M} \vdash s \Downarrow s', \Sigma', \mathcal{M}'$

As indicated by the notations, evaluating an expression does not have side effects, while the computation of statements may modify the program state.

5.3 Type Safety

We now define the formal memory safety guarantee that $\mathcal{L}_R$ satisfies. Intuitively, the language $\mathcal{L}_R$ being memory safe means that the type environment Γ always honestly describes the program memory state, which is abstracted by the pair of $(\Sigma, \mathcal{M})$.

We introduce the type store concept to help reason the consistency between Γ and $(\Sigma, \mathcal{M})$. A memory type store $\Pi : \mathbb{N} \to \mathcal{T}$ maps memory locations to types, where $\mathcal{T} = \mathcal{T}_R \cup \mathcal{T}_C \cup \{\textbf{byte}\}$. **byte** is the meta type describing type-less memory. It should not be confused with `Bytes` which is a valid $\mathcal{L}_R$ type serving as the meta data of an allocated chunk of type-less memory.

We then define for each type τ a set of valid values $[\![\tau]\!]_\Pi$. Since $\mathcal{L}_R$ has pointer types, the set of valid values of τ depends on Π, as shown in Figure 6. Note that we do not define $[\![\delta]\!]_\Pi$ when $\delta \in \mathcal{T}_C$, because it is agnostic to $\mathcal{L}_R$. The fundamental reason is that the implementation of Intel SGX SDK is transparent to Rust-SGX. Nevertheless, by assuming that Intel SGX SDK is securely implemented, we do not require this information to formalize or prove our type safety.

We can now formally define the type safety for program states.

Definition 5.1 (Type-safe program state). The notation

$$\Sigma, \mathcal{M} \models_\Pi \Gamma$$

indicates that the program state $(\Sigma, \mathcal{M})$ is safe in type environment Γ, with respect to the type store Π:

$$\begin{aligned} \Sigma, \mathcal{M} \models_\Pi \Gamma \quad \stackrel{\text{def}}{=\!=} \quad & \text{dom}(\mathcal{M}) = \text{dom}(\Pi) \\ \wedge \quad & \forall x \in \text{dom}(\Sigma).\Gamma(x) = \Pi(\Sigma(x)) \\ \wedge \quad & \forall n \in \text{dom}(\mathcal{M}).\mathcal{M}(n) \in [\![\Pi(n)]\!]_\Pi \end{aligned}$$

Values:

$$\textsc{Int}\ \frac{}{\vdash n : \texttt{Int}} \qquad \textsc{True}\ \frac{}{\vdash \mathit{true} : \texttt{Bool}} \qquad \textsc{False}\ \frac{}{\vdash \mathit{false} : \texttt{Bool}}$$

$$\textsc{SgxSuccess}\ \frac{}{\vdash \mathit{sgx\_success} : \texttt{SgxStatus}} \qquad \textsc{SgxError}\ \frac{}{\vdash \mathit{sgx\_error} : \texttt{SgxStatus}}$$

$$\textsc{HandleNull}\ \frac{}{\vdash \mathit{null} \equiv \mathit{handle}(0) : \bigsqcap_{\delta \in \mathcal{T}_C} \texttt{Handle}_\delta} \qquad \textsc{HandleNonNull}\ \frac{n \neq 0}{\vdash \mathit{handle}(n) : \bigsqcup_{\delta \in \mathcal{T}_C} \texttt{Handle}_\delta}$$

$$\textsc{Bytes}\ \frac{}{\vdash \mathit{bytes}(n_{\text{addr}}, n_{\text{start}}, n_{\text{end}}) : \texttt{Bytes}} \qquad \textsc{Failsafe}\ \frac{}{\vdash \mathit{failsafe} : \bigsqcap \mathcal{T}_R}$$

$$\textsc{SgxAPI}\ \frac{f_i : (\tau_1, \cdots, \tau_k) \to \tau_{k+1}}{\tau_{k+1} = \texttt{SgxStatus} \wedge \forall i \in \{1, \cdots, k\}.\tau_i = \texttt{Bytes} \vee \tau_i = \texttt{Int} \vee \tau_i \in \{\texttt{Ref}[\texttt{Handle}_\delta] : \tau \in \mathcal{T}_C\}}$$

Expressions:

$$\textsc{IntAdd}\ \frac{\Gamma \vdash e_1 : \texttt{Int} \quad \Gamma \vdash e_2 : \texttt{Int}}{\Gamma \vdash e_1 + e_2 : \texttt{Int}} \qquad \textsc{BytesArith}\ \frac{\Gamma \vdash e_1 : \texttt{Bytes} \quad \Gamma \vdash e_2 : \texttt{Int}}{\Gamma \vdash e_1 + e_2 : \texttt{Bytes}}$$

$$\textsc{Neg}\ \frac{\Gamma \vdash e : \texttt{Bool}}{\Gamma \vdash !e : \texttt{Bool}} \qquad \textsc{Eq}\ \frac{\tau \in \{\texttt{Int}, \texttt{Bytes}\} \cup \{\texttt{Handle}_\delta : \delta \in \mathcal{T}_C\} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 = e_2 : \texttt{Bool}}$$

$$\textsc{Ref}\ \frac{\Gamma \vdash x : \texttt{Handle}_\delta}{\Gamma \vdash \&x : \texttt{Ref}[\texttt{Handle}_\delta]} \qquad \textsc{TestHandleInit}\ \frac{\Gamma \vdash e : \texttt{Handle}_\delta}{\Gamma \vdash \texttt{if\_init}\ \ e : \texttt{Bool}}$$

Statements:

$$\textsc{DeclarVar}\ \frac{\Gamma \vdash x : \tau \quad \Gamma \vdash s : \tau}{\Gamma \vdash \textbf{let}\ \ x := s : \texttt{Void}} \qquad \textsc{SetVar}\ \frac{\Gamma \vdash x : \tau \quad \Gamma \vdash s : \tau}{\Gamma \vdash x := s : \texttt{Void}}$$

$$\textsc{CallSgxAPI}\ \frac{f_i : (\tau_1, \cdots, \tau_k) \to \tau_{k+1} \quad \Gamma \vdash e_1 : \tau_1 \quad \cdots \quad \Gamma \vdash e_k : \tau_k}{\Gamma \vdash f_i(e_1, \cdots, e_k) : \texttt{SgxStatus}}$$

$$\textsc{AllocBytes}\ \frac{\Gamma \vdash e : \texttt{Int}}{\Gamma \vdash \textbf{alloc\_bytes}\ \ e : \texttt{Bytes}}$$

$$\textsc{ToBytes}\ \frac{\Gamma \vdash e : \tau \quad \tau \preceq \texttt{ContinugousMemory}}{\Gamma \vdash \textbf{to\_bytes}\ \ e : \texttt{Bytes}} \qquad \textsc{FromBytes}\ \frac{\Gamma \vdash e : \texttt{Bytes} \quad \tau \preceq \texttt{Sanitizable}[\tau]}{\Gamma \vdash \textbf{from\_bytes}[\tau]\ \ e : \tau}$$

$$\textsc{Seq}\ \frac{\Gamma \vdash s_1 : \tau_1 \quad \Gamma \vdash s_2 : \tau_2}{\Gamma \vdash s_1; s_2 : \tau_2} \qquad \textsc{Branch}\ \frac{\Gamma \vdash e : \texttt{Bool} \quad \Gamma \vdash s_1 : \tau_1 \quad \Gamma \vdash s_2 : \tau_2}{\Gamma \vdash \textbf{if}\ \ e\ \ \textbf{then}\ \ s_1\ \ \textbf{else}\ \ s_2 : \texttt{Void}} \qquad \textsc{Loop}\ \frac{\Gamma \vdash e : \texttt{Bool} \quad \Gamma \vdash s : \tau}{\Gamma \vdash \textbf{while}\ \ e\ \ \textbf{do}\ \ s : \texttt{Void}}$$

Notations:

$\mathcal{T}_R$: Universe of valid $\mathcal{L}_R$ types $\mathcal{T}_C$: Universe of C types $\preceq$: Subtype relation $\sqcup$: Type union $\sqcap$: Type intersection

Figure 5: Typing Rules of $\mathcal{L}_R$.

For a program to be type safe, its program state should be type safe at every program point. Thus, the whole-program type safety can be established via the following two theorems.

