Enabling Verifiable Execution of Distributed Secure Enclave Platforms

Saharsh Agrawal
Karen Tu

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-153
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-153.html

May 21, 2021

Acknowledgement

To my always supportive friends and family.

Enabling Verifiable Execution of Distributed Secure Enclave Platforms

by Saharsh Agrawal

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

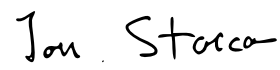
Committee:

Professor Raluca Ada Popa
Research Advisor

May 18, 2021

(Date)

* * * * * * *



Professor Ion Stoica
Second Reader

May 20, 2021

(Date)

Enabling Verifiable Execution of Distributed Secure Enclave Platforms

by

Saharsh Agrawal

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master's

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Raluca Ada Popa, Chair
Professor Ion Stoica

Spring 2021

Enabling Verifiable Execution of Distributed Secure Enclave Platforms

To my always supportive friends and family.

Abstract

Enabling Verifiable Execution of Distributed Secure Enclave Platforms

by

Saharsh Agrawal

Master's in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Raluca Ada Popa, Chair

Outsourcing data computations to a cloud provider is a common way to process large datasets. However, a user might not trust the cloud provider with sensitive data, and enclaves are a promising way to ensure data confidentiality and integrity. For distributed applications, code that affects data flow but not the data contents can be placed outside of the enclave; the execution of the data flow can be verified to have happened correctly. However, there are no existing frameworks to perform this verification.

We propose an execution flow verification library. Our library contributes (i) a way to securely log inputs and outputs of enclave functions, (ii) a verification strategy based on a ruleset specification and (iii) automatic API integration and ruleset generation. This saves developers from having to write their own custom application-specific verification code. Our library provides data flow integrity at the cost of a reasonable code footprint of about 500 lines and a latency overhead of roughly 3%.

An orthogonal line of research over the past several years has been blockchain and distributed ledger platforms, some with smart contract capabilities. Supporting private data and computation on such platforms using secure enclaves (e.g. Intel SGX) has become of interest as of late. Hyperledger Fabric Private Chaincode (FPC) is one such project; however it currently lacks a way to prevent *speculative execution* since the previous mechanism to prevent this (explicit *barrier* placed on-chain) is no longer feasible due to design constraints imposed by the Hyperledger Fabric maintainers. We demonstrate how our verification system is useful for synchronizing peers and preventing speculative execution in FPC by using runtime verification as the *barrier* instead.

Contents

List of Figures

List of Tables

Acknowledgments

Firstly, I want to extend a huge thank you to my research advisor Raluca Ada Popa for taking a chance on me and bringing me into the RISELab security group; my academic experience has been greatly enriched from all of the different opportunities I've been able to experience as a result. Thank you also to Wenting Zheng and Rishabh Poddar who initially proposed the core project idea and dedicated their time and energy towards guiding us throughout the initial phases of the project; I look forward to continue working with you all over the next several years!

A huge thanks to Karen Tu for being an amazing research partner, not only for this report but also for several of the graduate classes we both took together; you were always incredibly understanding, and I could not have completed our various assignments, projects, and this report without you. I would also like to thank Ion Stoica for his feedback on an early iteration of this project as part of a class submission and for serving as a second reader for this report. Thank you also to Jeffrey Chen who assisted greatly with the original iteration of this project! I would also like to thank Mic Bowman and the Hyperledger FPC Team for their initial guidance on the problem setting and identifying where and how a verification integrity mechanism could be useful for FPC.

I also want to express immense gratitude to all of the lifelong friends I have made while at Berkeley; you have all made my time at Berkeley extremely memorable and will have a special place in my heart forever. To all of my other friends and loved ones beyond Berkeley who kept in touch even when I didn't and were understanding of my work schedule over the past several months, I truly appreciate your love and support.

Finally, I would not be at this stage in my academic career today without the lifelong support and sacrifices of my grandparents, parents – Ambika and Sanjay – and my little sister – Aashi – who have been cheering me on my entire life; all of this is for you.

Chapter 1

Introduction

When running complicated data analysis over large datasets, it is common practice to outsource the computation to a third party cloud computing provider. However, sometimes for legal reasons, data cannot be read by third parties, such as a hospital's patient data. Additionally, with increasing concern over data privacy and potentially malicious third party cloud computing providers, there is a need for a way to outsource data analysis without compromising data security. Hardware enclaves provide a trusted execution environment that provides data confidentiality and integrity. Only security sensitive code is placed into the enclave, greatly reducing the trusted computing base.

There are several tools that port entire existing applications into an enclave [5], but this is not always a good idea because many applications are not hardened for side channel attacks. In addition, placing an entire application within the enclave results in an unnecessarily large trusted computing base. Only security sensitive code needs to be placed within the enclave; all of the other code can be placed outside. Civet [49] and Glamdring [31] partition applications into trusted and untrusted code in order to reduce the trusted computing base. All code that accesses sensitive data or affects data flow is considered trusted code and placed into the enclave.

Secure enclaves have large potential to be used for outsourcing data analysis, which is often distributed for large datasets or complicated analysis. The existing partitioning solutions, Civet and Glamdring, are not targeted for distributed applications, and for distributed applications they would not optimally partition the code. Scheduling and communication code affect data flow, and Civet/Glamdring would place such code inside of an enclave. However, it is possible to place scheduling and communication code outside of the enclave as long as it can be verified that such code executed correctly, as done in VC3 [41] and Opaque [56]. This is possible because in distributed data analysis applications, a job is broken down into several tasks that are distributed across nodes. The piece of code that breaks down the job into smaller tasks is placed in the enclave, while the code that distributes the tasks and communicates encrypted data is placed outside. The problem is that existing distributed enclave-based applications have custom verification mechanisms which require significant manual effort to implement and are not easily transferable to other distributed

enclave-based applications.

Aside from distributed data analysis tasks, secure enclaves have also recently seen usage in blockchain and distributed ledger platforms for enabling the secure/ confidential execution of smart contracts containing private data. Fabric Private Chaincode (FPC) [9] is a proposal to extend the Hyperledger Fabric permissioned blockchain platform with secure enclaves by executing *chaincode* inside of enclaves such that the peer node on which the chaincode is executing cannot view function inputs and execution. While the original design of FPC included an additional trusted ledger enclave component that was used to provide proof of consensus around blockchain state values, the current design of FPC removes this component as it replicated a large portion of the peer validation logic inside of an enclave, resulting in redundant and unmaintainable code. Removing the trusted ledger component requires an alternate method for synchronizing peer nodes and preventing malicious peers from performing *speculative execution* to learn private information.

In this report, we present a verification library that aims to assist developers of distributed enclave-based applications verify the proper execution of code outside of the enclave. Given a partitioned codebase for a distributed enclave application, our library can be used to verify data flow integrity and enforce code invariants. Within each enclave function, the developer uses our library to create an `AuditLogEntry` each time that function is called. During program execution, based on the data flow rules and ECALL constraints specified by the developer, the audit logs are cross-checked with the rules to make sure that all data computation tasks were completed. We show in an example of integrating our library with a distributed data analysis application that the API calls result in about 3% of latency overhead, and we also demonstrate the usability of this verification system with Hyperledger FPC to serve as a `barrier` in order to maintain execution integrity and disallow speculative execution.

This report was partially written in collaboration with another Master's student, Karen Tu. All discussion of private smart contracts and extending Hyperledger Fabric Private Chaincode with this verification system is my own work. The sections that I wrote completely independently are listed below.

- 2.3: Smart Contracts and Blockchain,
- 2.4 Techniques for Smart Contract Privacy
- 3.4: Private Smart Contracts via Secure Enclaves
- 4: Hyperledger Fabric Private Chaincode (FPC)
- 8: Extending Hyperledger FPC
- 10.4: Hyperledger FPC

All other sections have been written collaboratively, aside from portions of the sections which have to do with smart contracts or Hyperledger FPC.

Chapter 2

Background

2.1 Secure Enclaves

In many modern cloud computing models and services, applications are often deployed in untrusted environments such as public clouds which are controlled by third-party providers. As the underlying infrastructure is unknown (i.e. OS and hypervisor) these environments and their hosts are untrusted. Trusted Execution Environments can help mitigate these threats as they support memory and execution isolation of code and data from the untrusted environment.

2.1.1 Intel SGX

Intel's Software Guard Extensions (SGX) [24] help to protect the confidentiality and integrity of application code and data (running on hardware that supports SGX) even in the presence of an attacker with control over all software (OS, hypervisor, and BIOS). SGX provides a trusted execution environment called an enclave. Enclave code and data reside in the enclave page cache (EPC) and are protected by an on-chip memory encryption engine which encrypts and decrypts cache lines in the EPC that are written to and fetched from memory. Only application code executing inside the enclave can access the EPC. While non-enclave code cannot access the EPC, enclave code is free to access memory outside the enclave.

Furthermore, the Intel SGX SDK provides a feature Remote Attestation which allows the client (the Challenger in Figure 1) to verify that their code has been set up properly in the enclave and that the host has not tampered with the enclave in any way. This way, the host cannot modify or otherwise modify the contents of enclaves without being detected by the client. Thus, Intel SGX protects against the general class of attacks known as Iago attacks that occur when a malicious OS exploits an application by subverting the assumptions of correct non-malicious OS behavior.

It is up to the enclave developer to define the interface between the trusted code (that goes in the enclave) and the untrusted code (outside the enclave). A call into the enclave is

known as an enclave entry call (ECALL) whereas a call from within the enclave to transfer execution control to outside the enclave is known as an outside call (OCALL). Both ECALLs and OCALLs induce performance overhead as the processor needs to marshal and unmarshal parameters and maintain SGX's security guarantees.

2.2 Enclave Partitioning

While it is possible to execute entire applications inside enclaves by adding system support in the form of a library OS, this is not entirely desirable. Placing all application code inside the enclave creates a large trusted computing base (TCB) which violates the principle of least privilege. To solve this issue, many enclave programs are partitioned where trusted and untrusted code is delineated such that only security sensitive functions and code are placed inside of the enclave. The degree to which the enclave is partitioned is up to the discretion of the enclave developer.

