Automating Execution Verification in Distributed Enclave-Based Data Analysis Applications

Karen Tu

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-65
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-65.html

May 13, 2021

Copyright © 2021, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Automating Execution Verification in Distributed Enclave-Based Data Analysis Applications

by Karen Tu

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 12, 2021

(Date)

* * * * * * *

Professor Ion Stoica
Second Reader

May 12, 2021

(Date)

Automating Execution Verification in Distributed Enclave-Based Data Analysis Applications

by

Karen Tu

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

Automating Execution Verification in Distributed Enclave-Based Data Analysis Applications

Copyright 2021
by
Karen Tu

Abstract

Automating Execution Verification in Distributed Enclave-Based Data Analysis Applications

by

Karen Tu

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. In distributed applications, we can place code that affects data flow but not the data content outside of the enclave; we can then verify the correctness of the data flow execution. 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%.

To my always supportive friends and family.

Contents

List of Figures

List of Tables

Acknowledgments

Thank you to Professor Raluca Ada Popa for giving me the opportunity to be a part of RISELab. I also want to thank Rishabh Poddar and Wenting Zheng for being great mentors and giving me valuable feedback. I have learned a lot from all three of you, and it has been a privilege to work with you.

Thank you to Saharsh Agrawal for being a great research partner, I could not have done this without you.

Finally, thank you to my family for keeping me sane through a crazy year.

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 for a hospital's patient data. Additionally, with increasing concern over data privacy and potentially malicious third party cloud computing providers, there needs to be 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 [47] and Glamdring [29] 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 a lot of potential to be used for outsourcing data analysis, which is often distributed for large datasets or complicated analysis. However, the existing partitioning solutions 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 thus 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 [40] and Opaque [52]. 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.

In this report, we present a verification library that aims to help developers of distributed enclave-based data analysis applications verify the proper execution of code outside of the enclave. Given a partitioned codebase for a distributed data analysis application, our library can be used to verify data flow integrity. Within each enclave function, the developer uses our library to create an `AuditLogEntry` every time that function is called. During program execution, based on the data flow rules specified by the developer, the audit logs are cross checked with the rules to make sure that all data computation tasks were completed. Note that if there is no data dependency between two enclave functions, then no rule will exist to check whether the inputs and outputs between them corroborate. We also include several optimization options in our library, such as automatically inserting the API calls into enclave function definitions. 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.

This report was partially written in collaboration with another Master's student, Saharsh Agrawal. The sections that I wrote completely independently are listed below.

- 2.3: Background, Code Annotations
- 3.4: Related Work, Code Analysis
- 6.4.2: System Design, Optimizations, Automatic Ruleset Generation
- 8.4: Evaluation, Optimizations

The following sections contain paragraphs with discussion about the automatic ruleset generation. Those specific paragraphs are my independent work, but the rest of the section is a collaborative work.

- 7: Implementation
- 9: Limitation & Future work

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) [23] 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 remote attestation feature which allows the client 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 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 that the application's untrusted code makes the correct ECALLs. 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 Code Annotations

Code annotations are used to specify metadata about the code; they typically do not have an impact on the program execution, unless reflection is used during runtime to access the annotations. Annotations are useful to signal something to the compiler; for example, a `@deprecated` annotation on a function may prompt the compiler to print a warning when that function is used.

Some programming languages such as Scala, Java, C#, Python, and Ruby support user-defined annotations, which can be used for documentation or giving compilers custom directions. However, other languages such as C++ and C do not support custom code annotations. A way to circumvent the lack of support for custom annotations is to use compiler and static analysis tools. Many compiler and static analysis tools are used to debug, find security and memory leaks, or to check code style. However, these tools can also be used in more general ways to analyze application code, and can be extended to implement annotations. For example, both Clang and VC++ are C/C++ compilers that use attributes (a feature in both C/C++) to support custom annotations.