Theorem 5.2 (Type safety for program expressions). *Given an initial program state* $(\Sigma, \mathcal{M})$*, and its type environment* Γ*, if*

$$\Gamma \vdash e : \tau$$

and there exists a type store Π *such that*

$$\Sigma, \mathcal{M} \models_\Pi \Gamma$$

then one of the following cases is true:

(1) $\Sigma, \mathcal{M} \vdash e \Downarrow \mathit{failsafe}$
(2) $\exists v.\Sigma, \mathcal{M} \vdash e \Downarrow v$ *and* $v \in [\![\tau]\!]_\Pi$

Theorem 5.3 (Type safety for program statements). *Given an initial program state* $(\Sigma, \mathcal{M})$*, and its type environment* Γ*, if*

$$\Gamma \vdash s : \tau$$

and there exists a type store Π *such that*

$$\Sigma, \mathcal{M} \models_\Pi \Gamma$$

$$
\begin{array}{lcl}
[\![\textbf{byte}]\!]_\Pi & = & \{0_{\text{8-bit}}, \cdots, 255_{\text{8-bit}}\} \\
[\![\texttt{Bool}]\!]_\Pi & = & \{\mathit{true}, \mathit{false}\} \\
[\![\texttt{Int}]\!]_\Pi & = & \mathbb{N} \\
[\![\texttt{SgxStatus}]\!]_\Pi & = & \{\mathit{sgx\_success}, \mathit{sgx\_error}\} \\
[\![\texttt{Handle}_\delta]\!]_\Pi & = & \{\mathit{handle}(n) : n \in \mathrm{dom}(\Pi) \wedge \Pi(n) = \delta \in \mathcal{T}_C\} \\
[\![\texttt{Bytes}]\!]_\Pi & = & \{\mathit{bytes}(n, b, e) : b \leq n < e \wedge \\
 & & \qquad [b, e) \subseteq \mathrm{dom}(\Pi) \wedge \\
 & & \qquad \forall i \in [b, e).\Pi(i) = \textbf{byte}\} \\
[\![\texttt{Ref}[\tau]]\!]_\Pi & = & \{\mathit{ref}(n) : n \in \mathrm{dom}(\Pi) \wedge \Pi(n) = \tau\}
\end{array}
$$

Figure 6: Valid values of each type within type store Π

then one of the following cases is true:

(1) $\Sigma, \mathcal{M} \vdash s \Downarrow \mathit{failsafe}, *, *$

(2) $\exists v, \Sigma', \mathcal{M}', \Pi'.$
$\Pi \subseteq \Pi' \wedge \Sigma, \mathcal{M} \vdash s \Downarrow v, \Sigma, \mathcal{M}' \wedge (v = () \vee v \in [\![\tau]\!]_{\Pi'}) \wedge$
$\Sigma', \mathcal{M}' \models_{\Pi'} \Gamma$

For the simplicity of notations, we define two invariant judgments

$$\forall \tau \in \mathcal{T}_R. \forall v \in [\![\tau]\!]_\Pi. \Sigma, \mathcal{M} \vdash v \Downarrow v$$

in the context of expression evaluation. Similarly, we have

$$\forall \tau \in \mathcal{T}_R. \forall v \in [\![\tau]\!]_\Pi \cup \{()\}. \Sigma, \mathcal{M} \vdash v \Downarrow v, \Sigma, \mathcal{M}$$

when considering statement evaluation.

The two theorems indicate the progress and preservation properties of our type system. That is, a well-typed program never gets stuck during computation, and the computation preserves the types. Proofs of these two theorems are through induction on the structure of expressions and statements in $\mathcal{L}_R$. Note that the formalization of Theorem 5.3 assumes *all computation will terminate*. Although this in general is not true since $\mathcal{L}_R$ supports the while loop, the formalization can be easily extended to include the non-terminating situation.

5.4 Operational Semantics of $\mathcal{L}_R$

Figure 7 displays part of the operational semantics of $\mathcal{L}_R$. For conventional program semantics like integer and boolean operations, their formalization is mostly trivial and thus omitted in the paper. Also, the formalization in Figure 7 mixed notations of syntactical elements and symbols with operational effects. This is for simplicity and clarity of understanding.

As stated in the main text, some safety properties of our type system are dynamically checked; these are marked with blue frames in Figure 7. if any of these checks fail, the evaluation generates the *failsafe* value. We *do not* consider *failsafe* as a blocker of the computation. Also, Figure 7 does not include some of the *failsafe*-related semantic rules. The basic principle is that, if any of the sub-expressions or sub-statements is evaluated to *failsafe*, the entire expression or statement will be evaluated to *failsafe*. In other words, the imaginary value *failseafe* propagates to the next step of the computation indefinitely

To ensure type safety, the statements **to_bytes** and **from_bytes** always allocate new memory for the results instead of modifying the input object content in place. In the implementation, this can sometimes can be optimized for better performance.

The semantics of calling Intel SGX APIs is more complicated than others. As previously mentioned, an Intel SGX API only takes three types of parameters, i.e., integers, type-less arrays with bound checks, and handles to objects of private C types. We assume the implementation of Intel SGX APIs preserves type safety. Part of this assumption is formalized as predicates in red frames.

We do not list the semantics of `if_init` e expressions in Figure 7 because the implementation of this primitive is dependent on the type of the handle. In general, this primitive can be implemented by bundling each *handle* instance with a companion variable indicating whether the data structure has been properly initialized by a particular Intel SGX SDK API. Nevertheless, the initialization tests for *handle* objects are mostly irrelevant to memory safety. Instead, it is included by $\mathcal{L}_R$ to ensure certain temporal safety of stateful data structure operations.

5.5 Soundness Proof Sketch

5.5.1 Type Safety for Expressions. We prove Theorem 5.2 by inducting on the structure of e. In this proof sketch, we only prove the $e = e_1 + e_2$ case to demonstrate the idea. The other cases are either trivial or can be similarly proved, thus omitted.

Lemma 5.4 (Type safety for the plus expression). *Given an initial program state* $(\Sigma, \mathcal{M})$*, and its type environment* Γ*, if there exists a type store* Π *such that*

$$\Gamma \vdash e_1 + e_2 : \tau \wedge \Sigma, \mathcal{M} \models_\Pi \Gamma$$

then one of the following cases is true,

(1) $\Sigma, \mathcal{M} \vdash e_1 + e_2 \Downarrow \mathit{failsafe}$
(2) $\exists v. \Sigma, \mathcal{M} \vdash e_1 + e_2 \Downarrow v$ *and* $v \in [\![\tau]\!]_\Pi$

Proof. Assume there exists a type store Π such that $\Sigma, \mathcal{M} \models_\Pi \Gamma$.

Given $e_1 + e_2$, either typing rule IntAdd or typing rule BytesArith applies. The proof is thus split into two cases.

In case that typing rule IntAdd *applies,* we have $\Gamma \vdash e_1 : \texttt{Int}$. Since $\Sigma, \mathcal{M} \models_\Pi \Gamma$, by assumption it is true that for e_1, one of the following cases is true,

(1) $\Sigma, \mathcal{M} \vdash e_1 \Downarrow \mathit{failsafe}$
(2) $\exists v. \Sigma, \mathcal{M} \vdash e \Downarrow v \in [\![\texttt{Int}]\!]_\Pi$

A similar conclusion holds for e_2 as well. If

$$\Sigma, \mathcal{M} \vdash e_1 \Downarrow \mathit{failsafe} \vee \Sigma, \mathcal{M} \vdash e_2 \Downarrow \mathit{failsafe}$$

then we have

$$\Sigma, \mathcal{M} \vdash e_1 + e_2 \Downarrow \mathit{failsafe}$$

which is the first case of our proof obligation.