However, partitioned enclave programs have a greater amount of untrusted code. This leaves a larger attack surface for a malicious host to call the wrong ECALL, pass in incorrect parameters, replay old ECALLs, drop messages, etc. This is the problem that we aim to address - in partitioned enclave-based services, the enclave application needs some sort of verification mechanism in the trusted code in order to ensure that the untrusted host does not abuse the partitioning to tamper with or otherwise disrupt the application program's enclave execution. In other words, we want execution integrity for the application's untrusted code that the host runs. We aim to provide a general framework that can provide a verification mechanism for any such partitioned enclave program that requires minimal effort from the application developer to integrate.

2.3 Smart Contracts and Blockchain

First proposed by Szabo in 1994, a smart contract is "a set of promises, specified in digital form, including protocols within which the parties perform on these promises" [46]. Smart contracts today are primarily discussed in the context of blockchains/ distributed ledger platforms and are used to enforce various protocols upon the invocation of transactions on the underlying distributed ledger platform.

Some of the most common smart contract platforms today are Ethereum and Hyperledger Fabric. Ethereum is a public distributed ledger but also has support for creating private networks with restricted participation as well. Hyperledger Fabric is a permissioned network (restricts access to participation and chaincode functionality via access control list) with a consensus mechanism that relies on ordering as opposed to proof-of-work/ proof-of-stake as in Ethereum. These differ from several other well-known distributed ledger platforms such as Bitcoin which do not have support for the type of Turing-complete execution logic that smart contracts provide; instead Bitcoin features only a simple stack-based programming

language for managing payment transactions and does not feature Turing complete smart contract capabilities.

Smart contract state and execution is often replicated across several or all nodes in the distributed system for verification purposes. However, performing verification requires that contract state and user inputs are public, thus drastically limiting the range of applications that smart contracts can be utilized for. Thus, enabling *private smart contracts* is essential for expanding the utility of the technology.

2.4 Techniques for Smart Contract Privacy

There have been several efforts to bring confidentiality to smart contract execution atop public ledger systems, as this would enable many more use cases. Bitcoin is often touted as being anonymous, however it only provides pseudo-anonymity as transaction graph analysis can reveal link between transactions which can eventually also reveal a user's identity if linked to a known exchange/ wallet address. There is also no confidentiality for transaction amounts. Zerocash [6], a fork of Bitcoin, solves the problem of anonymity in payment transactions by not including origin, destination, or amount information in the transaction, and instead using zero-knowledge proofs to prove the correctness of a transaction. Monero/ CryptoNote [52] achieves something similar by using ring signatures to hide the sender, receive, and amount of a transaction. However, like Bitcoin, these platforms only seek to provide transaction-level anonymity and lack support for general purpose Turing-complete programming in the form of smart contracts.

Extending zero-knowledge proofs to enable private smart contracts has several challenges, including that multi-party computation is not supported with this design and that currently zero-knowledge protocols have high computational complexity for the *prover* and require a costly trusted setup.

Another strategy that has been explored by several works [8, 25] for enabling private smart contracts has been to privately execute smart contracts off-chain, meaning that peers/ nodes in the network do not directly execute the smart contract; rather, the smart contract logic is invoked by a single node (or some subset of nodes) and the resulting state transition is signed and submitted to the rest of the network.

A final strategy that has been explored is the use of trusted hardware in the form of trusted execution environments/ secure enclaves (e.g. Intel SGX) to execute smart contracts in a secure region of memory protected from the rest of the machine and attested to by a remote attestation process. Such an approach is much more efficient than computationally-complex cryptographic approaches such as using zero-knowledge proofs, but it is also more vulnerable in the case of malicious hardware providers. Section 3.4 discusses several related works in the area of achieving private smart contracts via secure enclaves.

Chapter 3

Related Work

3.1 Untrusted Cloud

CryptDB [39], MrCrypt [47], BlindSeer [37], Monomi [50], and [2] use cryptographic tools such as homomorphic encryption without any trusted hardware to provide data confidentiality on an untrusted cloud. Encrypting data alone is not enough for distributed data analysis applications which require complex functionalities beyond simple queries and thus need some way to ensure execution flow integrity for computations.

Virtual Ghost [17] uses compiler instrumentation and run-time checks to create a protected region of memory. Flicker [34], MUSHI [55] use TPMs (Trusted Platform Modules). The drawback of TPMs compared to TEEs (trusted execution environments) such as SGX is that TPMs provide many cryptographic tools, but do not allow developers to run their own code within the TPM. SeCage [32], InkTag [20], and Sego [27] rely on virtualization. Using virtualization techniques such as hypervisors typically result in large TCBs; using TEEs reduces the TCB size greatly.

3.2 Enclave Frameworks

Haven [5], Graphene-SGX [48], and Occlum [44] provide libOS's that run within enclaves, thus making it possible to put entire applications inside of an enclave, and thus have effectively no untrusted code (besides the code that creates the enclave and calls into it). The main problem with this approach is the resulting large trusted computing base and consequently a large attack surface. Another major issue with porting legacy applications into SGX is that SGX is vulnerable to side channel attacks [53, 11, 42, 19].

SCONE [4] and Ryoan [21] isolate containers and sandboxes, respectively, inside of an enclave. Similar to libOS's, this allows developers to run their unmodified applications within an enclave. Although the TCBs are certainly smaller than those of the libOS's, they are still unnecessarily large. The purpose of an enclave is to run security-sensitive code; all other code should be placed outside.

Panoply [28] enforces a strong integrity property for the inter-enclave interactions, ensuring that the execution of the application follows the legitimate control and data-flow even if the OS misbehaves. While Panoply puts all security sensitive code inside the TCB of the application, we aim to reduce the size of the TCB by bringing certain code outside the enclave by augmenting the application with an additional verification mechanism.

Civet [49] and Glamdring [31] automatically partition enclave applications (Java and C respectively) into trusted and untrusted components using annotations and code analysis. The problem is that for distributed applications, these frameworks would put all data-flow related code (such as a scheduler) into the enclave, which is not necessary.

3.3 Enclave Applications

EnclaveDB [40] is a database engine that places all sensitive data into enclaves. To maintain data integrity, it keeps a database log. ObliDB [18] is also an enclave database, but it uses oblivious algorithms to hide access patterns which incurs a large overhead. Visor [38] is a video analytics platform using a TEE across CPUs and GPUs, using Graphene [48] for certain video processing modules. There is limited partitioning in Visor; the GPU resource manager is untrusted, but most of the code is inside the TEE. Similar to Visor, the above applications are not focused on reducing the TCB so much of the code base is inside of the enclave.

3.3.1 Distributed Enclave Applications

Different distributed applications have common functionality that affect data flow such as schedulers and communication code, which can be placed outside of the enclave as long as the data flow is logged so it can be verified as correct.

VC3 [41] runs on Hadoop within SGX enclaves. Verification is performed by workers sending information about data inputs and outputs to a master node. Opaque [56] is a data analytics platform built on Spark SQL. Spark SQL's query plans are DAGs, where the edges represent data flow and the nodes each represent a computation task. In Opaque's design, verifying the dataflow, even though the job scheduler is considered untrusted, is done during runtime. Each worker node will only execute a task if it has received all required inputs. This is an improvement over VC3's design, which requires worker nodes to all communicate with a master node. Both VC3 and Opaque are manually partitioned and have custom verification logic. Currently, there is no simple way for developers of distributed enclave-based applications to implement dataflow integrity.

PySpark-SGX [29] is PySpark built on top of Scone [4] so that it can run in an SGX enclave. It is similar to Opaque as it uses core Spark components, but unlike Opaque, the scheduler is placed inside of the enclave. Another problem is that building on top of Scone includes Scone as part of the TCB.

3.4 Private Smart Contracts via Secure Enclaves

The primary technique we are concerned with for this paper is the use of trusted hardware in conjunction with smart contract platforms. Several works have explored such avenues of research.

In Hawk [26], on-chain privacy is achieved by encrypting state updates on the blockchain, using zero-knowledge proofs to enforce correct contract execution and relying on a third-party *manager* which may be implemented either with trusted hardware such as Intel SGX or replaced with a multi-party computation to execute the smart contract. ShadowEth [54] is built atop the public Ethereum blockchain and seeks to provide privacy for all three of smart contract code, contract execution, and internal contract state by using a combination of secure enclaves, encryption, and secure communication channels. Ekiden [16] presents a TEE-blockchain hybrid system that separates enclave-enabled *compute nodes* from *consensus nodes* which maintain the underlying blockchain and do not require trusted hardware.

Private Data Objects (PDOs) [8] is a solution that enables parties to privately run smart contracts off-chain in Intel SGX secure enclaves and then submit a transaction to the distributed ledger containing information about the new smart contract state for validation (signature is checked).

Finally, Hyperledger Fabric [1] is a permissioned, consortium blockchain platform for which there has been a recent proposal termed Fabric Private Chaincode (FPC) [9] to bring privacy to Hyperledger Fabric by running chaincode inside of Intel SGX enclaves. See Chapter 4 for a dedicated discussion of Hyperldger Fabric/ Fabric Private Chaincode.

Chapter 4

Hyperledger Fabric Private Chaincode (FPC)

4.1 Hyperledger Fabric

Hyperledger Fabric is a permissioned blockchain platform with support for smart contracts in the form of chaincode. A Fabric network is comprised of peers, clients, and an ordering service. Unlike Bitcoin, Ethereum, and several other public ledgers, Fabric does not use proof-of-work or proof-of-stake consensus; rather, consensus of transactions is achieved by the execute-order-validate architecture. Under this architecture, transaction proposals are made and then executed on some subset of peer nodes (endorsers), and then the transaction is sent for ordering to the *ordering service*. The ordering service will broadcast the transaction to all peers who then validate the state update by checking it against the chaincode endorsing policy and ensuring that the ordering does not invalidate prior transactions.

