Post Verification of Integrity of Remote Queries in Opaque

Andrew Law How Hung

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-44
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-44.html

May 11, 2021

Post Verification of Integrity of Remote Queries in Opaque

by

Andrew Law

A thesis submitted in partial satisfaction of the

requirements for the degree of

Masters of Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Raluca Ada Popa, Chair
Professor Joseph Gonzalez

Spring 2021

Post Verification of Integrity of Remote Queries in Opaque

Contents

List of Figures

List of Tables

Acknowledgments

A big thank you to Professor Raluca Ada Popa, who gave me a fair chance at participating in research, got me interested in computer security, and motivated me to pursue a graduate degree. Additional thanks to my mentors: Rishabh Poddar, Wenting Zheng, and Chester Leung, who have helped and directed me at every step of the way in my contributions at RISELab. Thank you to Professor Gonzalez for contributing feedback and taking the time to read over this work, and for being an incredible instructor providing insight into data science. Special thanks to Chester Leung again, who guided me in this project specifically, contributing to much of the ideas and code presented.

Abstract

Post Verification of Integrity of Remote Queries in Opaque

by

Andrew Law

Masters of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Raluca Ada Popa, Chair

Many companies and individuals in the modern age of big data outsource their data and computation to third parties who specialize in maintaining hardware and cloud services. However, they may want to keep their data and computation secret for business or privacy interests. Opaque is a system that offers secure data analytics in an untrusted cloud by leveraging special hardware called secure enclaves as well as a variety of other novel techniques. Opaque is built on Spark SQL, a powerful Spark module that performs data processing and analytics. Spark SQL, and by extension, Opaque, distributes data among nodes to parallelize workloads. While each such node is trusted, the job driver/scheduler, which resides in the cloud and delegates the tasks, is not. This work outlines a design and implementation to preserve query integrity in the face of the untrusted scheduler using logs in the form of HMAC outputs and graph computations.

Chapter 1

Introduction

1.1 A Motivation to Secure Analytics

The modern internet is backed by “big data” and “cloud computing.” In today’s age, more data than ever before about everything imaginable is being generated and relied upon by companies big and small. Tech giants like Google and Facebook collect and categorize consumer data to sell ad targeting to generate their yearly multi-billion dollar revenue. Other smaller companies claim to be “data driven,” performing specific analyses on more niche data sets. What these companies have in common is that they all often outsource their computation to cloud providers, who specialize in maintaining the hardware for both corporate and consumer use.

Customers of cloud providers, however, may not trust the cloud providers with their data, as the data could be sensitive, or maybe they want to protect their own business interests and hide their computation from prying eyes. *Secure analytics systems* leverage advanced hardware and/or cryptographic techniques to protect data stored in the cloud from malicious entities who have some degree of control over the hosting service, whether those actors are the cloud provider admins themselves, or perhaps hackers who have compromised the cloud services and have an interest in peering at private information.

1.2 Introduction to Opaque

Whatever one might want to use secure analytics for, Opaque [14] is a system that aims to fulfill all of the specifications outlined above; it provides *private data analysis on third party hardware.*

Opaque is a system developed by researchers at UC Berkeley that aims to be an “Oblivious and Encrypted Data Analytics Platform.” Essentially, it runs Spark SQL inside hardware enclaves on the computers maintained by the cloud provider to ensure that computation on sensitive data is secure. The Opaque paper offers two different modes that provide different levels of security. Oblivious mode uses new oblivious database operators to compute in a data-oblivious fashion, mitigating memory access side channel attacks on hardware enclaves. Encrypted mode foregoes data oblivious operations in exchange for better performance. In both modes, data confidentiality and integrity are preserved (save for side channel leakage on the hardware enclaves.)