If both e_1 and e_2 can be successfully evaluated, then for some $n_1, n_2 \in [\![\texttt{Int}]\!]_\Pi$ such that $\Sigma, \mathcal{M} \vdash e_1 \Downarrow n_1$ and $\Sigma, \mathcal{M} \vdash e_2 \Downarrow n_2$. At this point, the semantics rule *IntAdd* applies. Therefore, $\Sigma, \mathcal{M} \vdash e_1 + e_2 \Downarrow \underline{n_1 + n_2}$ where $\underline{n_1 + n_2}$ is the result of adding integers n_1 and n_2. By definition, $\underline{n_1 + n_2} \in [\![\texttt{Int}]\!]_\Pi$

In case that typing rule BytesPtrArith *applies,* we have

$$\Gamma \vdash e_1 : \texttt{Bytes} \wedge \Gamma \vdash e_2 : \texttt{Int}$$

By assumption, e_1 and e_2 are type safe. Similar to the first case, if either e_1 or e_2 is evaluated to *fialsafe*, then $e_1 + e_2$ will be evaluated

Expressions:

$$\text{Var}\ \frac{\Sigma(x) = n \quad \mathcal{M}(n) = v}{\Sigma, \mathcal{M} \vdash x \Downarrow v} \qquad \text{Ref}\ \frac{\Sigma(x) = n}{\Sigma, \mathcal{M} \vdash \&x \Downarrow ref(n)}$$

$$\text{IntAdd}\ \frac{\Sigma, \mathcal{M} \vdash e_1 \Downarrow n_1 \quad \Sigma, \mathcal{M} \vdash e_2 \Downarrow n_2}{\Sigma, \mathcal{M} \vdash e_1 + e_2 \Downarrow \underline{n_1 + n_2}}$$

$$\text{BytesPtrArith}\ \frac{\Sigma, \mathcal{M} \vdash e_1 \Downarrow bytes(n_1, b, e) \quad \Sigma, \mathcal{M} \vdash e_2 \Downarrow n_2 \quad \boxed{b \le n_1 + n_2 < e}}{\Sigma, \mathcal{M} \vdash e_1 + e_2 \Downarrow bytes(n_1 + n_2, b, e)}$$

Statements:

$$\text{SetVar}\ \frac{\Sigma(x) = n \quad \Sigma, \mathcal{M} \vdash s \Downarrow v, \Sigma', \mathcal{M}'}{\Sigma, \mathcal{M} \vdash x := s \Downarrow (), \Sigma', \mathcal{M}'[n \mapsto v]} \qquad \text{Seq}\ \frac{\Sigma, \mathcal{M} \vdash s_1 \Downarrow *, \Sigma', \mathcal{M}'}{\Sigma, \mathcal{M} \vdash s_1; s_2 \Downarrow s_2, \Sigma', \mathcal{M}'}$$

$$\text{IfTrue}\ \frac{\Sigma, \mathcal{M} \vdash e \Downarrow true}{\Sigma, \mathcal{M} \vdash \textbf{if}\ \ e\ \ \textbf{then}\ \ s_1\ \ \textbf{else}\ \ s_2 \Downarrow s_1, \Sigma, \mathcal{M}} \qquad \text{IfFalse}\ \frac{\Sigma, \mathcal{M} \vdash e \Downarrow false}{\Sigma, \mathcal{M} \vdash \textbf{if}\ \ e\ \ \textbf{then}\ \ s_1\ \ \textbf{else}\ \ s_2 \Downarrow s_2, \Sigma, \mathcal{M}}$$

$$\text{WhileTrue}\ \frac{\Sigma, \mathcal{M} \vdash e \Downarrow true}{\Sigma, \mathcal{M} \vdash \textbf{while}\ \ e\ \ \textbf{do}\ \ s \Downarrow s; \textbf{while}\ \ e\ \ \textbf{do}\ \ s, \Sigma, \mathcal{M}} \qquad \text{WhileFalse}\ \frac{\Sigma, \mathcal{M} \vdash e \Downarrow false}{\Sigma, \mathcal{M} \vdash \textbf{while}\ \ e\ \ \textbf{do}\ \ s \Downarrow (), \Sigma, \mathcal{M}}$$

$$\text{AllocBytes}\ \frac{\Sigma, \mathcal{M} \vdash e \Downarrow n_{\text{length}} \quad \boxed{n_{\text{length}} > 0} \quad \boxed{\exists n_{\text{start}}.[n_{\text{start}}, n_{\text{start}} + n_{\text{length}}) \cap \text{dom}(\mathcal{M}) = \varnothing}}{\Sigma, \mathcal{M} \vdash \textbf{alloc\_bytes}\ \ e \Downarrow bytes(n_{\text{start}}, n_{\text{start}}, n_{\text{start}} + n_{\text{length}} - 1), \Sigma, \mathcal{M}[n_{\text{start}} \mapsto 0_{8-\text{bit}}, \cdots, n_{\text{start}} + n_{\text{length}} - 1 \mapsto 0_{8-\text{bit}}]}$$

$$\text{ToBytes}\ \frac{\Sigma, \mathcal{M} \vdash e \Downarrow v \quad n_{\text{length}} = \text{sizeof}(v) \quad \Sigma, \mathcal{M} \vdash \textbf{alloc\_bytes}\ \ n_{\text{length}} \Downarrow bytes(n_{\text{start}}, n_{\text{start}}, n_{\text{start}} + n_{\text{length}}), \Sigma, \mathcal{M}'}{\Sigma, \mathcal{M} \vdash \textbf{to\_bytes}\ \ e \Downarrow bytes(n_{\text{start}}, n_{\text{start}}, n_{\text{start}} + n_{\text{length}}), \Sigma, \mathcal{M}[n_{\text{start}}, \cdots, n_{\text{start}} + n_{\text{length}} - 1 \mapsto (\!|v|\!)_{[\textbf{byte};\text{sizeof}(v)]}]}$$

$$\text{FromBytes}\ \frac{\begin{array}{c}\Sigma, \mathcal{M} \vdash e \Downarrow bytes(n_{\text{start}}, n_{\text{begin}}, n_{\text{end}}) \\ (\!|\mathcal{M}[n_{\text{start}}], \mathcal{M}(n_{\text{start}} + 1), \cdots, \mathcal{M}(n_{\text{end}} - 1)|\!)_\tau \in [\![\tau]\!]_\Pi \\ \Sigma, \mathcal{M} \vdash \textbf{alloc\_bytes}\ \ (n_{\text{end}} - n_{\text{start}}) \Downarrow bytes(n'_{start}, n'_{start}, n'_{end}), \Sigma, \mathcal{M}'\end{array}}{\Sigma, \mathcal{M} \vdash \textbf{from\_bytes}[\tau]\ \ e \Downarrow \Sigma, \mathcal{M}[n'_{\text{start}} \mapsto (\!|\mathcal{M}(n_{\text{start}}), \cdots, \mathcal{M}(n_{\text{end}} - 1)|\!)_\tau]}$$

$$\text{CallSgxAPI}\ \frac{\begin{array}{c}E_{\text{int}} = \{e_1^{\texttt{Int}}, \cdots, e_h^{\texttt{Int}}\} \quad E_{\text{ref-handle}} = \{e_1^{\texttt{Ref}}, \cdots, e_i^{\texttt{Ref}}\} \quad E_{\text{bytes}} = \{e_1^{\texttt{Bytes}}, \cdots, e_j^{\texttt{Bytes}}\} \\ h + i + j = k \\ E_{\text{int}} \cup E_{\text{ref-handle}} \cup E_{\text{bytes}} = \{e_1, \cdots, e_k\} \\ \begin{array}{ccc}\Sigma, \mathcal{M} \vdash e_1^{\texttt{Int}} \Downarrow n_1 & \cdots & \Sigma, \mathcal{M} \vdash e_h^{\texttt{Int}} \Downarrow n_h \\ \Sigma, \mathcal{M} \vdash e_1^{\texttt{Ref}} \Downarrow ref(m_1) & \cdots & \Sigma, \mathcal{M} \vdash e_i^{\texttt{Ref}} \Downarrow ref(m_i) \\ \Sigma, \mathcal{M} \vdash e_1^{\texttt{Bytes}} \Downarrow bytes(a_1, b_1, c_1) & \cdots & \Sigma, \mathcal{M} \vdash e_j^{\texttt{Bytes}} \Downarrow bytes(a_j, b_j, c_j)\end{array} \\ \boxed{\text{Any two of } bytes(a_1, b_1, c_1), \cdots, bytes(a_j, b_j, c_j) \text{ do not overlap}} \\ \boxed{\text{There exists some } \mathcal{M}' \text{ s.t. } \text{dom}(\mathcal{M}) \subseteq \text{dom}(\mathcal{M}')} \\ \boxed{\mathcal{M} \text{ and } \mathcal{M}' \text{ only differ at } \{m_1, \cdots, m_i\} \cup \{n : \mathcal{M}'(n) \text{ is a C value}\} \cup \{n : \exists t.n \in [b_t, c_t)\}}\end{array}}{\Sigma, \mathcal{M} \vdash f_x(e_1, \cdots, e_k) \Downarrow sgx\_\{success, error\}, \Sigma, \mathcal{M}'}$$

Propositions with blue frames *are dynamically checked. A failsafe value is generated when these checks fail.*

Propositions with red frames *are assumptions about Intel SGX API semantics.*

The notation $(\!|v|\!)_{[\textbf{byte};n]}$ *means interpreting the value* v *in memory as a sequence of type-less bytes of length* n.
The notation $(\!|b_1, b_2, \cdots, b_n|\!)_\tau$ *means interpreting the byte sequence* $b_1, b_2, \cdots, b_n$ *in memory as a value of type* τ.