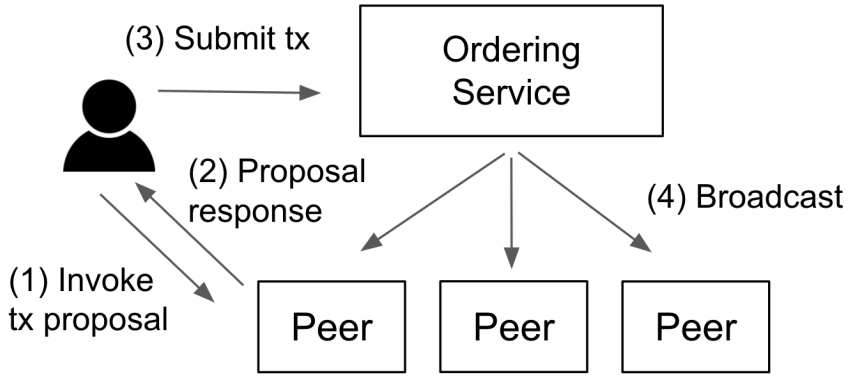


Figure 4.1: Hyperledger Fabric Architecture

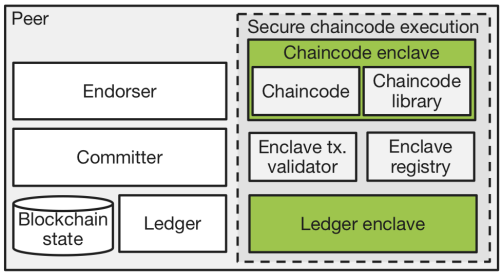


Figure 4.2: Initial FPC Architecture

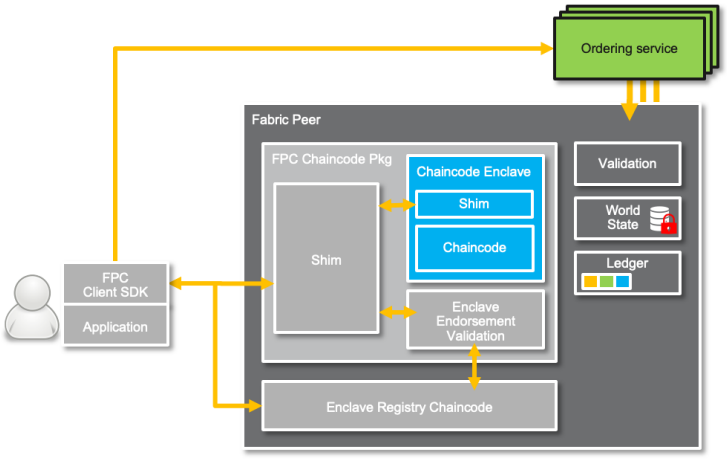


Figure 4.3: Updated FPC Architecture

4.2 Hyperledger FPC Design

Confidentiality in blockchain platforms is a trait that is often desired but not straightforward to achieve given the nature of blockchains which replicate data and computation across many nodes in the network. The lack of a confidential method to execute smart contract logic and manage data has precluded many use-cases from being implemented atop blockchains.

Hyperledger Fabric Private Chaincode (FPC) enables the execution of chaincodes using Intel SGX for Hyperledger Fabric by protecting the privacy of chaincode data and computation from potentially untrusted peers. The initial version of FPC as specified in [9] augments the base Hyperledger Fabric design by adding a *chaincode enclave* that executes a particular chaincode running inside SGX and a *trusted ledger enclave* that performs transaction validation and stores the ledger state in the form of hashes of each key-value pair in the ledger state. In the untrusted part of the peer, FPC adds an *enclave registry* that maintains a list of all the chaincode enclaves and an *enclave transaction validator* that validates transactions executed by a chaincode enclave. Figure 4.2 [22] depicts the augmented Hyperledger Fabric peer under this initial FPC design.

For a full in-depth explanation of the FPC design, view [9].

4.3 Speculative Execution

Fabric uses the *execute-order-validate* paradigm in which a peer executes a transaction before consensus on the order is reached. The initial execution of chaincode prior to ordering is *speculative* and does not affect the blockchain world state. Hence, the transaction can be

executed multiple times (provided that the peer performs state rollback) with different user-inputs provided. Such *speculative execution* may allow a malicious peer to glean confidential information about the application state.

One way to guard against speculative execution is by using *barriers*. A *barrier* is some piece of information committed to the blockchain world state indicating when a certain point in the code execution has been reached. Since we can make various chaincode functionality contingent on the presence of the barrier, this allows applications to prevent rollbacks across the barrier and simulate an order-execute design [9].

4.4 Issues with Initial FPC Design

The initial design of FPC included a trusted ledger enclave to be placed inside an enclave. However, the trusted ledger replicates a large portion of the peer validation logic, resulting in redundant and unmaintainable code according to feedback provided to the FPC development team. Figure 4.3 [23] displays the updated FPC architecture .

Without the trusted ledger enclave, there is no way for the chaincode to ascertain whether certain state has been committed to the blockchain ledger or if it is only present locally on the peer. Hence, preventing speculative execution via a barrier is not applicable in this scenario since the authenticity of the barrier cannot be checked.

The following section describes a particular example of an application running on the Hyperledger Fabric blockchain, how speculative execution may be prevented by the use of a barrier, and the challenges in achieving this as a result of the FPC design changes. The purpose of applying our verification system to FPC is to attempt to synchronize connections between the peers in a way that allows for detecting speculative execution by a malicious peer.

4.5 Sealed-Bid Auction

Consider a sealed-bid auction on a blockchain where bids are kept secret and winner revealed only after auction is closed (but before the auction is evaluated) and a *barrier* is placed on ledger. The auction chaincode can check that a barrier has been placed on-chain before evaluating the auction results, and refuse to do so if no barrier is present. However, without a trusted ledger enclave, the issue of speculative execution arises again since the existence of a barrier can be faked by a malicious peer and the chaincode enclave will have no way to verify whether or not it is legitimate. With a fake barrier, the peer could induce the chaincode enclave to evaluate the auction, learn the results, perform a state rollback to submit more bids, and then repeat this process until the peer ultimately learns the value of the current highest bid. Our proposed approach is to use the results of runtime-verification as a *barrier*; the verification takes place inside of a verification enclave and we check that extraneous/ malicious ECALLs were not made and that the appropriate

conditions for placing a barrier on-chain were met. Section 10.4 discusses the sealed-bid auction example in FPC with our verification system in more detail.

Chapter 5

Threat Model

We describe the model of secure enclaves that we consider, the capabilities of the adversary, and the attacks which are in/out-of-scope. We take inspiration from [45] in which the authors present a formalization of idealized enclave platforms, including a formal model of enclave programs and the adversary.

5.1 Abstract Enclave Model

We have designed our system using an abstract model of a hardware enclave that user applications can enter and exit during program execution. Our abstract model of a hardware enclave assumes the following enclave operations at minimum: `launch, destroy, enter, exit, attest`. Given such an enclave, we are able to apply our proposed run-time verification system to detect adversary actions which attempt to compromise the integrity of enclave application execution in the partitioned/ distributed setting.

In practice, hardware enclaves are susceptible to side channel attacks [53, 43, 13, 15] and software-based attacks [19, 51, 30]. Protecting against these vulnerabilities is beyond the scope of our threat model and are not considered as impacting our idealized abstract enclave model. Solutions to these attacks are complementary and are partially addressed in [14, 10, 35, 33].

This abstract enclave model may still fail to provide any security guarantees in the case of poorly-written enclave applications which lack confidentiality and may inadvertently leak data via network and memory accesses. For such applications, the developer must first secure any sources of inadvertent data leakage in order to utilize our verification system, as our system will not make an existing non-secure application (application logic leaks information about data) secure. The enclave application developer must implement data confidentiality by encrypting the outputs of ECALLs and obscuring lengths of output results by providing the appropriate padding for data.

The focus of our work is to provide integrity for partitioned enclave applications by making it easier to write verification logic; we do not provide any additional mechanisms to

make an existing non-secure application more confidential.

5.2 Adversary Actions

Formally, our scenario involves a client, an untrusted host (the adversary) that supports SGX, and an application running on this untrusted host. In our real world scenario, this untrusted host has the capability to both observe and tamper without being detected [2]. We define observation as the ability of the adversary to view any output as well as any memory access patterns via side channel attacks. We define tampering as being able to pause the enclave at any time to execute arbitrary instructions that modify the state of the enclave, the enclave's input, and launch or destroy enclaves. The problem we aim to solve is to reduce this real world scenario to our ideal world scenario where the adversary only has the ability to observe undetected - any attempts to tamper will be detected by our verification mechanism.

We make the assumption that the developers will write applications that protect against side channel attacks, as several hardware enclave platforms have known side channel attacks. Denial of service attacks are also out of scope, as an untrusted service provider can easily drop a client's messages and requests.

Chapter 6

System Overview

The overarching goal of this system is to reduce developer effort required for bringing inter-enclave execution verification to their applications. To achieve this, we make the following contributions:

- We model the enclave application execution flow as a directed acyclic graph (DAG).
- We augment the enclave TCB by writing an in-enclave library of verification primitives containing various data structures, cryptographic primitives, and communication code needed for enabling audit logging and verification.
- We provide a JSON ruleset configuration template to the client for indicating the expected execution flow.
- For a developers of applications with many enclave functions, we provide a script to automatically insert the necessary verification library API calls.

6.1 Architecture

We present the overall application architecture in Figure 6.1 for a distributed data analysis enclave-based application that uses our verification library and has been deployed into the cloud. In this section we, also describe all parties involved in a distributed application that uses our verification library.