1.3 Limitations of Opaque in the Open Source

Open source code for Opaque exists on GitHub [15]. However, at this time, the open source version of Opaque is lacking some of the functionality proposed in the original 2017 paper. The main limitation we will address is the lack of integrity enforcement. In security, we typically aim to enforce two key properties: confidentiality and integrity. Confidentiality is the general term for keeping data secret. This is taken care of intrinsically by the hardware enclaves and data encryption. On the other hand, integrity is a property that ensures that data has not been tampered with in transit. Computation integrity in Opaque is unquestionably important; not only should users be confident that their data is secret, but they should also be confident that when they request a query to be run on the data in the cloud, that Opaque has *actually* run that specific query and not some other query.

As a concrete example to put this into perspective, consider a bank (Bank A) that has a dataset of customers and their checking account balance. Bank A wants to compute the query SELECT name FROM users WHERE users.balance < 500. Recall that Opaque runs on Spark, which divides a query among different workers. Spark will schedule workers that each compute their own part of the query, and then send information to one another to eventually output a final result containing only the information asked by the user. In our model, the workers are trusted since they are running on hardware enclaves, but the task driver which orchestrates them and is responsible for their inputs and outputs is not. This query mentioned above includes a selection followed by a projection. Computational integrity in this case would ensure that between the projection and selection, no rows were dropped, added, or modified, and that the expected query plan computed by Spark's query planner is actually the one being run by the workers (ie. the task DAG actually was a selection followed by a projection and not some other combination of operations).

In Opaque's current version at the time of writing, the mechanism to ensure this property is not in place. While the dataframe records are encrypted with AES-GCM mode and are thus protected to some degree, the untrusted job driver residing in the cloud is able to tamper with data and communication flow while it is being passed between Opaque workers, and could theoretically affect the outcome of an Opaque query.

1.4 Aim of this Report and Contributions

In this MS report, my goal is to design and implement an integrity mechanism for Opaque such that the problem outlined in the section above can be solved. Although its specific implementation may be centered around Opaque's internals, this design is at a high level generally applicable to similar secure analytics systems that leverage distributed computing.

In addition, to initially familiarize myself with Opaque's codebase, we also contributed to an orthogonal effort to complete Opaque's functionality suite in the TPC-H benchmark, including implementing expressions such as "CaseWhen" and "Like".

Chapter 2

Background

2.1 Hardware Enclaves

Hardware enclaves provide a region of trusted memory on a computer that no other process is able to tamper with or see in plaintext without the permission of the enclave. This includes both user mode and kernel processes, including even the operating system of the machine. This property works well with the purpose of Opaque - running Spark SQL operators in the enclave allow a user to do analytics on data while keeping the data and computation secret from the host system (ie. the cloud provider). Thus, any code that runs in the enclave can be considered "trusted." Code that runs outside the enclave is considered "untrusted."

Hardware enclave software libraries may provide a remote attestation service whereby users are able to affirm that the code run by the hardware enclave is correct. In this case, an Opaque client performs remote attestation to ensure that Opaque code is being run by the hardware enclave.

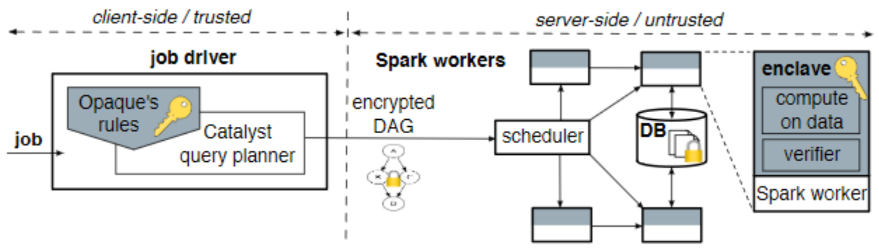
Opaque uses the Intel SGX [8] enclave as its hardware enclave of choice. This choice is relatively unimportant at a high level but is mentioned because the details of implementation do depend on the specific hardware. As long as the user trusts both Opaque and Intel, the user can be sure that their data is safe from the cloud provider after performing remote attestation.