Figure 7: Operational semantics.

to *failsafe*. It is now sufficient to prove that if there exists $v \in [\![\texttt{Bytes}]\!]_\Pi$ and $m \in [\![\texttt{Int}]\!]_\Pi$ such that

$$\Sigma, \mathcal{M} \vdash e_1 \Downarrow v \wedge \Sigma, \mathcal{M} \vdash e_2 \Downarrow m$$

Let $v_1 = bytes(n, b, e) \in [\![\texttt{Bytes}]\!]_\Pi$. At this point, the dynamic check $b \leq n + m < e$ will be commenced. If the check fails, $e_1 + e_2$ is evaluated to *failsafe*. Otherwise, the semantics rule *BytesPtrArith* applies. Per this rule,

$$\Sigma, \mathcal{M} \vdash e_1 + e_2 \Downarrow bytes(n + m, b, e)$$

It is easy to verify that $bytes(n + m, b, e) \in [\![\texttt{Bytes}]\!]_\Pi$. □

5.5.2 Type Safety for Statements. Similar to Theorem 5.2, Theorem 5.3, is also proved by inducting on the structure of statements s. Since all expressions are statements free of side effects, an immediate corollary of Theorem 5.2 is that,

Corollary 5.5. *Given an initial program state* $(\Sigma, \mathcal{M})$*, and its type environment* Γ*, if there exists a type store* Π *such that*

$$\Gamma \vdash e : \tau \wedge \Sigma, \mathcal{M} \models_\Pi \Gamma$$

then one of the following cases is true,

(1) $\exists \Sigma', \mathcal{M}'. \Sigma, \mathcal{M} \vdash e \Downarrow failsafe, \Sigma, \mathcal{M}$
(2) $\exists v. \Sigma, \mathcal{M} \vdash e \Downarrow v, \Sigma, \mathcal{M} \wedge v \in [\![\tau]\!]_\Pi \wedge \Sigma, \mathcal{M} \models_\Pi \Gamma$

For non-expression statements, there are three different cases:

- Statements assumed to be type safe. The only instance of this kind is the Intel SGX API invocation statement.
- Compound statements that are evaluated to smaller statements, without considering the potentially non-terminating while loop.
- Simple statements that can be immediately evaluated to a value.

For the second kind of statements, since the structure of statements is well founded, recursively proving the type safety of sub-statements eventually leads to the proof for the whole statement. In other words, in order to prove Theorem 5.3, it is sufficient to prove the following lemma

Lemma 5.6 (Recursive type safety for program statements). *Given an initial program state* $(\Sigma, \mathcal{M})$*, and its type environment* Γ*, if*

$$\Gamma \vdash s : \tau$$

and there exists a type store Π *such that*

$$\Sigma, \mathcal{M} \models_\Pi \Gamma$$

then one of the following cases is true,

(1) $\Sigma, \mathcal{M} \vdash s \Downarrow failsafe, *, *$
(2) $\exists v, \Sigma', \mathcal{M}', \Pi'.$
$\Pi \subseteq \Pi' \wedge \Sigma, \mathcal{M} \vdash s \Downarrow v, \Sigma, \mathcal{M}' \wedge (v = () \vee v \in [\![\tau]\!]_{\Pi'}) \wedge \Sigma', \mathcal{M}' \models_{\Pi'} \Gamma$
(3) There exists s' *which is a sub-statement of* s *such that*

$$\Gamma \vdash s' : \tau' \wedge \Sigma, \mathcal{M} \models s \Downarrow s', \Sigma', \mathcal{M}'$$

and there eixsts Π' *such that*

$$\Pi \subseteq \Pi' \wedge \Sigma', \mathcal{M}' \models_{\Pi'} \Gamma$$

For the third kind of statements, we pick the case of **alloc_bytes** statements as a demonstration of the general strategy of proving statement type safety. According to the typing rule AllocBytes, the type of **alloc_bytes** statements has to be `Bytes`.

Lemma 5.7 (Type safety for the bytes allocation statement). *Given an initial program state* $(\Sigma, \mathcal{M})$*, and its type environment* Γ*, if there exists a type store* Π *such that*

$$\Gamma \vdash \textbf{alloc\_bytes}\ \ e : \texttt{Bytes} \wedge \Sigma, \mathcal{M} \models_\Pi \Gamma$$

then one of the following cases is true,

(1) $\Sigma, \mathcal{M} \vdash \textbf{alloc\_bytes}\ \ e \Downarrow failsafe$
(2) $\exists v, \Sigma', \mathcal{M}', \Pi'.$
$\Pi \subseteq \Pi' \wedge \Sigma', \mathcal{M}' \vdash \textbf{alloc\_bytes}\ \ e \Downarrow v \wedge v \in [\![\texttt{Bytes}]\!]_\Pi$

Proof. Per typing rule AllocBytes, $\Gamma \vdash e : \texttt{Int}$. Since $\Sigma, \mathcal{M} \models_\Pi \Gamma$, by the induction assumption, either $\Sigma, \mathcal{M} \models e \Downarrow failsafe$ or $\Sigma, \mathcal{M} \models e \Downarrow n \in [\![\texttt{Int}]\!]_\Pi$. If e derives *failsafe*, due to *failsafe* propagation, $\Sigma, \mathcal{M} \vdash \textbf{alloc\_bytes}\ \ e \Downarrow failsafe, \Sigma', \mathcal{M}'$.

If otherwise $\Sigma, \mathcal{M} \vdash e \Downarrow n \in [\![\texttt{Int}]\!]_\Pi$, then the system dynamically checks if n is positive. If not, the evaluation again leads to a safe failure. Next, the heap allocator finds a free space in the memory $\mathcal{M}$ that is large enough to hold n contiguous bytes. If the heap is full, the evaluation also fails safely. After all dynamic checks succeed, the semantics rule AllocBytes applies.

$$\Sigma, \mathcal{M} \vdash \textbf{alloc\_bytes}\ \ e \Downarrow bytes(n_{\text{start}}, n_{\text{start}}, n_{\text{start}} + n), \Sigma, \mathcal{M}'$$

where n_{start} is the starting address of the newly allocated type-less memory bytes and

$$\mathcal{M}' = \mathcal{M}[n_{\text{start}} \mapsto 0_{8-\text{bit}}, \cdots, n_{\text{start}} + n - 1 \mapsto 0_{8-\text{bit}}]$$

Since $0_{8-\text{bit}}$ is a valid **byte** value with respect to any type store, $bytes(n_{\text{start}}, n_{\text{start}}, n_{\text{start}} + n) \in [\![\texttt{Bytes}]\!]_\Pi$.

Let $\Pi' = \Pi[n_{\text{start}} \mapsto \textbf{byte}, \cdots, n_{\text{start}} + n - 1 \mapsto \textbf{byte}]$. By assumptions on the correctness of heap memory allocation,

$$[n_{\text{start}}, n_{\text{start}} + n) \cap \text{dom}(\mathcal{M}) = \varnothing$$

Therefore, $\Pi \subset \Pi'$ and

$$bytes(n_{\text{start}}, n_{\text{start}}, n_{\text{start}} + n) \in [\![\texttt{Bytes}]\!]_{\Pi'}$$

By the definitions of $\mathcal{M}'$ and Π', $\text{dom}(\Pi') = \text{dom}(\mathcal{M}')$. We now divide the domain of $\mathcal{M}'$ into two disjoint parts, i.e., $\text{dom}(\mathcal{M}') = \text{dom}(\mathcal{M}) \cup [n_{\text{start}}, n_{\text{start}} + n)$. For the first part, we already know

$$\Sigma, \mathcal{M} \models_\Pi \Gamma$$

Since $\Pi \subset \Pi'$, the following is true

$$\Sigma, \mathcal{M}'\big|_{\text{dom}(\mathcal{M})} \models_{\Pi'} \Gamma$$

For the second part, again by the definitions of $\mathcal{M}'$ and Π', we have

$$\text{rng}(\mathcal{M}'\big|_{[n_{\text{start}}, n_{\text{start}}+n)}) = \{0_{8-\text{bit}}\} \wedge \text{rng}(\Pi'\big|_{[n_{\text{start}}, n_{\text{start}}+n)}) = \{\textbf{byte}\}$$

Thus,

$$\Sigma, \mathcal{M}'\big|_{[n_{\text{start}}, n_{\text{start}}+n)} \models_{\Pi'} \Gamma$$

By unifying the type safety of the two disjoint parts of memory layout $\mathcal{M}'$, we prove that

$$\Sigma, \mathcal{M}' \models_{\Pi'} \Gamma$$

□

Table 2: Summary of LOC in Current Rust-SGX Implementation.