Regardless of whether annotations are built into a programming language or not, the most straightforward method to analyze code annotations in a piece of source code is to leverage compilers. For Scala, this would mean using a compiler plugin. For C/C++, this would mean using a compiler that has built-in support for custom annotations.

Chapter 3

Related Work

3.1 Untrusted Cloud

CryptDB [38], MrCrypt [45], BlindSeer [36], Monomi [48], and Always Encrypted [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 [14] uses compiler instrumentation and run-time checks to create a protected region of memory. Flicker [33] and MUSHI [51] 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 [31], InkTag [20], and Sego [24] 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 [46], and Occlum [43] 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 large attack surface. Another major issue with porting legacy applications into SGX is that SGX is vulnerable to side channel attacks ([50], [9], [41], [18]).

SCONE [4] and Ryoan [22] 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 execute only security-sensitive code; all other code should be placed outside.

Panoply [26] 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 [47] and Glamdring [29] 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 [39] is a database engine that places all sensitive data into enclaves. To maintain data integrity, it keeps a database log. ObliDB [17] is also an enclave database, but uses oblivious algorithms to hide access patterns, which incurs a large overhead. Visor [37] is a video analytics platform using a TEE across CPUs and GPUs, using Graphene [46] 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 functionalities 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 [40] runs on Hadoop within SGX enclaves. Verification is performed by workers sending information about data inputs and outputs to a master node. Opaque [52] 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 [27] 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 Code Analysis

Although there are problems with static code analysis, such as complications arising from reflection capabilities [25], it is a straightforward tool to use to automatically construct the relationships between ECALLs. Code annotations are useful for analyzing program code, such as making static analysis easier [19] and for finding bugs [21]. For languages without built in annotations, compilers can be extended to support custom annotations. For example, in [15] C++ attributes are used as annotations and in [1] a Java compiler is extended.

Other code analysis techniques include taint tracking ([7], [16]) and data flow analysis [30]. Taint tracking is not suited for enclave applications; anything outside of the enclave is untrusted, so all data would be marked as tainted. Traditional data flow analysis tracks how variable values change during program execution, which is not so useful for enclave applications as the data is encrypted, and we want to track the data flow through the untrusted code *and* the enclave. However, for verifying correct execution of ECALLs, some way to construct the data flow is needed. The relationship between ECALLs depends on how data is passed from one ECALL to another. This information can be encapsulated using data annotations.

Chapter 4

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 [44] in which the authors present a formalization of idealized enclave platforms, including a formal model of enclave programs and the adversary.

4.1 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 [50, 42, 11, 13] and software-based attacks [18, 49, 28]. 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 [12, 8, 34, 32].

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.

4.2 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. 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 5

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. The ruleset configuration file can be generated automatically using code annotations.
- For developers of applications with many enclave functions, we provide a script to automatically insert the necessary verification library API calls.

5.1 Architecture

We present the overall application architecture in figure 5.1 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.