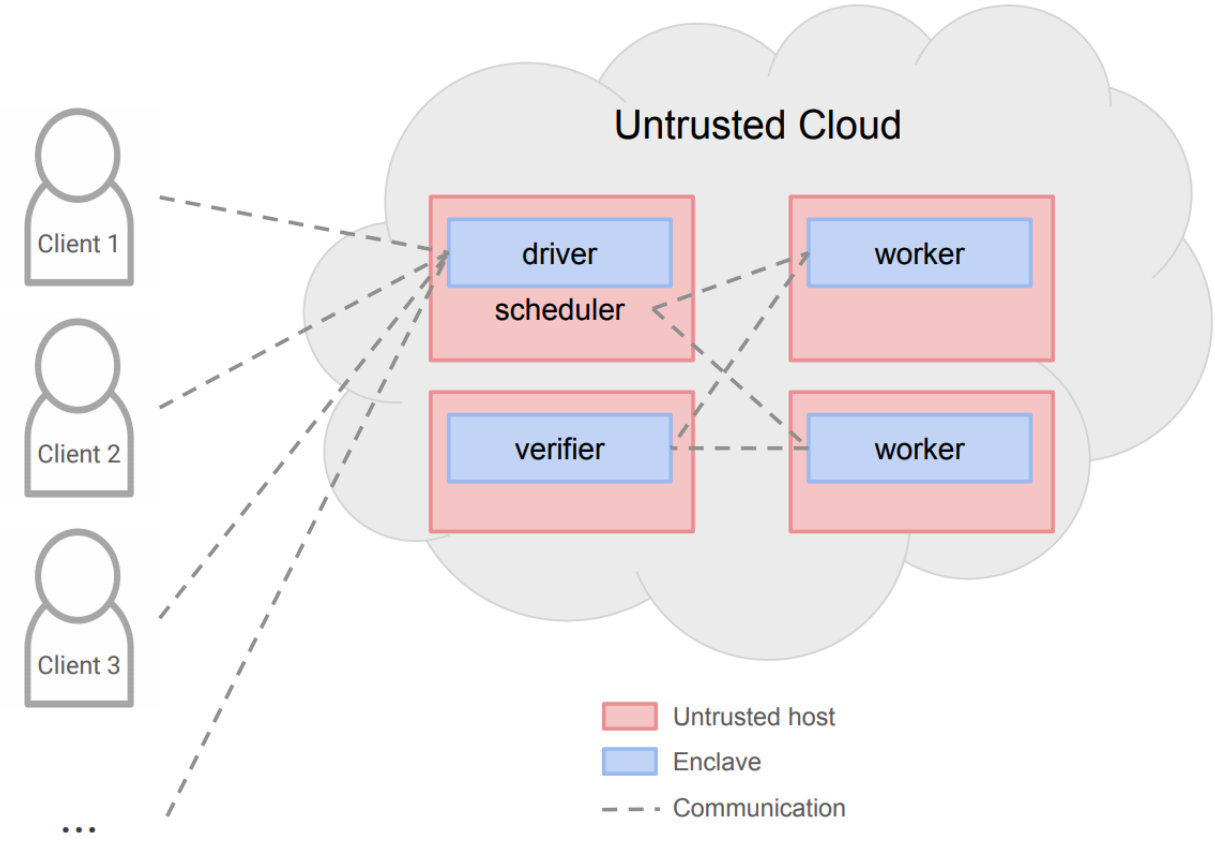


Figure 6.1: Distributed data analysis enclave-based application architecture after integrating our verification library and deployment.

6.1.1 Application Developer

The application developer is the one who either writes a distributed data analysis application from scratch, or finds an existing one to modify and run in an enclave. The developer is the party who will directly use our verification library. In Figure 6.1, the driver, scheduler, and workers are all written by the developer, while the verifier/ verification server is part of our verification library. The driver is the component of the application that receives client requests and splits up the request into data computation tasks that can be assigned to the workers. The scheduler distributes the workload of data computation tasks to the workers. A more detailed workflow for the developer is described in Section 7.2.

6.1.2 Client

The client directly uses the data analysis application created by the developer, and thus indirectly uses our verification library. The client is a trusted party; they are the ones who want to keep their data confidential.

6.1.3 Untrusted Cloud

The cloud is a third party service provider that the developer uses to run their distributed data analysis application. The cloud provides a cluster of virtual machines that support running Intel SGX enclaves. The untrusted host in Figure 6.1 corresponds to a single VM in the cloud, and each VM can have multiple enclaves.

Chapter 7

System Design

In this section, we explain the design details and decisions for our verification library. We describe the API, the workflow for how to use it, and how the execution flow is verified.

7.1 API

```
init_audit_log (...) // Initializes audit log entry; invoked at start of ECALL
log_input_data (...) // Computes hashes for data provided as input to ECALL
log_output_data(...) // Computes hashes for data outputted by ECALL
send_audit_log (...) // Sends audit log to verification server
```

Listing 7.1: The above API is made available to the developer.

The developer adds these API calls inside of the function definitions of the ECALLs that need to be verified. The two API calls `init_audit_log` and `log_input_data` are called in the beginning of an ECALL, while `log_output_data` and `send_audit_log` are called at the end. For details about API return values and parameters, a fully commented code snippet of the API can be found in the appendix A.1. A valid concern to have is that we are increasing the trusted computing base, by making the API calls within the ECALL functions definitions. The alternative solution is to modify the code function where the ECALL is called. However, ECALLs are made in untrusted code, and the verification code must be running inside of an enclave. Therefore, we either have to turn the API calls into additional ECALLs, or have the API calls made within the ECALL itself. Because of the overhead of marshaling inputs/outputs as well as context switching every time an ECALL is made, we decided to have the API calls made within the ECALL function definitions.

7.2 Workflow

Below we describe the general expected workflow for a developer who wishes to use our verification library. We assume that the developer handles data confidentiality, but needs our library for execution flow integrity.

1. The developer writes a distributed data analysis application with a scheduler and driver as described in 6.1.1.

2. The developer partitions their application into trusted and untrusted code; only security sensitive code is trusted code.

3. The developer integrates our verification library by:

   a) Importing our verification library in the files with ECALL (enclave function) definitions for the ECALLs that affect execution flow integrity.

   b) Extending data structures that contain sensitive data and are the types of parameters passed into ECALLs that affect execution flow integrity with the Iterator-Interface class. This allows our verification library to iterate through the chunk of data elements passed into an ECALL, one element at a time.

   c) Adding API calls to the ECALL function definitions; this can be done manually, or with the script we provide.

   d) Specifying a ruleset JSON file for which enclave functions correspond to specific data operations that the client can submit and how the inputs and outputs enclave functions related to each other.

4. Developer deploys application into the cloud to be used by clients.

7.3 Execution Flow Verification

We first discuss several design propositions for verifying correct execution flow to show how we came to our current verification design. Then we present our runtime verification process.

7.3.1 Design Tradeoffs

Keeping in mind our API as specified in 7.1, an audit log is created for every ECALL. In order to verify execution flow, we only need to make sure that no data was dropped or added. Enclaves are trusted, so we can be sure that the audit logs are trusted. The untrusted hosts that the enclaves are running on could refuse to send the audit logs, but this equivalent to a DoS attack which is outside of our threat model.

Design 1

The simple strawman solution is that for each ECALL, store all of the data that is processed in the audit log. While this solution is simple and easy to understand, it is not practical. Our verification library is targeting distributed data analysis applications, which typically process large amounts of data. Therefore, logging the data itself is a huge waste of

memory. In enclaves this is an even bigger problem as enclaves have limited memory; this strategy would cause massive overheads from page swaps.

Design 2

A more memory efficient solution is to store some sort of identifier for the chunk of data that the ECALL is processing. We can either require that the developer assigns an identifier to every data element, or we can include a hash function in our library to create an identifier for each data element.

The problem with this solution is that it makes the simplifying assumption that the output of one ECALL will directly be used as the input of another ECALL; this is not a safe assumption to make. In a collection of encrypted data elements, it is reasonable to expect that data operations will be applied to the individual data elements. For example, in a collection of encrypted row data (e.g. an encrypted data frame in Spark), encryption occurs at the granularity of individual rows as opposed to on the entire data frame/ table. Any transformations which modify the underlying plain-text data must take place inside of an enclave and not in the untrusted code. However, there are modifications to the data in untrusted code that are allowed such as combining and splitting up encrypted blocks. As a concrete example, in Opaque [56], `collect` is used in the untrusted code to aggregate chunks of encrypted data.

One way we can verify execution flow despite modifications of data in the untrusted code, is if we assume that each enclave knows which rows it will receive ahead of time. Considering how a typical distributed application scheduler works and how a specific task is not meant to be tied to a particular worker node, this is an unreasonable assumption. Therefore, we must be able to perform integrity checking over the smallest unit of data present in the application (e.g. rows, arbitrary bytes, etc). This means that the developer to provide some way for our verification library to iterate over the smallest unit of data; each ECALL processes multiple units of data.

Post-Verification vs. Runtime Verification

In early design discussions, we proposed a post-verification approach in which a list of audit log entries from within each enclave would be serialized and sent to a verifier (which could be the client's computer, a worker enclave, or a dedicated verifier enclave) for post-verification after the execution has finished for a client's data computation. We ultimately shifted away from a post-verification approach after identifying the following constraints:

- Limited Enclave Memory: Enclaves have limited memory and the initial post-verification approach introduced an additional enclave context in which the audit logs are stored.
- Missing Early-Termination: With a post-verification approach, program execution must finish before audit logs can be aggregated and checked for discrepancies. This prevents early-termination and potentially results in wasted compute resources.

We shifted to a runtime verification approach, knowing that it would likely result in additional latency at the end of each round and communication overhead, but with better memory and resource usage.

7.3.2 Runtime Verification

AuditLogEntry

The primary data structure that we use in our library is an `AuditLogEntry`. We use hashing instead of having IDs because it is simpler than requiring developers to assign an ID to each data element. If the developer's application does not already contain code logic with data IDs, it is not trivial to implement. If the developer's application *does* contain data ID code logic, different developers may have different data types for their data IDs, making it difficult to implement a generalized verification library.

```
struct AuditLogEntry {
    int ECALL_id;
    uint64_t** input_data_hashes;
    uint64_t** input_supp_data_hashes;
    uint64_t** output_data_hashes;
    uint64_t** output_supp_data_hashes;
}
```

The `AuditLogEntry` struct specfied above contains a field for `ECALL_id` and several fields containing the hashes of the various input and output data associated with the ECALL. These hash fields are used by the verifier node during run-time verification to check whether it correlates with the developer specified ruleset.