Component	Lines of Code
Trusted	
Trusted SGX Runtime	975
Trusted Platform Service	1,759
Trusted Cryptographic	1,799
Trusted Key Exchange	47
Trusted Protected File System	226
SGX Port of Rust Standard Library	35,214
Others	2,389
Untrusted	
Untrusted SGX Runtime	1,198
Untrusted SGX Cryptographic	1,889
Total	45,496

6 IMPLEMENTATION

As mentioned in §2.1, an SGX program typically has two components: trusted component running inside of the SGX enclave and the untrusted component that interacts with the enclaves. Accordingly, the implementation of Rust-SGX SDK also has two parts: a trusted part and an untrusted part. Most of the design challenges are related to the trusted part. From an engineering perspective, however, the untrusted part is also necessary to allow enclave developers to develop a complete SGX program in Rust.

The trusted part can be further broken down into two major components in Rust-SGX SDK. The first component is the Rust standard library ported into the SGX execution environment. The second component, which depends on the Rust standard library, is the Rust wrappers for Intel SGX SDK APIs. They can be further broken down into several smaller components, including the SGX runtime and platform service, in-SGX cryptography, trusted key exchange, and protected file system, etc.

Table 2 shows the amount of code for each component of Rust-SGX as the time of this writing[3]. In the rest of this section, we share the implementation details of interest for these components.

6.1 Porting Rust Standard Library to SGX

For every language, the standard library is crucial for its programming experience. The porting of a language to a new platform is incomplete if the standard library is excluded. Making the original Rust standard library compatible with SGX is non-trivial. There are some components of Rust `std` that must be refactored before they can be used inside SGX. These components are mostly related to threading, synchronization, and exception handling. Figure 8 illustrates the parts of the Rust `std` that require the refactoring. Rust-SGX only maintains these refactored parts. For the compatible components, we simply reuse the code from the upstream Rust so that we can always keep these components updated.

6.1.1 Threading. Like most modern programming languages that support concurrency, Rust often needs to initialize some data in the thread local storage before launching a thread. As mentioned in §3.2, SGX does not provide a systematic mechanism that allows developers to customize the construction of data in the thread local storage (TLS), so we have to implement such mechanisms by ourselves in Rust-SGX.

What makes the engineering more complicated is that SGX supports a so-called "unbound" threading policy, which allows an untrusted thread to enter the enclave using any existing thread control structure (TCS) allocated inside SGX. This can cause an untrusted thread to corrupt Rust TLS by preempting another untrusted thread when blocked by an `OCALL` operation. However, we are not aware of any secure mechanism to get the identity of the untrusted thread entering the enclave. Therefore, current Rust-SGX does not allow the use of thread local storage when the enclave is configured with the unbound TCS policy.

6.1.2 Mutex. The Rust implementation of mutex is based on an internal interface called `sys::Mutex`. This interface decouples the mutex abstraction visible to programmers from the actual mutex implementation which is system and hardware dependent. By studying the code of `sys::Mutex`, we confirmed that the primitives it provides can be re-implemented using the mutex of Intel SGX SDK. We therefore implemented a wrapper layer to convert the raw mutex of Intel SGX SDK to Rust `sys::Mutex`, enabling the use of Rust standard mutex in SGX.

6.1.3 Exception Handling. To support Rust-style exception handling, we redefined the Rust panic mechanism to support customized panic handlers. We also implemented the whole Rust unwind mechanism, whose standard implementation is not compatible with SGX. In the programming model of Rust-SGX, developers need to first set up their own exception handlers and use our SGX-specific unwind mechanisms to handle the exceptions.

6.2 Static Data Initialization

On most Unix-like platforms, static data in a dynamic library is initialized when the library is loaded by `dlopen`. For SGX, this procedure is different. Static data in an enclave is initialized when the first ECALL happens. The trusted runtime service of SGX has implemented a mechanism to initialize C/C++ static objects defined in an enclave, which are placed in a special ELF section called `.init_array`.

Rust has its own static data initialization process, whose implementation is platform specific. We have to port this process into SGX, using the low-level static initialization functionalities provided by the trusted SGX runtime service. In particular, Rust adopts a lazy initialization scheme, meaning static data will not be fully initialized until the first time it is accessed. Therefore, each static variable has to be associated with an initialization state. Since Rust-SGX supports concurrency, we need to make sure this lazy initialization is thread safe, using the mutex primitives we ported into SGX.

6.3 The Secure Binding Between Rust and C/C++

In section 5, we introduced the formalized type system used to sanitize the interactions between Rust and unsafe languages,

[3]The source code of Rust-SGX has been released on GitHub at https://github.com/baidu/rust-sgx-sdk.

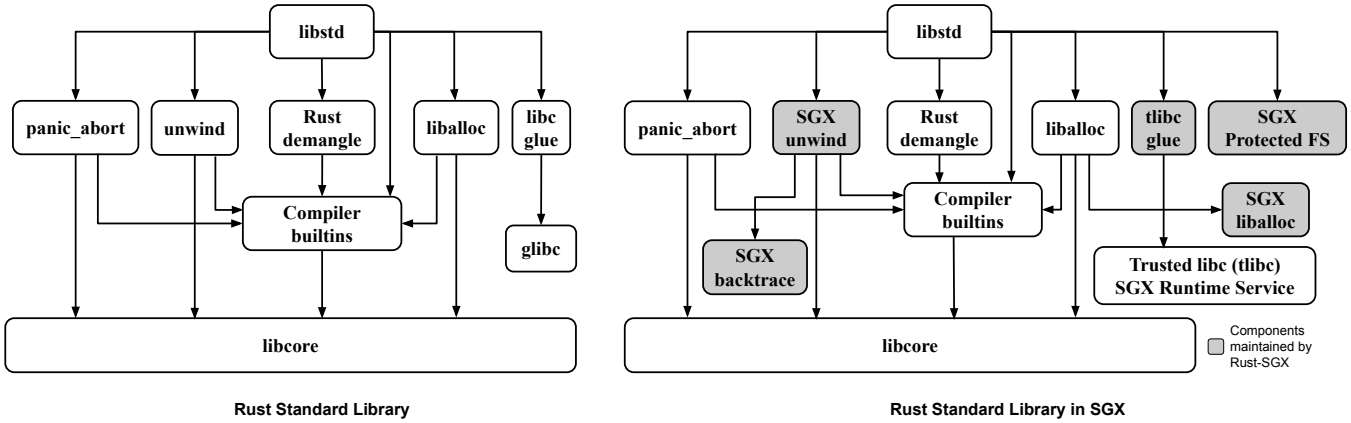

Figure 8: Rust-SGX `std` Compared to Rust `std`.

Table 3: Number of Program Locations Sanitized by Our Secure Rust-C/C++ Binding.

Rust-SGX Component	#Securely Wrapped APIs	#Safe Memory Management	#Safe C/C++ Object Manipulation	#Safe Raw-Byte Memory Access
Trusted SGX Runtime	5	2	5	0
Trusted Platform Service	29	5	24	6
Trusted Cryptographic	37	24	32	27
Trusted Key Exchange	4	4	4	0
Trusted Protected File System	15	15	13	2

particularly the C/C++ interfaces provided by Intel SGX SDK. Our implementation of this sanitization scheme purely relies on the native Rust type system. Table 3 displays the detailed breakdown of program points protected by this type system.

To improve performance, we carefully optimized out some of the redundant operational semantics. For example, the operational semantics of **from_bytes** statements (see Figure 7 in §5.4) require a new chunk of memory to be allocated when constructing a `Sanitizable` Rust object. In some performance-critical functions, we optimize this scheme to an in-place construction style. Each program point optimized by this strategy has been carefully audited to make sure the modification does not lead to any security breaches.

7 EVALUATION

In this section, we provide the performance evaluation of Rust-SGX. We designed two sets of benchmarks to evaluate the performance overhead for software developed with Rust-SGX. One is the microbenchmark that characterizes the overhead of the individual SGX-specific APIs under our Rust bindings compared to Intel SGX SDK (§7.1). The other is the macrobenchmark that characterizes the individual Rust programs developed with Rust-SGX (§7.2).

Our experiments were performed on a machine with Intel I9-9900K CPU, with 64G DDR 3466 RAM and 512G HDD, running 16.04.1-Ubuntu Operating System. As to the CPU microcode level, the revision is 0xb4 with date 2019-04-01. We measured the total execution time of each of the benchmark program by utilizing the operating system clock with the finest granularity of nanoseconds. We run each benchmark multiple times to get the average. The source code of all of our benchmarks is released at Github at https://github.com/mesalock-linux/rust-sgx-benchmark.