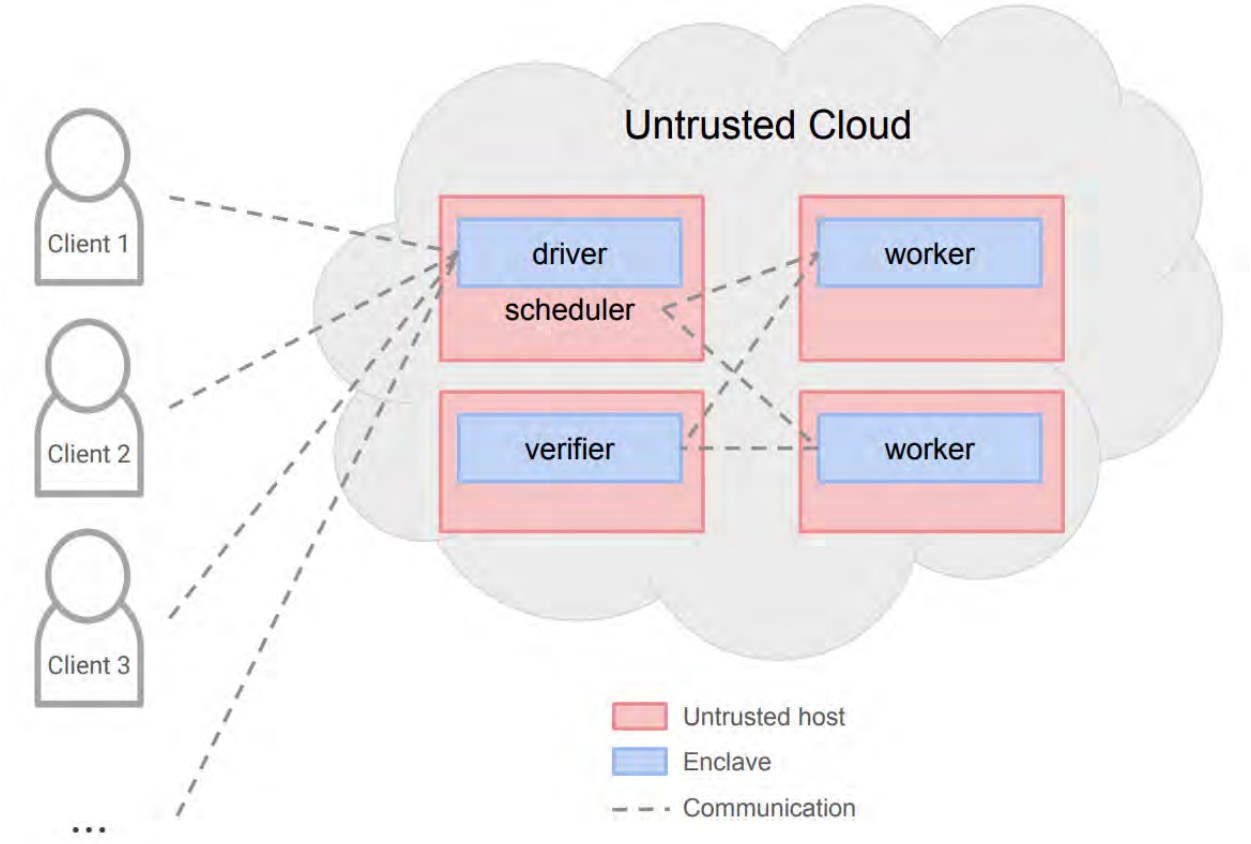

Figure 5.1: Distributed data analysis enclave-based application architecture after integrating our verification library and deployment.

5.1.1 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 5.1, 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 6.2.

5.1.2 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.

5.1.3 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 5.1 corresponds to a single VM in the cloud, and each VM can have multiple enclaves.

Chapter 6

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.

6.1 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 6.1: 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.

6.2 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 5.1.1.

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.

6.3 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.

6.3.1 Design Tradeoffs

Keeping in mind our API as specified in 6.1, 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 [52], in the untrusted code, the function `collect` is used 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 needs to provide some way for our verification library to iterate over the smallest unit of data as each individual ECALL call is expected to process 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.

6.3.2 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 the inputs and outputs of different ECALLS match 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 its 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 6.1. 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.

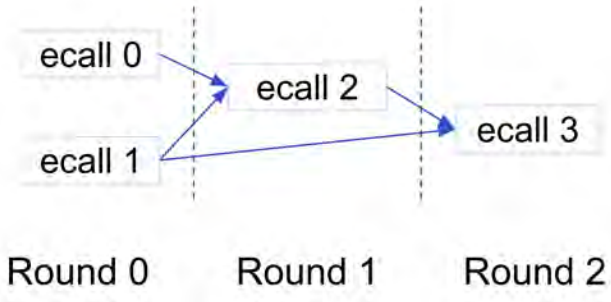

Figure 6.1: Example of ECALL execution flow split into rounds.

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 6.1 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.

