POSTER: Rust SGX SDK: Towards Memory Safety in Intel SGX Enclave

Yu Ding, Ran Duan, Long Li, Yueqiang Cheng, Yulong Zhang, Tanghui Chen, Tao Wei
Baidu X-Lab
Sunnyvale, CA
{dingyu02,duanran01,lilong09,chengyueqiang,ylzhang, chentanghui,lenx}@baidu.com

Huibo Wang*
UT Dallas
Richardson, Texas
hxw142830@utd.edu

ABSTRACT

Intel SGX is the next-generation trusted computing infrastructure. It can effectively protect data inside enclaves from being stolen. Similar to traditional programs, SGX enclaves are likely to have security vulnerabilities and can be exploited as well. This gives an adversary a great opportunity to steal secret data or perform other malicious operations.

Rust is one of the system programming languages with promising security properties. It has powerful checkers and guarantees memory-safety and thread-safety. In this paper, we show Rust SGX SDK, which combines Intel SGX and Rust programming language together. By using Rust SGX SDK, developers could write memory-safe secure enclaves easily, eliminating the most possibility of being pwned through memory vulnerabilities. What's more, the Rust enclaves are able to run as fast as the ones written in C/C++.

CCS CONCEPTS

• **Security and privacy → Trusted computing**; **Software security engineering**;

KEYWORDS

Intel SGX; Rust programming language; SDK

1 INTRODUCTION

Intel SGX provides a hardware based Trusted Execution Environment (TEE) called 'SGX enclave', along with Intel Active Management Technology (AMT) module and Internet remote attestation infrastructure for the whole Intel SGX ecosystem. The core of Intel SGX technique is the memory encryption engine [7] in CPU and the CPU is the key component inside the trusted boundary. It limits memory access by enforcing checks on TLB access and memory address translation. Also, the memory encryption engine automatically encrypts data when evicting pages to the untrusted memory region. However, SGX enclaves could have memory corruption vulnerabilities and could be exploited and hi-jacked [9, 12] and thus secrets would be leaked in such attacks. Researchers have proposed several techniques for hardening Intel SGX [8, 11], but these solutions are only exploit mitigations. We still need an ultimate solution with memory safety guarantee for Intel SGX enclaves.

Rust programming language [10] is becoming more and more popular in system programming. It intrinsically guarantees memory safety and thread safety. The performance of Rust program is almost the same to C++ program [1]. Servo [5] and Redox [3] are browser and operating system written in Rust, indicating that Rust can do almost everything on popular architectures. We believe that Rust best fits for developing basic system components.

In this paper, we show Rust SGX SDK a framework that connects Intel SGX and Rust programming language, making it easy for developers to write safe and memory-bug-free SGX enclaves. By building enclaves in Rust on top of our Rust SGX SDK, there is no need for adapting any advanced exploit mitigation techniques such as ASLR, ROP gadget mitigation or CFI enforcement.

Rust SGX SDK has been open-sourced on Github [4]. Intel recommends this SDK on its official SGX homepage.

In summary, our contributions are:

(1) We first introduce Rust ecosystem to the Intel SGX community, bringing both security and functionality to Intel SGX programming.
(2) We propose the rules-of-thumb in memory safe/unsafe hybrid SDK designing to achieve good balance between security and functionality.
(3) We identify the main challenges in connecting Intel SGX and Rust, and then we design and build Rust SGX SDK that addresses all identified challenges efficiently and effectively.

2 RATIONALE AND CHALLENGES

2.1 Memory safety rules-of-thumb

Based on whether using Intel SGX SDK or not, there are two reasonable directions to build Rust SGX SDK. The first one is to build it from scratch, without relying on the Intel's SGX SDK. The pure Rust version could achieve better safety, but it weakens the functionality. It would become even worse with many new features added into the Intel SGX SDK periodically. `sgx-utils` [6] is such an open-source project and but it is outdated now. The second direction is to build it upon the Intel SGX SDK. It is not a pure Rust version, but it could significantly benefit from Intel's efforts, and is able to achieve good balance between safety and functionality with carefully designed architecture. In our Rust SGX SDK project, we choose the second one. To get better functionality along with strong

*Huibo Wang contributed to this work during her internship at Baidu X-Lab.

CCS'17, , October 30–November 3, 2017, Dallas, TX, USA
© 2017 Copyright held by the owner/author(s).
ACM ISBN ISBN 978-1-4503-4946-8/17/10.
https://doi.org/10.1145/3133956.3138824

security guarantees, we come up with the following rules-of-thumb for hybrid memory-safe architecture designing:

(1) Unsafe components should be appropriately isolated and modularized, and the size should be small (or minimized).
(2) Unsafe components should not weaken the safe, especially, public APIs and data structures.
(3) Unsafe components should be clearly identified and easily upgraded.