7.1 Microbenchmark Test

Our microbenchmark aims to evaluate the overhead of our Rust-bindings with respect to the APIs provided by Intel SGX SDK. Therefore, we used the Intel SGX SDK for the baseline comparison, and installed the latest Intel SGX SDK and corresponding drivers (version 2.4). The comparison was conducted with software developed with Intel SGX SDK and our Rust-SGX SDK.

Since there are hundreds of APIs in Intel SGX SDK, we cannot report the result for all of them. Instead, we only evaluate and report those APIs with Rust bindings that involve SGX specific instructions (e.g., EENTER and EEXIT through `ECALL` and `OCALL`), features (e.g., attestation), and services (e.g., sealing, and unsealing). Meanwhile, these APIs are likely used by the enclave programs multiple times. That is, we did not measure the overhead for those APIs with one-time overhead (e.g., enclave creation and destruction). Therefore, with this criteria, eventually we have 9 microbenchmark programs, and we run them each one million times to get their average overhead.

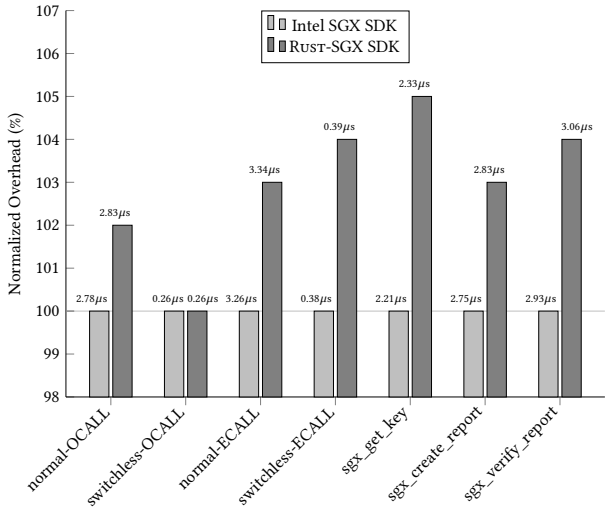

Figure 9: Evaluation Result of Microbenchmark Enclave Enter/Exit, and Attestation.

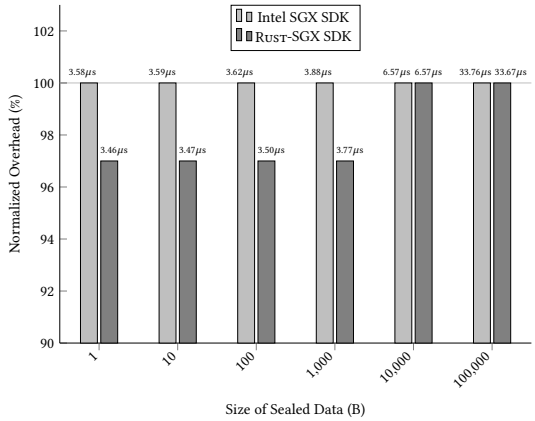

Figure 10: Evaluation Result of Microbenchmark `sgx_seal_data` with Varied Size Input.

- **Enclave Enter and Exit.** Rust-SGX provides APIs for enclave developers to enter and exit an enclave. To evaluate the overhead of this Rust bindings, we have designed four microbenchmarks: normal `ECALL`, switchless `ECALL`, normal `OCALL`, and switchless `OCALL`. Note that Intel provides a collection of switchless features to eliminate enclave transitions from SGX applications. Switchless `ECALL` and `OCALL` characterize our Rust impact on this important feature. The normalized overhead for this experiment is presented in Figure 9. We can see that only switchless-ECALL has a slightly 0.01μs difference (an extra maximum 4% overhead), and the rest are all very small.
- **Trusted Services.** Intel SGX SDK provides a set of APIs for trusted services such as attestation, e.g., retrieving a cryptography key, creating the attestation report, and verifying it. Rust-SGX has a corresponding binding for them as well. We have thus designed three microbenchmarks to test them: `sgx_get_key`, `sgx_create_report`, and `sgx_verify_report`. While creating and verifying a report can take different length of the input, we used an empty one to maximize the potential overhead of our Rust bindings. The normalized overhead for these tests is also presented in Figure 9, and the overhead is ranged from 3% to 5%.

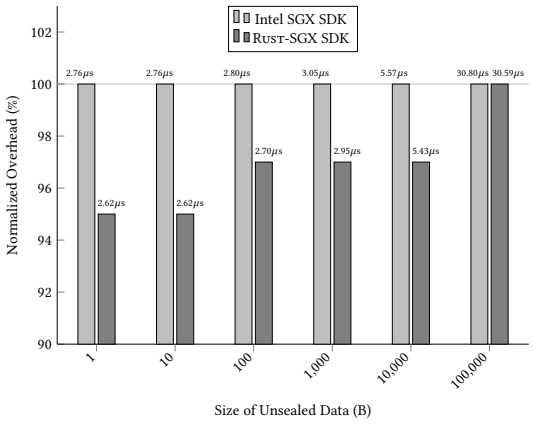

Figure 11: Evaluation Result of Microbenchmark `sgx_unseal_data` with Varied Size Input.

- **SGX Sealing and Unsealing.** An enclave program could seal data in the enclave into disk storage and later unseal it (multiple times). We thus have `sgx_seal_data` and `sgx_unseal_data` to characterize this overhead in our test. Also, for them, we provide different sizes (from 1 byte to 100 kilobytes) of input to evaluate their overhead. From the seal and unseal, we can also infer the overhead of commonly used cryptography functions (e.g., Rijndael AES-GCM, HMAC) since these functions will be involved during sealing and unsealing. The normalized overhead for this experiment is presented in Figure 10 and Figure 11.

  Interestingly, we can notice that the Rust version of sealing/unsealing APIs actually performed slightly faster than Intel's sealing/unsealing for small size input. The reason is that Rust-SGX has largely rewritten the original sealing and unsealing APIs with Rust and made them directly invoke the Intel's AES-GCM cryptography primitives, because the original C APIs did not follow the memory safety principles proposed in this paper and we have to redefine their interfaces and make them type-safe. Our re-implementation is a transparent statement-to-statement C-to-Rust translation but it results in a safer and well-typed interface, and slightly better performance (about 3% for small size input, and no difference for large size input because the processing time dominates the Rust-bindings for them). The observed performance gain may result from the Rust compiler for our particular implementation.

According to the above microbenchmark evaluation, we can see that there is a very light overhead with a maximum 4% compared to the baseline Intel SGX SDK. Therefore, Rust-SGX has small

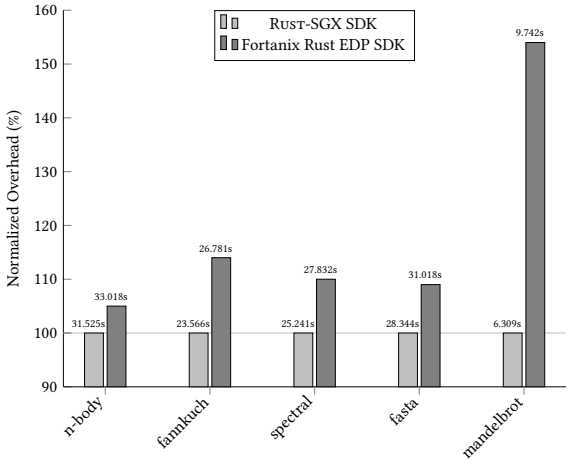

Figure 12: Evaluation Result of our Macrobenchmark Programs Developed with Fortanix Rust EDP and Rust-SGX.

performance sacrifice when built atop Intel SGX SDK, at least with respect to these SGX specific features. We also have to note that there might be noise in our test due to the tiny differences in the timing, though we have minimized this by executing these microbenchmark functions one million times each.

7.2 Macrobenchmark Test

Our macrobenchmark aims to evaluate the end-to-end overhead of running a Rust program developed by Rust-SGX. We cannot compare a Rust program with a C/C++ program since that comparison is inappropriate, as essentially it is the comparison of language specific issues such as compiling, linking, and optimization. Instead, to really quantify the overhead of Rust-SGX, we need to compare to other Rust SGX SDKs. Fortunately, there was a public release of a repository called Fortanix Rust EDP [5] on Github recently, which is also a project to provide a platform for developing Rust programs in SGX enclave. Therefore, we can compare the Rust programs developed with Rust-SGX and Fortanix Rust EDP. Then, we have to look for or develop typical Rust programs. Luckily, we found there is a programming language benchmarking website [2] that provides a set of benchmarks to test different languages. There are 10 rust programs in that website, and we successfully ported five of them and evaluated the overhead with Rust-SGX and Fortanix Rust EDP. The reasons for why the other five programs cannot be ported is: four of them depend on Rayon[4], which requires multi-threading for parallelism, and these 4 benchmarks are incompatible with Intel SGX; the other one depends on `libgmp`, which is not available in SGX.