6.4 Optimizations

6.4.1 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 the application Opaque [52], all of the ECALLs have arguments `input_rows` and `output_rows` as input and output data sources, respectively. To use our verification library, 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.

6.4.2 Automatic Ruleset Generation

Another natural part of the workflow that is well suited to being automated is the step where the developer must specify a ruleset JSON file. Manual specification is always prone to human error, and when manually creating the ruleset file, it is quite easy to get the numbers wrong for the ECALL ID mapping and the data source parameter indices. It is easier for the developer to mark their code files directly with annotations rather than to copy over ECALL metadata (function names, parameter indices, data source types, etc.) to a separate file.

There are four parts of the ruleset JSON that each must be automatically generated. In this section, we present how the developer should annotate their code files. Then, we explain how the annotations are used to generate the corresponding section of the ruleset JSON file.

1. ECALL name to ECALL ID mapping
   The developer needs to mark the ECALLs affecting data flow with `@verify` in the EDL file. We automatically assign ECALL IDs in order that they appear in the EDL file.

2. operation name to operation ID mapping
   This is implemented the same way as the ECALL name to ECALL ID mapping; the developer needs to mark the operations (which are defined in the untrusted code - ECALLs are called within the operation code).

3. for each ECALL, metadata for parameters that are data sources
   In the ECALL function declaration, annotate the parameters that are data sources with their data source types: `@input`, `@supplementary_input`, `@output`, and `@supplementary_output`. The counts for each data source type is computed automatically; if two of the parameters are tagged with `@input`, then the count of data sources of type input is set at 2.

4. list of rules for each operation
   Annotate variables that contain ECALL outputs with `@single`, `@union` or `@each`. This signifies that the output of the ECALL will be the input to another ECALL. We use static analysis to link which ECALL the output came from and for which ECALL it becomes the input. Then, we use this information to build the dependency graphs between the ECALLs. From there, we can split the ECALL execution flow into rounds. One limitation is that we have to enforce the ECALLs to only have one output source; there is no easy way to figure out what the relative ordering of the outputs are (we cannot generate the index number for the output in the rule).

Annotation Examples

Shown below is an example of the annotations that would be added to an EDL file. The added annotations are bolded. `ecall_sample` would be assigned with ID 0 and `ecall_find_range_bounds` would be assigned with ID 1.

```
    public void ecall_sample(
     [user_check] @input uint8_t *input_rows, size_t input_rows_length,
     [out] uint8_t @output **output_rows, [out] size_t *output_rows_length);
    public void ecall_find_range_bounds(
     [in, count=sort_order_length] uint8_t *sort_order, size_t sort_order_length,
     uint32_t num_partitions,
     [user_check] @input uint8_t *input_rows, size_t input_rows_length,
     [out] uint8_t @output **output_rows, [out] size_t *output_rows_length);
```

Shown below is an example of annotating a code snippet from EncryptedSortExec.scala, and the interface function FindRangeBounds from SGXEnclave.scala in Opaque, in order to specify a ruleset for the operation EncryptedSortExec.

```
    // Collect a sample of the input rows
      @join val sampled = applyLoggingLevel(childRDD, "enclave.Sample") {
        childRDD =>
        Utils.concatEncryptedBlocks(childRDD.map { block =>
          val (enclave, eid) = Utils.initEnclave()
          val sampledBlock = enclave.Sample(eid, block.bytes)
          Block(sampledBlock)
        }.collect)
      }

      // Find range boundaries parceled out to a single worker
      @single val boundaries = applyLoggingLevel(childRDD,
                                                 "enclave.FindRangeBounds") {
        childRDD => childRDD.context
          .parallelize(Array(sampled.bytes), 1)
          .map { sampledBytes =>
            val (enclave, eid) = Utils.initEnclave()
            enclave.FindRangeBounds(eid, orderSer, numPartitions,
                                    @single sampledBytes)
          }
          .collect
          .head
      }
```

```
    @native def FindRangeBounds(
      eid: Long,
      order: Array[Byte],
      numPartitions: Int,
      @input input: Array[Byte]
  ): Array[Byte]
```

The rule that would be constructed for operation EncryptedSortExec is:

```
join ECALL_0 output_0 round_0 = single ECALL_1 input_0 round_1
```

Chapter 7

Implementation

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 [10], 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 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 operation 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 [35] 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 [6] to serialize the `AuditLogEntry` data structure prior to sending it over the network.