Here the unsafe components include both the modules written in memory-unsafe languages (such as C/C++), and the `unsafe` codes which reside in the modules written in memory-safe languages (such as Rust). Memory-safety oriented SGX SDK would benefit a lot from following these rules. Enclaves built on top of our Rust SGX SDK would benefit from the strong safety guarantees, as well as new features and performance optimizations brought by Intel.

2.2 Main Challenges

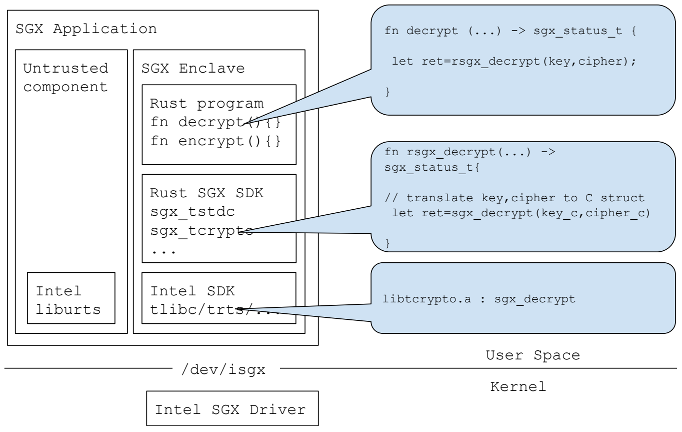
The first real challenge is threading. Thread of Intel SGX enclave does not have its own life cycle, no matter the enclave's `TCSPolicy` is BOUND or UNBOUND [2]. In an enclave with BOUND TCSpolicy, an enclave 'thread' consumes one fixed 'TCS' slot, and binds its life to a POSIX thread. When the POSIX thread exits, the corresponding enclave 'thread' releases the occupied 'TCS' thread slot . In an enclave with UNBOUND TCSpolicy, an enclave 'thread' comsumes an available 'TCS' slot from the pool and releases it on `EEXIT`. As a result, raw SGX thread has neither constructor nor destructor, and TLS data even remains after `EEXIT`. All these characterstics conflict with Rust thread model.

The second challenge is the initiation of static data. For example, C++0x allows initializer lists for standard containers. In normal user space applications, the global data structures are initiated before `main` begins. However, in SGX, there is no such initiation procedure. After the enclave is loaded, nothing would be executed until the first `ECALL` instruction. How to implement the static data initiation in Rust is a challenge.

The third challenge is to implement Rust style mutex using SGX style mutex. The SGX style mutex is very similar to pthread mutex. But Rust mutex is vastly different from it. Rust mutex directly binds to the protected data and needs to be constructed together with the data it binds to. We need to properly implement Rust style mutex using SGX style mutex.

3 SDK OVERVIEW

Fig. 1 shows the architecture of our Rust SGX SDK. In the trusted environment, a.k.a. SGX enclave, the SGX enclave is loaded into the protected memory along with the enclave's metadata. The SGX enclave binary is linked to Intel SDK libraries, such as `libsgx_tstdc.a`. Intel SGX SDK exposes standard C/C++ interface to the upper level. Rust SGX SDK is built on top of these Intel SGX SDK libraries, providing Rust style data structures and APIs to developers. For example, Rust has its own vector data structure `collections::vec::Vec` and Rust SGX SDK includes the implementation and exports it. In addition, Intel SGX SDK has APIs such as `sgx_rijndael128GCM_encrypt` and Rust SGX SDK re-exports them in Rust calling convention, e.g., using name `rsgx_rijndael128GCM_encrypt` for the above API. In this way, Rust SGX SDK allows developers to write Rust codes in Intel SGX enclaves.



```
fn decrypt (...) -> sgx_status_t {

 let ret=rsgx_decrypt(key,cipher);

}
```

```
fn rsgx_decrypt(...) ->
sgx_status_t{

// translate key,cipher to C struct
 let ret=sgx_decrypt(key_c,cipher_c)

}
```

```
libtcrypto.a : sgx_decrypt
```

Figure 1: Overview of Rust SGX SDK

Rust SGX SDK is written in Rust using 19K SLoC. We ship it along with code samples and documents and has 44K SLoC in total.

4 SOLUTION

For the threading problem, we limit the ability of Intel SGX threading.

(1) For the enclave with BOUND TCSpolicy, our Rust SGX SDK supports TLS. Developers could write constructor function for the TLS data in Rust SGX enclave. But in the current version of Rust SGX SDK, these constructors would not be executed automatically , which means that developers need to initiate TLS data explicitly. Destructors of TLS data are unsupported in the current version. Supporting destructors and automatically destructing TLS data require re-writing the code in the untrusted part, which will be our next step.
(2) For the enclave with UNBOUND TCSpolicy, our Rust SGX SDK does not support TLS. The reason is that in such programs, every `ECALL` would trap into the enclave with an undetermined TCS slot. So it is impossible to support TLS in this scenario. The only way to support TLS data in UNBOUND TCSpolicy is software simulation, instead of using native TCS slot.