Ruleset

The ruleset is a developer specified JSON which contains the ECALL metadata, such as ECALL to ECALL ID mappings, which parameters of each ECALL are data sources to be verified, the type of data source, and the rules. The rules are specified for each operation; an operation is a data computation that the client can make in the application. For example, join and sort would be two different operations. For a complete example of a ruleset JSON file, refer to the appendix B.1.

A rule consists of an equal sign (=) where on each side, the following format is used to represent how the data from one ECALL in a specific round should be combined:

`[single | union | each] ECALL_{ecall ID} [input | supp_input | output | supp_output]_{parameter index} round_{round number}.`

The first element (`single`, `union`, or `each`) represents how many times the ECALL is made in that round. `single` means that the ECALL is made once. `union` means that the ECALL is made multiple times in that round, and the data specified by the third element

(input, supp_input, etc.) should be unioned together. `each` means that the ECALL is made multiple times, but across all ECALLs in that round, the data specified by the third element is the same. The second element specifies the ECALL ID (the ecall function name to ID mapping is also specified in the ruleset file). The third element specifies which data type is being checked, as well as it's index in the ECALL's parameter list. The final element specifies the round number that the ECALL is made.

The following list shows some example rules.

- `single ECALL_0 supp_output_0 round_2 = single ECALL_3 input_4 round_3`
- `union ECALL_0 output_1 round_0 = single ECALL_1 input_4 round_1`
- `union ECALL_0 output_1 round_2 = each ECALL_3 input_4 round_5`
- `union ECALL_0 supp_output_1 round_2 = union ECALL_3 input_4 round_5`

We define a round as a set of ECALLs within the execution flow for a particular data operation which have no dependencies amongst each other. An example of an ECALL execution flow split into rounds is shown in Figure 7.1. Because of the way we define a round, we cannot support programs with cyclical ECALL dependencies. We can support any program with an ECALL execution flow that can be formulated as a DAG (this includes MapReduce style programs). This is a reasonable constraint because well known distributed data-analysis applications such as Hadoop and Spark have execution flows are DAGs.

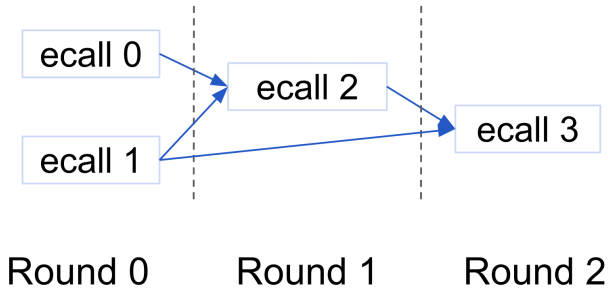


Figure 7.1: Example of ECALL execution flow split into rounds.

Verification Process

One enclave is responsible for verifying the audit logs after each round. This enclave runs a verification server to receive audit logs from all of the worker processes running in other enclaves. The first thing that the verification enclave receives is the input rows into round 0 from the trusted driver in the application. Upon receiving an `AuditLogEntry`, the verification server stores it until all audit logs for the current round are received. The verifier

knows that a round has completed when it has received the correct number of audit logs. Besides round 0, where the input data elements are provided by the trusted driver, the verifier can use the ruleset to calculate the number of audit logs to expect and thus know when a round ends to start verification. To perform verification at the end of a round, the verifier enclave obtains the relevant rules for the ECALLs in the current round from the developer specified ruleset and cross-checks the relevant audit logs with the rules.

The runtime-verification serves as an implicit barrier in the code; in other words, the verification server will not receive any audit logs from a subsequent round until it has finished processing all audit logs from the current round and has notified the relevant enclaves that the audit log has been verified. This is because if verification fails at any round, then the verification server notifies all of the enclaves.

The verification server is stateful and can retain audit logs from prior rounds in order to corroborate them with the current round's logs. Prior round state is maintained depending on the specification provided in the ruleset logic. For example, in Figure 7.1 ECALL 3 in round 2 needs the results of ECALL 1 from round 0, so the verification node must keep the relevant output of ECALL 1 until the end of round 2.

7.4 Optimizations

7.4.1 Automatic API Integration

We provide a script to make it even easier to integrate our API; it can be tedious to manually create the list of pointers to the data sources, and to manually specify the counts of each type of data source. If each ECALL has the same parameters, then the same API calls are made within each ECALL function definition. For example, in Opaque, all of the ECALLs have arguments `input_rows` and `output_rows` as input and output data sources, respectively. The same code is copied to the beginning and end of each ECALL, and the only difference between the ECALLs are the supplementary data sources, which can be specified in the ruleset JSON file. The script automatically adds in the API calls at the beginning and end of the ECALLs based on the data source metadata provided in the ruleset JSON file.

Chapter 8

Extending Hyperledger FPC

Our motivation for extending FPC with our verification system is to overcome the issue of the chaincode enclave being unable to trust that consensus has been achieved around any barriers placed on the blockchain. In order to extend FPC with our verification system, we make the following modifications:

1. Add audit log generation code to the single `ecall_cc_invoke` ECALL
2. Specify ruleset for a sample FPC chaincode
3. Port the verification server logic to Fabric chaincode

There is a single top-level ECALL present in FPC, `ecall_cc_invoke(...)`, which is a wrapper ECALL used to run all chaincode functionality within enclaves. The `ecall_cc_invoke(...)` function takes in an `t_shim_ctx_t ctx` struct as argument which is used to extract the wrapped chaincode function name and its parameter list. Prior to runtime, the chaincode developer is required to implement an `invoke()` function in their chaincode that gets called by the top-level `ecall_cc_invoke(...)` ECALL. Unlike in Opaque, instead of adding the audit log generation code to the top-level ECALL, we instead add it to the developer-specified implementation of the `invoke` function call for the specific chaincode. As a result, we do not attempt to use our automatic ECALL transfer script. It is added in almost the same way as is done for Opaque (see Appendix section A.3).

The ruleset that is specified for any sample application seeking to enforce verifiable barriers in code needs to specify the dependencies between chaincode function calls in order to determine whether the appropriate condition has been met in order to place a *barrier* (the result of performing runtime verification) on-chain. This can include information about ECALL ordering and specify restrictions on certain sequences of ECALLs being invoked. An example of what such a ruleset may contain for a sample FPC application is discussed in Section 10.4. The current ruleset format primarily deals with data flow by tracking inputs and outputs of ECALLs. For verifying barriers in FPC, we are more concerned with *which* ECALLs were invoked during a particular round. We adjust the ruleset specification slightly to account for this requirement.

In order to make minimal changes to Fabric, we implement the verification server as a peer running chaincode that contains the verification logic. We also use the blockchain as the communication layer rather than using out-of-band communication, such as a separate SSL connection originating and terminating within enclaves (for Opaque, we do use a separate SSL connection). Once audit logs are generated after an ECALL invocation, a *chaincode-to-chaincode* invocation is made by the application chaincode to the verification chaincode. The results of the verification for a given round will then be published to the blockchain along with a signature and freshness values. Since we are using the underlying blockchain for communication, we would typically need proof that the `send_audit_log` transaction and corresponding state update was actually committed to the blockchain. However, without the trusted ledger component (which had to be removed from the FPC design due to code redundancy and maintainability concerns), the chaincode enclave cannot get proof that the audit log state was actually committed to the blockchain. However, the chaincode enclave can still access the ledger and world state via the *shim* library present, and by enforcing that the verification chaincode includes a signature over its verification results, the peer cannot falsify verification results. This will require adding signature-checking capabilities to the application chaincode. Note that the result of the runtime verification serves as the *barrier* which was specified in Section 4.3.

Chapter 9

Implementation

9.1 Core System

Our verification system has minimal dependencies and is comprised of:

1. The core in-enclave verification library which is implemented in C++. We utilize xxHash [12], a fast (non-cryptographic) hash algorithm in order to efficiently compute unique identifiers for individual pieces of data passed to an ECALL to be processed by our audit log generation code an inserted into the `AuditLogEntry`.

2. A verification server which accepts incoming audit logs from all enclaves that is also implemented in C++ in order to be able to easily run inside of an enclave. The developer specified rulesets (which are used for performing run-time verification) are registered with the verification server and are specified as JSON objects which contain information about which ECALLs are associated with a particular program execution and what logic to apply to check whether an audit log is valid.

3. A Python script which utilizes the Python bindings for `libclang` (a library that simplifies analyzing C/C++/ObjC code) in order to parse C++ code to output a transformed EDL file and ECALL implementation file. This script implements the optional optimizations; developers can either use this script or manually change the EDL file and ECALL implementation file.

Communication between the executing enclaves and the verification server occurs at the end of each round via an SSL connection which begins and terminates within enclaves. This was achieved by modifying the `attested_tls` example [36] provided as part of the OpenEnclave SDK. MbedTLS [3] is used for all cryptographic prerequisites needed for establishing the SSL connection. We use the `boost::serialization` library [7] to serialize the `AuditLogEntry` data structure prior to sending it over the network.

For the Hyperledger Fabric setting, rather than using a separate SSL connection, we rely on the underlying blockchain communication mechanism to transmit audit logs by invoking

chaincode functions and storing the logs in the world state for various chaincodes and peers to access rather than explicitly sending audit logs between peers. This allows us to minimize the changes made to Hyperledger Fabric and the to peer nodes participating in the system.

The OpenEnclave SDK is also used for remote attestation, generating trusted-untrusted enclave interface from EDL file via oeedger8r tool, and other tasks typically associated with operating enclaves, but these should be handled separately by the developer writing the enclave application. As a result, the entire SDK is within the TCB of the enclaves, but our system does not directly utilize much of the provided SDK functionality.