The evaluation result for these benchmarks is presented in Figure 12. We can see that all five Rust programs running in Fortanix Rust EDP have the overhead ranging from 8% to 54% greater than in Rust-SGX. Note that we compared the running time of these Rust programs directly in Rust-SGX and Fortanix Rust EDP from an end to end perspective by using the operating system clock. The reason that Rust-SGX has better performance than that of Fortanix is due to the design of these two systems. First, Fortanix Rust EDP SDK depends on Rust's libstd, which does not use any optimization features for SGX. Secondly, Fortanix Rust EDP SDK replaces `ECALL/OCALL` designed by Intel with a usercall. This usercall was heavily used by enclave programs so that the performance is significantly affected. Rust-SGX instead has an SGX customized Rust standard library which has all SGX optimizations.

[4]Rayon: A data parallelism library for Rusthttps://github.com/rayon-rs/rayon

8 APPLICATIONS

Rust-SGX is designed as an infrastructure for memory safe SGX enclave program development. To demonstrate this, we present three applications: one is developed by us, and the other two are mostly developed by community contributors. Through the demonstration of these applications, we show that programming with Rust-SGX is productive, efficient, and reliable.

8.1 TLS with SGX Remote Attestation

One of the most attractive security features of Intel SGX is that before sending confidential data, clients can remotely attest an enclave to assure the execution environment is indeed secure. Intel SGX SDK provides a reference implementation for such attestation. Through a modified Sigma protocol [20], the client and the enclave will be able to share a common secret after remote attestation succeeds. Rust-SGX has made such development easy. In this application, we show that Rust-SGX can support implementing complicated secure communication protocols with modest engineering effort.

Theoretically, a shared secret enables the establishment of secure communication channels. However, the functionality of setting up such communication mechanisms is not part of Intel SGX SDK. An Intel white paper [19] proposed to integrate SGX remote attestation into the TLS protocol to provide more friendly interfaces for developers of SGX enclaves and clients. We therefore implemented the algorithm described in the white paper in Rust using Rust-SGX. The in-enclave code consists of only 752 lines of Rust code (excluding certain third-party library dependencies we ported into SGX), with which we are able to validate critical x.509 version 3 extensions. Note that it has been known that verifying X.509 certifications is a difficult and error-prone engineering task

. With Rust-SGX, this has been made easier.

8.2 High-Performance Scientific Computation

Rust-SGX is also suitable for processing large-scale scientific data and meanwhile preserving the confidentiality. In 2017, the IDASH privacy & security workshop [4] held a genomic data privacy and security protection competition. One of the problems is to find the top 10 most significant Single-Nucleotide Polymorphisms in a database of genome records using chi-square tests. Competitors are required to process the data inside SGX enclaves to prevent privacy leakage. The judging criteria of winning is computation time and correctness.

Several participants of this contest chose to develop their solutions on top of Rust-SGX. Table 4 shows the performance of the top teams. Encouragingly, the winning team is the one using Rust-SGX. It is worth noting that none of the winning team members are

Table 4: The Final Results of 2017 IDASH Secure Genome Analysis Competition.

Team	Pre-processing time (s)	The input size after pre-processing (GB)	Computation time (s)	Correctness
CEA†	56	6	7	10/10
Indiana Univeristy	251	2.5	50	10/10
Fujitsu	12	4.7-4.8	94	10/10
UT Dallas/Vanderbilt	480	1.7	14	7/10
U. of Manitoba	203	36	640	10/10

† *The winner team CEA used Rust-SGX to build their contestant program, open sourced at https://github.com/CEA-LIST/sgntx*

involved in the development of Rust-SGX. We interpret this as the evidence of Rust-SGX being user friendly and also performance efficient.

8.3 Machine Learning

The preservation of user privacy in machine learning tasks has become a major concern over AI-boosted computations. SGX provides a reliable platform for secure multi-party computation such that cloud providers, model providers, and clients are securely segregated. In this case study, we show that developers can use Rust-SGX to perform SGX-specific optimization to relax the constraint of the limited SGX physical memory.

In particular, a group of researchers [22] has built an inference engine for the gradient boosted decision tree (GBDT) algorithm using Rust-SGX. By sophisticatedly optimizing the memory access patterns, the in-SGX implementation is able to fit a large volume of data inside the 128MB SGX physical memory with only 10% performance slowdown [22], compared with the same implementation running in the untrusted setting of the same hardware.

9 DISCUSSION

Since the first release in 2017, Rust-SGX has been adopted by many developers in the community to build secure SGX enclaves. The usability and reliability of Rust-SGX has been fully demonstrated through real-world software engineering practice. However, we have to admit that Rust-SGX cannot and intend not to promise absolute security. There are several weaknesses that our secure design methodology fails to cover, but we argue that it is necessary to trade some security for practical engineering benefits. Also, Rust-SGX is not meant to be a mere research prototype. Instead, it aims to be an industry-quality infrastructure to support production software development.

More specifically, one major source of potential insecurity in Rust-SGX is the C/C++ code it built upon. As previously explained, Rust-SGX depends on two groups of C/C++ code. One is the SGX port of the standard C library and the other is the Intel SGX SDK. The dependency on `libc` is inherited from the official Rust language implementation. If we were able to to avoid using `libc`, we would have to extensively refactor the official Rust implementation. This is impractical, since Rust itself is constantly involving at a rapid pace. Introducing a massive amount of code that is incompatible with the upstream makes the maintenance of Rust-SGX extremely difficult.

Similar reasons drove us to build Rust-SGX based on Intel SGX SDK instead of inventing our own enclave ABI with pure Rust abstraction layers, like what the Fortanix Rust EDP did. Intel SGX SDK provides high-quality implementations of various cryptography algorithms written in C/C++ and assembly. Reimplementing them in Rust, if possible at all, can lead to problems like performance degradation, incompatibility with NIST standards, and side channels. On the other hand, Intel is the provider of a number of libraries important to many privacy-preserving applications, e.g., the Intel Math Kernel Library. Intel has been working on porting these libraries into SGX and the ported versions will very likely depend on Intel SGX SDK.

10 RELATED WORK

Attacks and Defenses with Enclave Memory Corruption. Enclave software developed by systems language such as C/C++ can have memory corruption vulnerabilities, which can be exploited. Dark-ROP [21] is the first memory corruption attack against SGX. It exploited a memory corruption vulnerability in SGX enclave by utilizing return-oriented programming (ROP). ROP attacks traditionally rely on gadget instructions like return statement where an attacker can control execution by inserting malicious function addresses into the stack. However, performing ROP attacks in SGX is significantly different from traditional ROP attacks since target code is running under protection of hardware. Enclave code and data are not accessible from outside of the enclave. Dark-ROP achieves its attack by exploiting the exception handling mechanism in SGX, it constructs three oracles which will give a hint for attackers about internal status of enclave execution so that attackers could use it to find the useful gadgets for exploits. However Dark-ROP has strong assumption that the memory layout is constant and not randomized.

SGX-Shield [28] performs Address Space Layout Randomization (ASLR) to SGX environment for preventing memory-corruption. It employs fine-grained randomization along with non-readable code to render traditional attacks difficult yo perform. However, the randomization is not fully implemented in SGX. A safe transition between host code and enclave code in trusted run-time library is not presented, leaving a window for attacks.

The Guard's Dilemma work [9] exploited memory corruption with efficient code-reuse attacks against Intel SGX even with fine-grained randomization protected by SGX-Shield. Previous attacks depend on either enclave crashes, kernel privileges, or strong

assumption that memory layout is constant and non-randomized. The Guard's Dilemma bypasses the enclave crashes and ALSR provided by SGX-Shield. In particular, since SGX-Shield does not have trusted run-time randomized, there is an open avenue for attackers. The Guard's Dilemma employs this knowledge to implement run-time attacks and bypass SGX-Shield without kernel privilege.

System and Language Support for SGX. Various efforts have been made to ease the development and deployment of SGX enclave programs. The library-OS line of work, including Haven [8], SCONE [7], PANOPLY [30], and Graphene-SGX [33], aims to support legacy programs in SGX without modifications. These approaches mimic low-level system programming primitives (e.g., POSIX system calls and pthread APIs) inside SGX such that programs are not aware of the hardware environment change. To the best of our knowledge, all existing Library-OSes are implemented in unsafe languages and they significantly increase the TCB size (from 20K to millions of lines of code). Moreover, since legacy programs are not designed to align with the SGX threat model, some of them can be vulnerable to the infamous Iago attack [12] even if they can run as secure enclaves.

