Building Secure SGX Enclaves using F*, C++ and X64

Presentation Proposal for PRISC'18

Anitha Gollamudi
Harvard University
agollamudi@g.harvard.edu

Cédric Fournet
Microsoft Research
fournet@microsoft.com

Abstract

Intel SGX offers hardware mechanisms to isolate code and data running within enclaves from the rest of the platform. This enables security verification on a relatively small software TCB, but the task still involves complex low-level code.

Relying on the Everest verification toolchain, we use F* for developing specifications, code, and proofs; and then safely compile F* code to standalone C code. However, this does not account for all code running within the enclave, which also includes trusted C and assembly code for bootstrapping and for core libraries. Besides, we cannot expect all enclave applications to be rewritten in F*, so we also compile legacy C++ defensively, using variants of /guard that dynamically enforce their safety at runtime.

To reason about enclave security, we thus compose different sorts of code and verification styles, from fine-grained statically-verified F* to dynamically-monitored C++ and custom SGX instructions. This involves two related program semantics: most of the verification is conducted within F* using the target semantics of Kremlin—a fragment of C with a structured memory—whereas SGX features and dynamic checks embedded by defensive C++ compilers require lower-level X64 code, for which we use the verified assembly language for Everest (VALE) and its embedding in F*.

1 Introduction

Hardware features like Intel SGX [5], ARM TrustZone [1], and virtualization enable applications to build and deploy protected subsystems, or *enclaves*, for their security-critical functionalities. An enclave provides isolated execution, and is thus an attractive option to defend against powerful adversaries that may control the rest of the software stack.

We consider typical SGX subsystem whose software consist of an SDK for managing enclaves and their security features (e.g., hardware-protected keys and remote attestation), some supporting libraries (e.g., cryptography), and some user application. The SDK is written using a mix of C/C++ and X64 assembly (e.g. for SGX instructions). Any bug in code running within the enclave can lead to serious vulnerabilities. For example, a malicious application could gain access to memory outside the enclave, or exploit a poorly designed library to leak enclave secrets, such as their signing keys.

We intend to verify the safety and security of enclave subsystems using F* [9], leveraging the Everest verification infrastructure [2, 4]. Compared with general-purpose systems, this task seems relatively easy, inasmuch as each enclave implements simple, specific application code with a clear security model; each enclave include only a few fixed, minimal libraries and runtimes; and thus, unlike their hosts, enclaves need not trust much legacy code. However, this task is challenging for multiple reasons, described below.

(1) It involves low-level system code, consisting of C and inlined X64 assembly. We address this challenge by verifying that code using a combination of Low*—a shallow embedding of a small, sequential, well-behaved subset of C in F*, and of VALE [3, 7]—a deep embedding of X64 assembly within F*.

(2) Low* and VALE operate at different levels of abstraction. The former is used to verify programs with structured memory (e.g., C-like arrays and structures, allocated on the stack or on the heap) whereas the latter is used to verify lower-level assembly programs with flat memory (e.g. registers and a RAM of machine integers) where all structure information is lost. Composing them securely requires techniques to reconcile these two levels of abstractions.

(3) Enclaves are usually coded in unsafe languages such as C and C++. Writing entire enclaves in F* is costly, and convincing enclave programmers to use F* is hard.

(4) Enclave applications are usually ported from legacy code with no functional specification, hence the most we can hope for is to enforce their generic runtime safety and isolation. This requires composing those coarse properties with finer-grained properties obtained by deeper verification of critical enclave components.

(5) Enclaves functionally rely on services provided by the trusted host. This involves an assembly-level protocol for entering and exiting enclaves, and for marshalling their inputs and outputs across the enclave boundary.

To enforce secure composition among various components of an SGX system, we must precisely express what we need to verify, as well as our trust assumptions in the unverified components. In the following sections, we elaborate on how we are addressing these challenges.

This is work in progress—we have enclaves running code compiled from Low and VALE, and small-scale verification experiments, but we still have to integrate them.*

Dual Semantics We compose programs at two different levels: a *defensive C semantics* and a lower-level *platform semantics*. Defensive C semantics is the target of the C backend of F*, named Kremlin, which compiles Low* to Clight, a

formal semantics of C used by Compcert [6]. It uses a structured memory model similar to that of C. The data stack is made explicit, but control flow is still abstract. Since compilation preserves and reflects the behavior of the source language, we use Low* for reasoning about C-like programs written in F*. Programs compiled using the defensive semantics are type-safe, hence memory-safe. We need such a strong property if we are to prove anything compositional.