Our core verification library is under 500 lines of code. Adding support for SSL connections adds about 600 lines each for the worker enclave and the verification server enclave. Our codebase is open source and available at: https://github.com/saharshagrawal/verified-enclaves

Chapter 10

Evaluation

For all evaluation we used SGX-enabled VMs on the Azure Confidential Computing cloud with plan 18_04-lts-gen2 running Ubuntu 18.04 each with 2 vCPUs and 8 GiB of RAM. We first evaluate the impact of our library on the TCB. Then we discuss the performance of the verification and communication in our library. Finally, we evaluate several API calls from our library on a single application, Opaque. We only tested our library on Opaque because secure enclaves are yet to become widely adopted, so there are a limited number of distributed data enclave-based applications.

10.1 Library Size

One primary goal of our verification library is to help developers reduce their TCB by moving code sections such as the scheduler and communication outside of the enclave. The table 10.1 contains the lines of code for different files in our library. We show that our verification library adds very little to the TCB; there are about 500 total lines of code, which is small enough to easily audit to make sure it is not malicious. The lines of code for `ecall_init_verifier` are an estimate, because integrating this ECALL includes modifying the interface code between the application and the enclave; the lines of code needed to do this depend on the application implementation. The estimate is based on the lines of code needed to modify Opaque.

File/Function Name	Description	Lines of Code
Audit.cpp	audit log API	180
Verifier.cpp	code run by the verifier enclave	270
IteratorInterface.cpp	class that data sources must inherit to be compatible with our library	7
ecall_init_verifier	send the number of data elements expected as input for round 0 to the verifier	≈25
Total		≈ 482

Table 10.1: Lines of code across verification library files.

10.2 Communication

In order for worker enclaves to send audit log entries to the verification server enclave, we must establish a secure communication channel. As mentioned in Chapter 7, an SSL connection is established between each worker enclave and the verification server by adapting the in-enclave client-server code provided in the `attested_tls` sample included as part of the OpenEnclave SDK.

Currently, we launch a new SSL client every time an audit log needs to be sent from a particular enclave and MbedTLS is used to configure the connection. A potential optimization here may be to use the same SSL connection within an enclave to send multiple audit logs rather than creating a new connection each time. We present timing data for the communication overhead involved in a specific ECALL in Opaque in the subsequent section.

10.3 Opaque

We tested the `init_audit_log`, `log_input_data`, and `log_output_data` API calls of our verification library on the latest released version of Opaque; this version of Opaque does not have any execution flow integrity implemented. The following results were obtained by adding audit log generation functionality to the `non_oblivious_sort_merge_join` ECALL and then running query 13 of the TPC-H benchmarking test suite which tests `left outer join` (which is implemented using `non_oblivious_sort_merge_join` in Opaque). The `non_oblivious_sort_merge_join` processes 165,000 total rows as input partitioned across 2 vCPUs/ enclaves where each enclave processes approximately half of the total number of input rows.

Below are the modifications we make to Opaque to integrate our library:

- We extended the data source data structure in Opaque, class `RowReader` with the IteratorInterface class. The lines of code added are minimal; a mere five lines of code were added to `FlatbuffersReaders.h`, where `RowReader` is defined.
- We defined a ruleset JSON file with 25 lines
- We added the API calls to the ECALL definition in `NonObliviousSortMergeJoin.cpp`, which only took 15 lines of code; the code sample can be found in A.3. Opaque is an example of an application where the API calls have to be manually inserted. We could not use our script as described in Section 7.4.1 to automatically modify the ECALL definition, because our script assumes that pointers to the data sources' data structures are passed in as parameters. However, in Opaque, the data source' data structure is initialized within the ECALL from two of the ECALL parameters.

The latency incurred by our API calls is shown in the following table (since our node has 2 vCPUs, the computation is distributed across 2 enclaves where each enclave processes approximately 80,000 table rows; we present only the max timings measured for each portion):

	Overhead (ms)
Log Input Data	12
Log Output Data	27
Ruleset Processing	0.12
Log Transmission	450
Verification	424
Total	913.12

Table 10.2: Evaluation of our library's performance using Query 13 from the TPC-H benchmarking test suite in Opaque

The Log Transmission timing value in table 8.2 is computed by serializing and sending a single audit log to the verification server running on the same physical node. This is a reasonable communication pattern, as it is entirely feasible for the verification server enclave to be running on the same physical node that worker enclaves are running on. Currently, we do not measure the communication overhead of transmitting audit logs across physical nodes since differing network conditions can result in high variability in timing measurements.

The API call for initializing the audit log is not included because it has negligible overhead. Without our verification library, this test suite takes about 29 seconds on average to complete. Therefore, our library adds roughly 3% of overhead.

The overhead from ruleset processing is minimal, as it does not take much to initialize variables and parsing through the JSON file. It is even faster because the `non_oblivious_sort_merge_join` ECALL does not have any explicit rules in the ruleset JSON file. Most of Opaque's operations that a client can use only have one round and one ECALL. Such operations only have the implicit rule that the initial data values sent by the trusted driver code to the verifier matches the input data to that one ECALL. ON the other hand, verification has the biggest overhead; this makes sense, as it requires iterating over every data element. The only step for verification of operation `non_oblivious_sort_merge_join` was to check that the initial data values match the input data to the ECALL. `non_oblivious_sort_merge_join` is an operation that has one round and one ECALL; only the implicit rule (as described in the previous paragraph) needed to be checked. The timing of checking this implicit rule is equivalent to checking any rule where one side of the equality has a non-`single` data source. For example, checking that the rule (`union` of the output of `ECALL_1` in round 0 is equal to the `union` of the input of `ECALL_2` in round 1) would take the same amount of time as the implicit rule, assuming that for both rules n data elements are iterated over to be cross checked.

10.4 Hyperledger FPC

The sample application that is tested with Hyperledger Fabric Private Chaincode is the sealed-bid auction which was briefly discussed in 4.5. Recall that the sealed-bid auction results are only evaluated once all bids are placed and the auction is closed, thus placing a *barrier* (some piece of information which indicates that the auction has been closed) on-chain to prevent *speculative execution* from occurring. The sealed-bid auction chaincode supports the following functions: `init, create, submit, close, eval`. For evaluating the verification system with FPC, we want to prevent the scenario where a bidder invokes the `submit` function followed by the `close` and `eval` functions, learns the results of the auction, and then colludes with the peer to perform a local state rollback, and then repeats the same ECALLs with a different bid value for the `submit` call multiple times in order to learn what the previous highest bid value is.

Removing the trusted ledger component from FPC makes it difficult to verify the existence of the barrier on-chain, and thus we evaluate our modified FPC design with runtime verification to determine if it can provide similar functionality as the trusted ledger component and barrier for this use-case.

Much of the process of integrating our library with Hyperldger FPC is similar to the proces for Opaque. Below are the modifications we make to Hyperledger FPC to evaluate our library:

1. We write a `FPCReader` class which is overloaded and able to accept a `shim_ctx_ptr_t` value or a `uint8*` pointer as input. This is used to parse the inputs and outputs to the `invoke` ECALL for a particular chaincode. Appendix A.3 displays a code snippet containing audit log generation code which uses the `FPCReader` class.

2. We define a minimal ruleset specification for the sealed-bid auction sample containing rules which instruct the verification chaincode to remember the set of ECALLs that have been invoked by a particular bidder thus far and to vail to verify if an invalid sequence of ECALLs is detected from a particular bidder (e.g. `submit` followed by `eval` followed by `submit`).

3. We port the in-enclave verification server from the Opaque evaluation to Fabric chaincode by adding a following the sample auction chaincode example and implement the `invoke` function call. The verification chaincode additionally has a `publish_results` function which adds the verification results and a signature over the results to the world state.

Since Hyperledger FPC is not designed for the type of large-scale data analysis workloads that Opaque is meant for, performing timing analysis on individual blockchain operations does not yield any significant results. Instead, we discuss the number of lines that needed to be added to Hyperledger FPC to integrate with the verification system and also repeat experiments multiple times in order to see the effects of verification.

We evaluate the auction sample in FPC with two peers in a permissioned network with both the auction chaincode installed as well as the verification chaincode installed on each peer. This is needed in order to perform chaincode-to-chaincode invocations. We also set the chaincode endorsement policy for the verification chaincode to only require a single endorser, which can just be the peer that the code executes on. Since any world state updates made by the verification chaincode will contain a unforgeable signature, unless the peer performs a denial-of-service (which is outside of the threat model) and refuses to forward the verifier's state update transaction, the verification result (which serves as a barrier) will become visible to all other chaincode enclaves.

The verification logic (which was 270 LoC as enclave code in Opaque) contains 373 LoC in FPC since we must additionally implement the `invoke(...)` function call and add code to perform blockchain state updates. We augment the auction chaincode ECALL implementation file in FPC by adding 20 LoC of audit log generation code.

Upon receiving audit logs (encrypted and published to world state by auction chaincode enclave), the verification chaincode runs, references the specified ruleset and successfully checks that no bidder called an invalid sequence of ECALLs (specifically, `submit` was not called again after `close` and `eval` without the auction being reinitialized first).

Below we present the latency incurred by augmenting FPC with verification logic. We test by considering four scenarios, in each of which a different order and set of ECALLs are performed. For example, test scenario #3 initializes and creates an auction, submits `num_rounds` number of unique bids, closes the auction, and evaluates the auction. We modify

the four tests slightly by repeating each sequence 10 times in order to be able to more easily detect the effects of the latency introduced by the verification library. The full list of test scenarios and the associated set of operations can be viewed at https://github.com/hyperledger/fabric-private-chaincode/blob/main/integration/auction_test.sh.

Scenario	FPC Timing (s)	FPC Timing w/ Verification (s)	Time Delta (s)
1	13.21	15.36	2.15
2	16.12	20.84	4.72
3	29.35	34.21	4.86
4	15.77	19.59	3.82
Average Latency			3.89

Table 10.3: Evaluation of Hyperledger FPC augmented with verification system; we test four scenarios, each of which was repeated 10 times.

The average latency found is 3.89 seconds for 10 invocations of each scenario. We can observe that the latency introduced is proportional to the number of ECALLs made in each scenario. For instance, scenarios 1 and 4 have the fewest number of ECALLs and, as a result they have a smaller latency introduced. For each ECALL in the system, the additional latency introduced is approximately the same since very small amounts of data are being processed in each audit log generation code block. As a result, only the number of ECALLs invoked has an effect here. The more fine-grained breakdown of individual verification steps is omitted here, as it is very similar to the one shown in the Opaque evaluation section above.