Memory safe Languages other than Rust have been ported to SGX to enable secure and productive enclave development. MesaPy[5] is an open source Python spin-off for SGX. It is based on PyPy and mostly written in RPython, a typed Python dialect. Part of its C code is formally verified by commercial abstract interpreters. GOTEE [14] is a research project porting the Go language runtime into SGX, allowing programmers to execute a goroutine within an enclave.

Rust Security and Use in SGX. RustBelt [17] was created to secure the principles of the Rust Programming Language by formal verification. As a memory and type safe language, none of Rust's safety has been formally proven. Rustbelt is the first formal safety proof for a subset of Rust, and also includes machine-checked safety proof. Rustbelt designed a language that formalized the static and dynamic semantics of features of Rust type system and proved the fundamental theorem by using the Coq proof assistant. RustBelt has proven the safety of a few important Rust standard library members using unsafe code, most of which are related to memory management and synchronization.

Since the first release of Rust-SGX in 2017, our methodology of building secure enclaves using memory safe languages has inspired similar efforts. The most related one is the Fortanix Rust Enclave Development Platform (Fortanix Rust EDP) [5] released in 2019. Fortanix EDP invented its own application binary interface with the SGX hardware. Fortanix Rust EDP has a slightly different engineering focus compared with Rust-SGX. Specifically, Rust-SGX focused more on memory safety and compatibility with trusted execution environment, whereas Fortanix Rust EDP engaged more in memory safety and compatibility with Rust's standard library. Some portions of the Rust's standard library cannot be securely implemented in SGX, such as environment variable, timing, and networking. Rust-SGX chooses not to provide these components. Users who need these functionalities have to find their own solutions and be responsible for their choices. While Fortanix EDP tries to be fully compatible with standard Rust. Everything that cannot be implemented inside SGX is delegated to the untrusted applications and OS outside of the enclaves.

[5]https://github.com/mesalock-linux/mesapy

11 CONCLUSION

We have presented Rust-SGX, an SDK for SGX programmers to develop memory safe enclave programs. We did not build it from scratch in Rust, and instead we used a layered approach by building it atop Intel SGX SDK with a formally proven secure interface between the Rust and C/C++ world. We have implemented Rust-SGX and tested with a number of benchmarks. Our evaluation results show that Rust-SGX SDK impose little extra overhead, at least for the SGX related services, compared to the software directly developed with Intel SGX SDK. Overall, while enclave programs developed with Rust-SGX may not have the same performance as with Intel SGX SDK, they will have stronger memory safety at the application layer because of the use of Rust.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers and our shepherd Frank Piessens for their very helpful comments. Also, Huibo Wang and Zhiqiang Lin were partially supported by the NSF grants 1834213 and 1834216, as well as a research gift from Baidu X-Lab.

REFERENCES

[1] 1988. Morris worm. https://en.wikipedia.org/wiki/Morris_worm. (1988).
[2] 2007. The Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/. (2007).
[3] 2014. Intel Software Guard Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf. (Oct. 2014).
[4] 2017. IDASH workshop 2017. http://www.humangenomeprivacy.org/2017/competition-tasks.html. (2017).
[5] 2019. The Fortanix Rust Enclave Development Platform. https://edp.fortanix.com. (2019).
[6] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow Integrity Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur.* 13, 1, Article 4 (Nov. 2009), 40 pages. https://doi.org/10.1145/1609956.1609960
[7] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 689–703. http://dl.acm.org/citation.cfm?id=3026877.3026930
[8] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst.* 33, 3, Article 8 (Aug. 2015), 26 pages. https://doi.org/10.1145/2799647
[9] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. 2018. The Guard's Dilemma: Efficient Code-reuse Attacks Against Intel SGX. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC'18)*. USENIX Association, Berkeley, CA, USA, 1213–1227. http://dl.acm.org/citation.cfm?id=3277203.3277294
[10] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking Blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*. IEEE Computer Society, Washington, DC, USA, 227–242. https://doi.org/10.1109/SP.2014.22
[11] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-oriented Programming: A New Class of Code-reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS '11)*. ACM, New York, NY, USA, 30–40. https://doi.org/10.1145/1966913.1966919
[12] Stephen Checkoway and Hovav Shacham. 2013. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. *SIGPLAN Not.* 48, 4 (March 2013), 253–264. https://doi.org/10.1145/2499368.2451145

[13] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7 (SSYM'98)*. USENIX Association, Berkeley, CA, USA, 5–5. http://dl.acm.org/citation.cfm?id=1267549.1267554

[14] Adrien Ghosn, James R. Larus, and Edouard Bugnion. 2019. Secured Routines: Language-based Construction of Trusted Execution Environments. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 571–586. https://www.usenix.org/conference/atc19/presentation/ghosn

[15] Marcus Hähnel, Weidong Cui, and Marcus Peinado. 2017. High-resolution Side Channels for Untrusted Operating Systems. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '17)*. USENIX Association, Berkeley, CA, USA, 299–312. http://dl.acm.org/citation.cfm?id=3154690.3154719

[16] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 969–986.

[17] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages. https://doi.org/10.1145/3158154

[18] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. 2006. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *Proceedings of the 22Nd Annual Computer Security Applications Conference (ACSAC '06)*. IEEE Computer Society, Washington, DC, USA, 339–348. https://doi.org/10.1109/ACSAC.2006.9

[19] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. 2018. Integrating Remote Attestation with Transport Layer Security. *arXiv preprint arXiv:1801.05863* (2018).

[20] Hugo Krawczyk. 2003. SIGMA: The 'SIGn-and-MAc 'approach to authenticated Diffie-Hellman and its use in the IKE protocols. In *Annual International Cryptology Conference*. Springer, 400–425.

[21] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. 2017. Hacking in Darkness: Return-oriented Programming Against Secure Enclaves. In *Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17)*. USENIX Association, Berkeley, CA, USA, 523–539. http://dl.acm.org/citation.cfm?id=3241189.3241231

[22] Tianyi Li, Tongxin Li, Yu Ding, Yulong Zhang, Tao Wei, and Xinhui Han. 2019. Poster: `gbdt-rs`: Fast and Trustworthy Gradient Boosting Decision Tree. Posters In 2019 IEEE Symposium on Security and Privacy (SP).

[23] Nicholas D. Matsakis and Felix S. Klock, II. 2014. The Rust Language. *Ada Lett.* 34, 3 (Oct. 2014), 103–104. https://doi.org/10.1145/2692956.2663188

[24] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '13)*. ACM, New York, NY, USA, Article 10, 1 pages. https://doi.org/10.1145/2487726.2488368

[25] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-safe Retrofitting of Legacy Software. *ACM Trans. Program. Lang. Syst.* 27, 3 (May 2005), 477–526. https://doi.org/10.1145/1065887.1065892

[26] Aleph One. 1996. Smashing the stack for fun and profit. *Phrack magazine* 7, 49 (1996), 14–16.

[27] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, Washington, DC, USA, 745–762. https://doi.org/10.1109/SP.2015.51

[28] Jaebaek Seo, Byounyoung Lee, Seongmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. 2017. SGX-Shield: Enabling address space layout randomization for SGX programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*.

[29] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. ACM, New York, NY, USA, 552–561. https://doi.org/10.1145/1315245.1315313

[30] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. PANOPLY: Low-TCB Linux Applications With SGX Enclaves. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS '17)*.

[31] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, Washington, DC, USA, 574–588. https://doi.org/10.1109/SP.2013.45

[32] Gang Tan, Andrew W Appel, Srimat Chakradhar, Anand Raghunathan, Srivaths Ravi, and Daniel Wang. 2006. Safe Java native interface. In *Proceedings of IEEE International Symposium on Secure Software Engineering*, Vol. 97. Citeseer, 106.

[33] Chia-Che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '17)*. USENIX Association, Berkeley, CA, USA, 645–658. http://dl.acm.org/citation.cfm?id=3154690.3154752

[34] Arjan van de Ven and Ingo Molnar. 2004. Exec shield. http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf. *Retrieved March* 1 (2004), 2017.

[35] RN Wojtczuk. 2001. The advanced return-into-lib (c) exploits: PaX case study. *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e* (2001).

[36] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, Washington, DC, USA, 640–656. https://doi.org/10.1109/SP.2015.45