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 8

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.

8.1 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 8.1 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 8.1: Lines of code across verification library files.

8.2 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.

8.3 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.2. 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 6.4.1 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 8.2: 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.

8.4 Optimizations

We do not consider the optimization code as part of the library size evaluation, the purpose of evaluating the library size is to measure the impact on the TCB within the enclave. The optimization code is not run within the enclave; it is run by the developer separately. Instead, we evaluate the optimizations based on how useful they are to the developer.

8.4.1 Automatic API Integration

Using the script results in the developer only having to write 4 lines detailing ECALL parameter metadata for each ECALL instead of 14 lines of API calls. Although this may seem like a trivial improvement, for complicated applications with many ECALLs a significantly larger amount of developer effort is saved.

8.4.2 Automatic Ruleset Generation

Manual specification of the ruleset JSON file becomes error prone as the number of operations and ECALLs for an application increases; it is easy to put in the wrong ecall ID or round number. Automatic ruleset generation streamlines the developer's workflow; instead

of having to transfer much of the information contained in the code where the ECALLs to an external JSON file, the developer can annotate the code directly.

The overhead of running the ruleset generation code is amortized, as it is only run each time the application is recompiled or updated. In addition, any overhead of running the code would be offset by the time saved from having to manually specify the ruleset.

Ruleset File Length

One way of evaluating the effectiveness of the automatic ruleset generation optimization is to consider the number of lines saved. The length of the ruleset JSON file depends on the number of operations, the total number of rules, and the total number of ecalls. For o operations, r total rules, e total ecalls, the ruleset JSON file is expected to be $4 * o + 6 * e + r$ lines long. A typical data analysis application will have at least 20 possible operations, and thus at least 20 ECALLs, which leads to a ruleset JSON file that is at least 200 lines long.

Annotations

- ECALL mapping and operation mapping: To annotate the ECALLs and operations with `@ecall, @operation` respectively for e ECALLs and o operations takes $e + o$ annotations, which is easier than writing out $e + o$ lines of mapping in a JSON file.

- ECALL parameter data source metadata - data source type, data source counts: We annotate the parameters with data source type in the EDL and in the untrusted code that interfaces with the enclave with the data source type `@input, @supp_input, @output, @supp_output`. The counts for each data source type for an ECALL is computed automatically. If we consider the average number of parameters per ECALL that are data sources as p, for e ECALLs this results in $2 * p * e$ annotations. We can reasonably expect p to be between 2 and 5 for most ECALLs. For this step the amount of work done is comparable with the manual specification.

- Rules for each operation: Each rule corresponds to an input, output data source pair for two different ECALLs. In the untrusted code where the ECALL is called, it is annotated with `@single, @each, @union`. The number of rounds for an operation is automatically calculated, which makes one less value per operation for the developer to manually specify. Not having to manually specify the rules significantly reduces the potential for human error. Although we tried to make the rule format simple, there are still 10 fields for each rule that a user must specify. 6 of these fields are indices which are especially prone to human error. The automatic rule generation greatly reduces the chance of such errors.

Chapter 9

Limitations & Future Work

9.1 Limitations

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 partially due to the implicit barrier induced by runtime verification which makes the code *blocking* until a round has been verified. The verification itself also has latency overhead, 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.

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 use something similar to a libOS.

9.2 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 Section 6.3. 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 6.3.2, 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 ECALLs.