The minimal lines of code added and the reasonable overhead (typically around 15%) show that our verification system is able to integrate with FPC and provide basic barrier functionality without the need for a trusted ledger component.

Chapter 11

Limitations & Future Work

11.1 Limitations

The two key limitations of our approach are (1) the latency introduced at the end of each round due to runtime-verification and (2) the need for a manual ruleset specification.

The current system design has additional latency in each round proportional to the number of ECALLs executed during that round and the amount of data processed by each ECALL in addition to any constant network overhead terms. This is due to implicit barrier induced by runtime verification which makes the code *blocking* until a round has been verified.

Requiring developers to manually specify any sort of rules is susceptible to unintentional errors and can be time-consuming. In some cases, specifying such a ruleset for a particular class of applications may even be impossible.

Additionally, we can only support C/C++ applications, or applications such as Opaque that import native C function definitions. This is a limitation of using Intel SGX enclaves; it is not possible to run other languages in the enclaves, unless we utilize something similar to a libOS.

The bulk of the latency overhead comes from the verification step, because the verifier iterate through all data elements passed as inputs and outputs to the ECALLs. This is especially bad for large datasets, as the underlying data analysis application already iterates over all of the data to perform computations.

One of the goals of this project is to make it easier for the enclave application developer to build applications with support for execution integrity for partitioned and distributed applications. However, the requirement of a manual ruleset specification hinders this goal.

11.2 Future Work

In an attempt to address the latency concerns, we can adjust the *round verification frequency* so that instead of performing verification every round, verification is only performed every k number of rounds. This would require informing both the worker enclaves and the

verification server about the modified frequency so that it knows to retain logs for all those rounds. This may not be possible if there is limited memory on the verification server (since it is also running inside an enclave). In the extreme case where granularity is adjusted to the maximum number of rounds, this approach would begin to resemble a post-verification approach whose limitations were described in 7.3.1. Another way to address the latency overhead is to target the problem of iterating through all of the elements during verification. The overhead could be significantly reduced if we expose the logging for one element at a time in an API call to avoid double iteration over the data elements.

In our evaluation of Opaque, we run computation and verification on a single physical node. Opaque supports distributing computation across many physical nodes as it is built atop Spark SQL and Spark supports several cluster managers (e.g. Mesos, Kubernetes). Such cluster managers typically provide some form of fault tolerance, but we must evaluate further the usage of secure enclaves with such cluster managers and determine what modifications may be needed to our verification system to support non-manual node management.

As mentioned in 7.3.2, our library currently supports a limited number of types of rules. In the future, we aim to support rules that capture more complicated data relationships between ECALL inputs and outputs. In the evaluation of the auction chaincode in Hyperledger FPC, we construct a basic ruleset which instructs the verifier to retain state regarding which ECALLs were made and to fail to verify if an inappropriate sequence of ECALLs was detected. Enhancing the verifier with support for performing more complex analysis on ECALLs made and the relationships between ECALLs themselves will be of value in supporting a broader range of applications. Additionally, we may also want to support a distributed verification mechanism for load balancing purposes.

We apply our verification system to Hyperledger FPC, but other distributed ledger platforms could benefit with such a system as well if they are extended to support secure enclaves. For the purposes of this report, we only evaluate the auction chaincode sample; evaluating applications with more execution paths may require adding support for discriminating between different types of barriers placed on-chain.

Another direction for future work is to integrate our library with more distributed data analysis enclave-based applications beyond Opaque to make sure that the library generalizes well to work with a wide variety of applications. Even while integrating our library with Opaque, we realized that parts of our initial design were flawed. For example, we originally assumed that the inputs to ECALLs are the data sources, but for Opaque this is not true - the data sources are initialized inside of the ECALL based on the ECALL parameters.

Chapter 12

Conclusion

In this report, we present a runtime verfication system for partitioned and distributed secure enclave applications in the form of an in-enclave verification library in order to save time and effort for developers who otherwise would need to write custom execution flow integrity code logic for each new enclave application. We also provide optimizations to automate parts of the workflow to further decrease the developer effort required. We integrate our library into an existing distributed, encrypted data analytics platform (Opaque) to show that it not only has a reasonable code footprint but also a low latency overhead. We additionally discuss the value of performing confidential computation atop blockchain/ smart contract platforms to enable a wider range of applications than are presently possible and demonstrate how to extend one such platform (Hyperledger FPC) with our runtime verification system to synchronize code execution between peer nodes and prevent *speculative execution* from occurring. We show that our library is practical to use and has modest overhead while providing flexibility and convenience for the developer.

Bibliography

[1] Elli Androulaki et al. "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains". In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys '18. Porto, Portugal: Association for Computing Machinery, 2018. ISBN: 9781450355841. DOI: 10.1145/3190508.3190538. URL: https://doi.org/10.1145/3190508.3190538.

[2] Panagiotis Antonopoulos et al. "Azure SQL Database Always Encrypted". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 1511–1525. ISBN: 9781450367356. DOI: 10.1145/3318464.3386141. URL: https://doi.org/10.1145/3318464.3386141.

[3] ARMmbed. *mbedtls*. https://github.com/ARMmbed/mbedtls. 2021.

[4] Sergei Arnautov et al. "SCONE: Secure Linux Containers with Intel SGX". In: *OSDI*. 2016.

[5] Andrew Baumann, Marcus Peinado, and Galen Hunt. "Shielding Applications from an Untrusted Cloud with Haven". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 267–283. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann.

[6] Eli Ben Sasson et al. "Zerocash: Decentralized Anonymous Payments from Bitcoin". In: *2014 IEEE Symposium on Security and Privacy*. 2014, pp. 459–474. DOI: 10.1109/SP.2014.36.

[7] Boost. *Boost Serialization*. https://www.boost.org/doc/libs/1_75_0/libs/serialization/doc/index.html.

[8] Mic Bowman et al. *Private Data Objects: an Overview*. 2018. arXiv: 1807.05686 [cs.CR].

[9] Marcus Brandenburger et al. *Blockchain and Trusted Computing: Problems, Pitfalls, and a Solution for Hyperledger Fabric*. 2018. arXiv: 1805.08541 [cs.DC].

[10] Marcus Brandenburger et al. *Rollback and Forking Detection for Trusted Execution Environments using Lightweight Collective Memory*. 2017. arXiv: 1701.00981 [cs.DC].

[11] Ferdinand Brasser et al. "Software Grand Exposure: SGX Cache Attacks Are Practical". In: *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. Vancouver, BC: USENIX Association, Aug. 2017. URL: https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser.

[12] Stephan Brumme. *xxHash*. https://github.com/stbrumme/xxhash. 2018.

[13] Jo Van Bulck et al. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 991–1008. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[14] Guoxing Chen et al. "Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 178–194. DOI: 10.1109/SP.2018.00024.

[15] Guoxing Chen et al. "SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution". In: *2019 IEEE European Symposium on Security and Privacy (EuroSP)* (June 2019). DOI: 10.1109/eurosp.2019.00020. URL: http://dx.doi.org/10.1109/EuroSP.2019.00020.

[16] Raymond Cheng et al. "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts". In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2019, pp. 185–200.

[17] J. Criswell, Nathan Dautenhahn, and V. Adve. "Virtual ghost: protecting applications from hostile operating systems". In: *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems* (2014).

[18] Saba Eskandarian and Matei Zaharia. "An Oblivious General-Purpose SQL Database for the Cloud". In: *CoRR* abs/1710.00458 (2017). arXiv: 1710.00458. URL: http://arxiv.org/abs/1710.00458.

[19] Johannes Götzfried et al. "Cache Attacks on Intel SGX". In: *Proceedings of the 10th European Workshop on Systems Security*. EuroSec'17. Belgrade, Serbia: Association for Computing Machinery, 2017. ISBN: 9781450349352. DOI: 10.1145/3065913.3065915. URL: https://doi.org/10.1145/3065913.3065915.

[20] Owen S. Hofmann et al. "InkTag: Secure Applications on an Untrusted Operating System". In: *SIGPLAN Not.* 48.4 (Mar. 2013), pp. 265–278. ISSN: 0362-1340. DOI: 10.1145/2499368.2451146. URL: https://doi.org/10.1145/2499368.2451146.

[21] Tyler Hunt et al. "Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 533–549. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/hunt.

[22] hyperledger. *Hyperledger Fabric Private Chaincode*. https://github.com/hyperledger/fabric-private-chaincode. 2021.

[23] hyperledger. *Hyperledger Fabric Private Chaincode*. https://github.com/hyperledger/fabric-private-chaincode. 2021.

[24] "Intel Software Guard Extensions (SGX)". In: URL: https://software.intel.com/en-us/isaextensions/intel-sgx/.

[25] Harry Kalodner et al. "Arbitrum: Scalable, private smart contracts". In: *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 2018, pp. 1353–1370.

[26] Ahmed Kosba et al. "Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts". In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 839–858. DOI: 10.1109/SP.2016.55.

[27] Youngjin Kwon et al. "Sego: Pervasive Trusted Metadata for Efficiently Verified Untrusted System Services". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '16. Atlanta, Georgia, USA: Association for Computing Machinery, 2016, pp. 277–290. ISBN: 9781450340915. DOI: 10.1145/2872362.2872372. URL: https://doi.org/10.1145/2872362.2872372.

[28] Dat Le, Shruti Tople, and Prateek Saxena. "Panoply: Low-TCB Linux Applications with SGX Enclaves". In: Jan. 2017. DOI: 10.14722/ndss.2017.23500.

[29] Do Le Quoc et al. "SGX-PySpark: Secure Distributed Data Analytics". In: *The World Wide Web Conference*. WWW '19. San Francisco, CA, USA: Association for Computing Machinery, 2019, pp. 3564–3563. ISBN: 9781450366748. DOI: 10.1145/3308558.3314129. URL: https://doi.org/10.1145/3308558.3314129.