Our Platform semantics is a deep embedding of VALE in F*: assembly code are lists of X64 instructions interpreted in F*; their properties are specified as pre- and post-conditions of their interpretation. (These lists can of course also be printed and compiled to machine code.) VALE operates at a lower level, which is required for at least two reasons. First, this is the level that a program executes and we care to provide defenses against assembly-code software attacks. Second, we rely on custom hardware security instructions that are specified at this level. Verifying programs using platform semantics is expensive. To get strong security guarantees for SGX subsystems without having to prove everything at the platform level, we tightly integrate these two semantics using a dual memory model in F*, described next.

Dual Memory Model We program in F* a state monad that maintains two related view of the memory: *structured* and *flat*. During the transition from defensive to platform world, the structured view of the memory is mapped to the flat view and the verification is carried out using this flat view of the memory. At the transition boundaries, the pre- and post-conditions are verified in both views. The transition in the opposite direction is handled analogously. Specifically, the embedding of Vale semantics in F* ensures that the composition is secure.

Trusted Components Pragmatically, we address the third challenge by initially fixing the common code inside an enclave (i.e., the SGX SDK and the supporting libraries) to be part of the Trusted Code Base (TCB) and then gradually replacing the components of an enclave subsystem with similar and statically verified components. Conversely, C++ legacy applications and the host software are not part of the TCB.

Transitions between User Code and SDK within Enclaves We address the fourth challenge by requiring that user applications (U) be sandboxed within some enclave memory region, using for instance some dynamic checks inserted by a defensive C++ compiler, such that the user application at least complies with memory access control: U may read and write data only within its statically-assigned region of the enclave [8]. The components U and V interact with one another through controlled entry points. Given a defensive semantics and a manifest that specifies these entry points, we automatically generate Low* wrappers for safely calling U from V. These wrappers reflect the dynamic safety of the code compiled from C++ using precise Low* type, enabling us for instance to reason about the security of keys and private mutable state held in Low* as if U were coded in F*.

Transitions between Enclaves and their Hosts We address the fifth challenge by explicitly modelling and verifying the protocol between enclaves and their untrusted hosts. Verified code running inside an enclave can call the host operating system to serve an interrupt or a system call (note that direct system calls are illegal from within the enclave). We automatically generate safe wrappers for carrying out such transitions. Simiarly, the wrappers enforce that arguments passed to the host and return values received from the host are well formed. The wrappers reflect our modelling of the untrusted host by guaranteeing that the protected regions of memory inside the enclave are left unchanged.

Evaluation As a preliminary experiment to validate our approach, we have built and partially verified an SGX enclave that runs a variant of *Pneutube*, a cryptographic sample application for Low* [7].

This application implements an asynchronous secure file-transfer server. The client connects to the server and issues commands to send or receive files. The server, upon receiving a command, encrypts or decrypts the file accordingly. The SDK provides (trusted) support to enter and exit the enclave, enabling the server to temporarily exit the enclave to execute system calls (e.g., `socket()`) on the host. Pneutube is built on top of HACL* [10], a cryptographic library verified using Low* (for code extracted to C) and VALE (for code extracted to assembly). Overall, we thus reduce the safety, correctness, and security of our *Pneutube* enclave to F* verification, and assumptions on the SGX hardware, our enclave SDK, and the cryptographic security of standard encryption and authentication algorithms.

Acknowledgments This work was done at Microsoft Research. We thank the Everest team for their contributions and their support.

References

[1] ARM. [n. d.]. ARM Security Technology — Building a Secure System using TrustZone Technology. ([n. d.]).

[2] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch, Kenji Maillard, Jianyang Pan, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella-Béguelin, and Jean-Karim Zinzindohoué. 2017. Everest: Towards a Verified, Drop-in Replacement of HTTPS. In *2nd Summit on Advances in Programming Languages (SNAPL 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.), Vol. 71. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 1:1–1:12.

[3] Barry Bond, Chris Hawblitzel, Manos Kapritsos, M. Leino K. Rustan, Jacob R. Lorch, Parno Bryan, Rane Ashay, Setty Srinath, and Thompson Laure. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. *In Proceedings of the USENIX Security Symposium* (2017).

[4] Project Everest. 2016. Everest VERified End-to-end Secure Transport. https://project-everest.github.io/. (2016).

[5] Intel. 2014. Intel Software Guard Extensions (Intel SGX) Programming Reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf. (2014).

[6] Xavier Leroy. 2006. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM symposium on Principles of Programming Languages*. ACM Press, 42–54.

[7] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified Low-level Programming Embedded in F*. *Proceedings of the ACM on Programming Languages* 1, ICFP (Aug. 2017).

[8] Rohit Sinha, Manuel Costa, Akash Lal, Nuno Lopes, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. 2016. A Design and Verification Methodology for Secure Isolated Regions. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[9] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying Higher-order Programs with the Dijkstra Monad. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. New York, NY, USA, 387–398.

[10] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL * : A Verified Modern Cryptographic Library. In *ACM Conference on Computer and Communications Security (CCS)*. Dallas, United States.