We also implemented Rust style `park` and `unpark` for threading control.

To support global data initiation, we have done the following:

(1) We utilize the undocumented function `init_global_object` provided in `libsgx_trts` from Intel SGX SDK. This function would retrieve the `.init_array` section of the enclave and initialize them during the first `ECALL`. It gives us the ability to initiate global data.
(2) We implemented a Rust macro `init_global_object!` to put data in a special section : `.init_array`. By using this macro, developers could put data directly into the global data section, which will be initiated during the first `ECALL`.

```
#[no_mangle]
pub extern "C" fn say_something(some_string: *const u8, some_len: u32) -> sgx_status_t {
    unsafe {
        ocall_print_string(some_string as *const c_uchar, some_len as size_t);
    }
    let hello_string = "This is a Rust String!";
    unsafe {
        ocall_print_string(hello_string.as_ptr() as *const c_uchar,
                           hello_string.len() as size_t);
    }
    sgx_status_t::SGX_SUCCESS
}
```

Figure 2: A helloworld enclave code sample in Rust

(3) We implemented Rust style `Once` in Rust SGX SDK by utilizing native `std::sync::atomic::AtomicPtr`. `Once` is useful in such one-time initialization.

(4) We ported a Rust crate `lazy_static` with the support of `Once`. `lazy_static` is the most easy-to-use and well adopted crate to initiate global data.

With the above supports, Rust SGX SDK offers the global data initiation elegantly.

To provide Rust style mutex, we looked into Rust's source code. We found that Rust's implementation of mutex is based on `sys::Mutex`. The primitives provided by `sys::Mutex` can be re-implemented using Intel SGX's mutex. Based on this observation, we implemented a wrapper layer to convert Intel SGX's raw mutex to Rust style raw mutex. Thus we can smoothly port Rust mutex over the wrapper layer.

To support Rust style exception handling, we redefined Rust 'panicking' and 'panic' mechanism. In Rust, throwing an exception is triggering the `panic!` macro and catching an exception requires `panic::unwind` function. Rust's `std` provides this mechanism. But in SGX, we do not have `std`, same as many embedded systems. To solve this, we provided panic handling setter `set_panic_handler` to customize panic handler, and we implemented the whole unwind mechanism to support `panic::unwind`. Developers need to customize the exception handler at first, and use `panic::unwind {..}` to handle all exceptions.

5 A RUNNING EXAMPLE

Fig. 2 shows an example of Rust code to print "Hello World" from a Rust enclave. This example is self-explained and easy to understand with basic knowledge of Rust and SGX. For more code examples, please refer to our Github repository [4].

6 FUTURE WORK

The current latest release of Rust SGX SDK is v0.2.0. In our v1.0.0 version, we plan to provide a full-fledged `sgx_tstd` library which could be used as `std` inside enclave. By leveraging this library, it is easier for developers to port third-party crates into Intel SGX. Meanwhile, we are also working on an untrusted runtime library `sgx_urts`, which enables the development of Rust code in the SGX untrusted environment as well.

REFERENCES

[1] BENCHMARKING DYNAMIC ARRAY IMPLEMENTATIONS. https://lonewolfer.wordpress.com/2014/09/24/benchmarking-dynamic-array-implementations/.

[2] Intel Software Guard Extensions SDK for Linux OS Developer Reference. https://download.01.org/intel-sgx/linux-1.9/docs/Intel_SGX_SDK_Developer_Reference_Linux_1.9_Open_Source.pdf.

[3] Redox OS. https://www.redox-os.org/.

[4] Rust SGX SDK. https://github.com/baidu/rust-sgx-sdk.

[5] Servo, the Parallel Browser Engine Project. https://servo.org/.

[6] sgx-utils. https://github.com/jethrogb/sgx-utils.

[7] S. Gueron. 2016. Memory Encryption for General-Purpose Processors. *IEEE Security Privacy* 14, 6 (Nov 2016), 54–62. https://doi.org/10.1109/MSP.2016.124

[8] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 205–221. https://doi.org/10.1145/3064176.3064192

[9] 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 *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 523–539. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-jaehyuk

[10] 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

[11] Jaebaek Seo, Byounyoung Lee, Seongmin Kim, Ming-Wei Shih, Insik Shin, 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)*.

[12] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. 2016. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *European Symposium on Research in Computer Security*. Springer International Publishing, 440–457.