2.2 Apache Spark/Spark SQL

Apache Spark [9] is the industry standard all in one data analytics engine for "large scale data processing" and is what Opaque is built on. It provides almost any functionality one could think of in the data analytics space, including implementations of SQL, DataFrames, machine learning utilities, and more. Additionally, it is widely accessible, as it is available as a package in Java, Scala, Python, and R, where it is able to be used both in large applications and in quick interactive sessions. "Apache Spark achieves high performance for both batch and streaming data, using a state-of-the-art DAG scheduler, a query optimizer, and a physical execution engine." Using parallelization and various other optimizations for computing on large datasets, Spark's website claims to be able to run workloads 100x faster than normal. It is also very portable, being able to run standalone in a local cluster, or in various other standardized environments such as Amazon's EC2, Hadoop YARN, or Kubernetes.

2.3 Opaque System Design Overview/Driver

Figure 2.1: Figure 2 from the original Opaque [14] paper illustrates the high level architecture of the system.



Opaque extends Spark's query planner with new strategies and plans that handle encrypted data. When a query is submitted, the query is processed by Spark's complex query optimizer, *Catalyst*, which, with help from Opaque's new rules and strategies to handle encrypted data, outputs a task DAG. This DAG encapsulates the partitioning of the dataset, which database operators should be computed on these data partitions, and how data partitions are sent from worker to worker. The job driver, which has since been moved to be run on the cloud server, is responsible for interpreting the task DAG and orchestrating workers to actually execute the query plan.

Chapter 3

Related Work

In this section, we discuss a selection of existing works broadly related to securing systems with trusted hardware, and then more specifically, works related to the provision of integrity for interaction with remote databases. For these, we compare approaches and threat models to those used in Opaque.

3.1 Haven [4]

Haven [4] is a system that offers trusted execution of unmodified legacy applications in an untrusted environment. Like Opaque, the security relies on a hardware enclave, and its implementation also uses Intel SGX. Haven focuses on portability of applications; a user should be able to run their program in Haven's environment and have it "just work."

3.2 Graphene-SGX [13]

Graphene-SGX is a work that aims to demonstrate a library OS than can run unmodified applications on SGX hardware with lower overhead costs than previously thought to be possible.

3.3 Panoply [11]

Panoply [11] is another work that leverages Intel SGX enclaves to run Linux applications in lightweight sandboxes the authors call "microns." Panoply "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." This property is similar to the aims of this work, since we also aim to enforce integrity for inter-enclave interactions.

3.4 SCONE [6]

SCONE [6] is a similar work to Panoply that attempts to secure Docker containers using Intel SGX hardware while retaining similar performance and throughput as normal Docker without SGX.

3.5 Visor [17]

Visor [17] is a system that secures the data pipeline involved in processing video stream data with machine learning modeling. It uses a hybrid trusted execution environment spanning both CPU and GPU and uses data oblivious algorithms to protect against leakage via memory access pattern side channel attacks.

3.6 Authentication and Integrity in Outsourced Databases [1]

Authentication and Integrity in Outsourced Databases (AIOD) [1] is an early work (2006) pertaining to integrity verification of individual tuples of a remote database. It has a similar threat model to this work in that there is a trusted client which interacts with an untrusted server hosting the client's outsourced database.

However, unlike Opaque which aims to provide confidentiality of the data, integrity of computation, and a wide range of computation functionality over the remote data, AIOD aims simply to provide basic database functionality (create, store, delete, access), or in other words, does not provide the means to do computation over data, nor does it aim to preserve confidentiality. It only tries to enable the client to fetch some subset of the database while preventing tampering of the returned tuples.

AIOD accomplishes this via cryptographic methods - specifically, by attaching RSA and DLP signatures to database records to preserve integrity. This is similar in some ways to our approach, which relies heavily on cryptographic MAC functions to attach trusted logs to encrypted data, but also different since hardware enclaves provide a source of trust colocated with the untrusted server whereas AIOD does not use (and did not have access to) this technology at the time.