The current implementation of automatic ruleset generation only works with Scala and C/C++ code, but the part of the application outside of the enclave can be written in any arbitrary programming language. The C/C++ code analysis can still be used, as the enclave function definitions must be written in C/C++. However, the untrusted code can be written in any arbitrary language, so in the future we hope to support more languages using the same strategy of analyzing code annotations. Parsers and static analyzers exist for any reasonably popular programming languages, so the strategy of using annotations is easily extensible even to languages that don't support annotations.

There is also potential for more automation, such as for partitioning the application into trusted and untrusted code. Automatic partitioning solutions exist (Civet [47] and Glamdring [29]), but they are not targeted for distributed applications. An automatic partitioning framework for distributed applications would be very useful if used together with our verification library. The framework can identify code that affects data flow but not the actual data content and categorize such code as untrusted, thus resulting in a smaller trusted computing base than existing partitioning solutions can provide. Then our library can be used to verify that the execution and data flow happened correctly. We can borrow strategies from Civet and Glamdring, such as using annotations and static analysis to mark the sensitive data.

Another direction for future work is to integrate our library with more distributed data analysis enclave-based applications beyond Opaque, to make sure 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 10

Conclusion

Overall, we presented a library that saves the developer of a distributed data analysis enclave-based application the effort of writing their own execution flow integrity code logic. We even provide optimizations to automate parts of the workflow to further decrease the developer effort required. We integrated our library into an existing application to show that it not only has a reasonable code footprint but also a low latency overhead. Most importantly, we showed that our library is practical to use.

Bibliography

[1] Edward Aftandilian et al. "Building Useful Program Analysis Tools Using an Extensible Java Compiler". In: *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. 2012, pp. 14–23. DOI: 10.1109/SCAM.2012.28.

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

[7] Ajay Brahmakshatriya et al. "ConfLLVM: A Compiler for Enforcing Data Confidentiality in Low-Level Code". In: *Proceedings of the Fourteenth EuroSys Conference 2019*. EuroSys '19. Dresden, Germany: Association for Computing Machinery, 2019. ISBN: 9781450362818. DOI: 10.1145/3302424.3303952. URL: https://doi.org/10.1145/3302424.3303952.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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).

[15] Marco Danelutto et al. "Introducing Parallelism by Using REPARA C++11 Attributes". In: Feb. 2016, pp. 354–358. DOI: 10.1109/PDP.2016.115.

[16] William Enck et al. "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones". In: *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. Vancouver, BC: USENIX Association, Oct. 2010. URL: https://www.usenix.org/conference/osdi10/taintdroid-information-flow-tracking-system-realtime-privacy-monitoring.

[17] 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.

[18] 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.

[19] Kalle Hjerppe, Jukka Ruohonen, and Ville Leppänen. "Annotation-Based Static Analysis for Personal Data Protection". In: *CoRR* abs/2003.09890 (2020). arXiv: 2003.09890. URL: https://arxiv.org/abs/2003.09890.

[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] David Hovemeyer, Jaime Spacco, and William Pugh. "Evaluating and tuning a static analysis to find null pointer bugs". In: vol. 31. Jan. 2005, pp. 13–19. DOI: 10.1145/1108768.1108798.

[22] 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.

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

[24] 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.

[25] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. "Challenges for Static Analysis of Java Reflection - Literature Review and Empirical Study". In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 2017, pp. 507–518. DOI: 10.1109/ICSE.2017.53.

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

[27] 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.

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

[29] 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.

[30] Yin Liu and Ana Milanova. "Static Information Flow Analysis with Handling of Implicit Flows and a Study on Effects of Implicit Flows vs Explicit Flows". In: *2010 14th European Conference on Software Maintenance and Reengineering*. 2010, pp. 146–155. DOI: 10.1109/CSMR.2010.26.

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

[32] 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.

[33] 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.

[34] 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.

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

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

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

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

[43] 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.

[44] 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.

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

[46] 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.

[47] 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.

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

[49] 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.

[50] 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.

[51] 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.

[52] 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

```
/**
 * 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 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.1: 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 ----------------------
}
```

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": []
        }
    ]
}
```