[30] Sangho Lee et al. *Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing*. 2017. arXiv: 1611.06952 [cs.CR].

[31] Joshua Lind et al. "Glamdring: Automatic Application Partitioning for Intel SGX". In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, July 2017, pp. 285–298. ISBN: 978-1-931971-38-6. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/lind.

[32] Yutao Liu et al. "Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation". In: Oct. 2015. DOI: 10.1145/2810103.2813690.

[33] Sinisa Matetic et al. "ROTE: Rollback Protection for Trusted Execution". In: *Proceedings of the 26th USENIX Conference on Security Symposium*. SEC'17. Vancouver, BC, Canada: USENIX Association, 2017, pp. 1289–1306. ISBN: 9781931971409.

[34] Jonathan M. McCune et al. "Flicker: An Execution Infrastructure for Tcb Minimization". In: *SIGOPS Oper. Syst. Rev.* 42.4 (Apr. 2008), pp. 315–328. ISSN: 0163-5980. DOI: 10.1145/1357010.1352625. URL: https://doi.org/10.1145/1357010.1352625.

[35] Oleksii Oleksenko et al. "Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks". In: *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '18. Boston, MA, USA: USENIX Association, 2018, pp. 227–239. ISBN: 9781931971447.

[36] OpenEnclave. *OpenEnclave Sample attested$_t$ls*. https://github.com/openenclave/openenclave/tree/master/samples/attested_tls. 2021.

[37] Vasilis Pappas et al. "Blind Seer: A Scalable Private DBMS". In: *2014 IEEE Symposium on Security and Privacy* (2014), pp. 359–374.

[38] Rishabh Poddar et al. "Visor: Privacy-Preserving Video Analytics as a Cloud Service". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1039–1056. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/poddar.

[39] Raluca Ada Popa et al. "CryptDB: Protecting Confidentiality with Encrypted Query Processing". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: Association for Computing Machinery, 2011, pp. 85–100. ISBN: 9781450309776. DOI: 10.1145/2043556.2043566. URL: https://doi.org/10.1145/2043556.2043566.

[40] Christian Priebe, Kapil Vaswani, and Manuel Costa. "EnclaveDB: A Secure Database Using SGX". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 264–278. DOI: 10.1109/SP.2018.00025.

[41] Felix Schuster et al. "VC3: Trustworthy data analytics in the cloud using SGX". In: 2015 (July 2015), pp. 38–54. DOI: 10.1109/SP.2015.10.

[42] Michael Schwarz et al. "Malware Guard Extension: Using SGX to Conceal Cache Attacks". In: *CoRR* abs/1702.08719 (2017). arXiv: 1702.08719. URL: http://arxiv.org/abs/1702.08719.

[43] Michael Schwarz et al. *ZombieLoad: Cross-Privilege-Boundary Data Sampling*. 2019. arXiv: 1905.05726 [cs.CR].

[44] Youren Shen et al. "Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX". In: vol. abs/2001.07450. 2020. arXiv: 2001.07450. URL: https://arxiv.org/abs/2001.07450.

[45] Pramod Subramanyan et al. "A Formal Foundation for Secure Remote Execution of Enclaves". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2435–2450. ISBN: 9781450349468. DOI: 10.1145/3133956.3134098. URL: https://doi.org/10.1145/3133956.3134098.

[46] Nick Szabo. *Smart contracts*. 1994.

[47] Sai Tetali et al. "MrCrypt: Static Analysis for Secure Cloud Computations". In: vol. 48. Nov. 2013, pp. 271–286. DOI: 10.1145/2544173.2509554.

[48] Chia-che Tsai, Donald E. Porter, and Mona Vij. "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX". In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, July 2017, pp. 645–658. ISBN: 978-1-931971-38-6. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai.

[49] Chia-che Tsai et al. "Civet: An Efficient Java Partitioning Framework for Hardware Enclaves". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 505–522. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/tsai.

[50] Stephen Tu et al. "Processing Analytical Queries over Encrypted Data". In: vol. 6. Mar. 2013, pp. 289–300. DOI: 10.14778/2535573.2488336.

[51] Jo Van Bulck et al. "Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution". In: *Proceedings of the 26th USENIX Conference on Security Symposium*. SEC'17. Vancouver, BC, Canada: USENIX Association, 2017, pp. 1041–1056. ISBN: 9781931971409.

[52] Nicolas Van Saberhagen. *CryptoNote v 2.0*. 2013.

[53] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems". In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 640–656. DOI: 10.1109/SP.2015.45.

[54] Rui Yuan et al. "Shadoweth: Private smart contract on public blockchain". In: *Journal of Computer Science and Technology* 33.3 (2018), pp. 542–556.

[55] Ning Zhang et al. "MUSHI: Toward Multiple Level Security cloud with strong Hardware level Isolation". In: *MILCOM 2012 - 2012 IEEE Military Communications Conference*. 2012, pp. 1–6. DOI: 10.1109/MILCOM.2012.6415698.

[56] Wenting Zheng et al. "Opaque: An Oblivious and Encrypted Distributed Analytics Platform". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 283–298. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zheng.

Appendix A

API

A.1 API Code Snippet

Listing A.1: Verification API

```
/**
 * Initialize a new AuditLogEntry.
 *
 * @param ecall_id id number corresponding to the current ecall
 * @return a pointer to the initalized AuditLogEntry
 */
AuditLogEntry* init_audit_log(int ecall_id);

/**
 * Logs the input data of an ecall.
 *
 *
 */
void log_input_data(AuditLogEntry* entry,
            uint64_t* input_data_pointers, int num_input_data,
            uint64_t* input_supp_data_pointers, int num_supp_input_data);

/**
 * Logs the output data of an ecall.
 *
 * @param entry
 * @param output_data_pointers
 * @param num_output_data
 * @param output_supp_data_pointers
 */
void log_output_data(AuditLogEntry* entry,
            uint64_t* output_data_pointers,int num_output_data,
            uint64_t* output_supp_data_pointers, int num_supp_output_data);
/**
 * Sends an audit log to the verifier enclave.
```

```
 *
 * @param entry pointer to the AuditLogEntry
 */
void send_audit_log(AuditLogEntry* entry);
```

A.2 Opaque: Example API Usage

An example of how to add our verification library API calls into Opaque's `ecall_non_oblivious_sort_merge_join` function.

Listing A.2: Modified ecall_non_oblivious_sort_merge_join

```
    void non_oblivious_sort_merge_join(
        uint8_t *join_expr, size_t join_expr_length,
        uint8_t *input_rows, size_t input_rows_length,
        uint8_t **output_rows, size_t *output_rows_length) {

      // ---------------------- AUDIT LOG ----------------------
      struct AuditLogEntry* log = init_audit_log(1);
      int num_input_data = 1;
      RowReader r2(BufferRefView<tuix::EncryptedBlocks>(
                           input_rows, input_rows_length));
      uint64_t input_data_pointers[num_input_data] = {(uint64_t)&r2};
      int num_supp_input_data = 0;
      uint64_t* input_supp_data_pointers = 0;
      log_input_data(log, input_data_pointers, num_input_data,
                     input_supp_data_pointers, num_supp_input_data);
      // ---------------------- AUDIT LOG ----------------------


        ~ EXISTING CODE ~


     // ---------------------- AUDIT LOG ----------------------
      RowReader r3(BufferRefView<tuix::EncryptedBlocks>(
                       *output_rows, *output_rows_length));
      int num_output_data = 1;
      uint64_t output_data_pointers[num_output_data] = {(uint64_t)&r3};
      int num_supp_output_data = 0;
      uint64_t* output_supp_data_pointers = 0;
      log_output_data(log, output_data_pointers, num_output_data,
                      output_supp_data_pointers, num_supp_output_data);
      send_audit_log(log);
      free_audit_log(log, num_input_data, num_supp_input_data,
                     num_output_data, num_supp_output_data);
     // ---------------------- AUDIT LOG ----------------------
}
```

An example of how to add our verification library API calls into Opaque's `ecall_non_oblivious_sort_merge_join` function.

A.3 Hyperledger FPC: Example API Usage

Listing A.3: Modified ecall_non_oblivious_sort_merge_join

```
    int invoke(uint8_t* response, uint32_t max_response_len,
        uint32_t* actual_response_len, shim_ctx_ptr_t ctx) {

      // ---------------------- AUDIT LOG ----------------------
      struct AuditLogEntry* log = init_audit_log(1);
      int num_input_data = 1;
      FPCReader fpcIn(ctx);
      uint64_t input_data_pointers [num_input_data] = {(uint64_t)&fpcIn};
      int num_supp_input_data = 0;
      uint64_t* input_supp_data_pointers = 0;
      log_input_data(log, input_data_pointers, num_input_data,
                     input_supp_data_pointers, num_supp_input_data);
      // ---------------------- AUDIT LOG ----------------------


        ~ EXISTING CODE ~


     // ---------------------- AUDIT LOG ----------------------
      FPCReader fpcOut(response);
      int num_output_data = 1;
      uint64_t output_data_pointers [num_output_data] = {(uint64_t)&fpcOut};
      int num_supp_output_data = 0;
      uint64_t* output_supp_data_pointers = 0;
      log_output_data(log, output_data_pointers, num_output_data,
                      output_supp_data_pointers, num_supp_output_data);
      send_audit_log(log);
      free_audit_log(log, num_input_data, num_supp_input_data,
                     num_output_data, num_supp_output_data);
     // ---------------------- AUDIT LOG ----------------------
}
```

Appendix B

Ruleset

B.1 Ruleset JSON Example

```
{
    "ecall_id": {
        "ecall_non_oblivious_sort_merge_join": 0
    },
    "operator_id" : {
        "nonObliviousSortMergeJoin" : 0
    },
    "ecall_data_source_counts": [
        {
            "ecall_id": 0,
            "input": 1,
            "supp_input": 0,
            "output": 1,
            "supp_output": 0
        }
    ],
    "rulesets": [
        {
            "operator_id" : 0,
            "ecall_ids": [0],
            "num_rounds" : 1,
            "rules": []
        }
    ]
}
```