Finally, although the end goal of query integrity is the same, Opaque's setting involves a distributed computation system whereas AIOD tackles problems pertaining to a monolithic RDBMS.

3.7 A query integrity assurance scheme for accessing outsourced spatial databases [2]

This work tackles a similar problem to the one in AIOD, in that rather than enabling query/computational integrity over a dataset, Ku et al. [2] enables integrity over the database itself. It differs from AIOD in that it specializes in protecting the integrity specifically for encrypted spatial data, or data that describes objects defined in a geometric space. Its approach is based on an "auditing" method in which data is probabilistically duplicated and encrypted with a different key, and then is subject to being checked by the client, pressuring the server to remain honest. The discussion of similarities and differences with Opaque is largely the same as the one when examining AIOD.

3.8 Efficient Query Integrity for Outsourced Dynamic Databases [3]

This work by Zheng, Xu, and Ateniese implements query integrity defined very similarly to that in Opaque for a slightly different setting; in particular, the authors split up the "trusted client" into two parties: the database owner and a database querier. The work provides query integrity to the database querier while allowing the database owner to do updates to the outsourced database stored in the cloud. Their approach combines a variety of cryptographic techniques including Merkle B-Tree datastructures and homomorphic linear tags.

3.9 Ryoan [7]

Ryoan's [7] setup is similar to Opaque's in that it provides a remote distributed data analytics system on secret data in an untrusted environment. Like Opaque, it leverages Intel SGX to establish trusted "sandboxes". However, Ryoan's threat model and system framing is also different from Opaque, since there are multiple "clients" in Ryoan who are mutually distrustful. Ryoan protects these clients from each other. Most importantly, the equivalent of Opaque's untrusted job driver, which is responsible for scheduling and coordinating Opaque workers and is one of the main threats in our work, is initialized in a "master enclave" in Ryoan and is trusted. Therefore, the executed query graph is intrinsically trusted in Ryoan. Overall, Ryoan concerns itself more with making sure each individual node behaves properly (does not leak data), unlike our work, which verifies the structure of the execution graph at large as well as the integrity of the edges between each node.

3.10 VC3 [5]

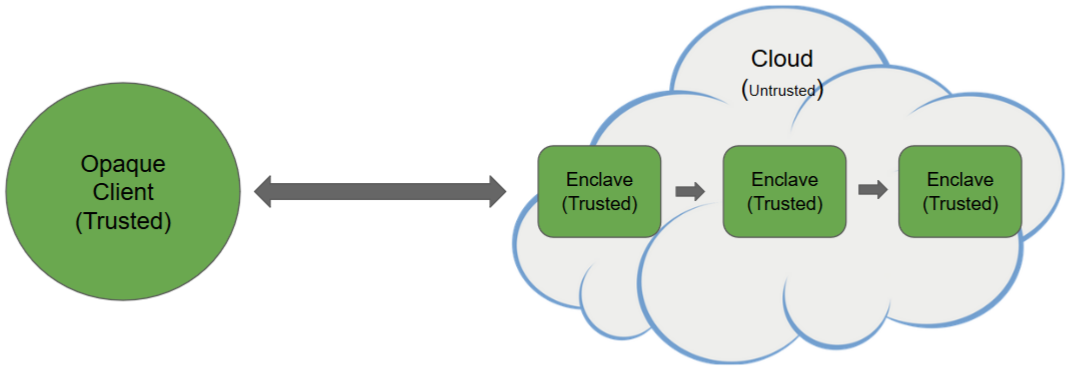
VC3 [5] is the most similar system to Opaque in this list; it is a distributed secure analytics platform supporting outsourced cloud computation on encrypted data and, like Opaque, leverages Intel SGX to provide a secure memory region in the cloud. Parts of VC3's integrity scheme are quite similar to Opaque's runtime verification procedure, where successor nodes verify part of the input received from the predecessor node. VC3 additionally provides safeguards protecting enclave code memory safety using a custom compiler.

On the other hand, my work focuses mostly on post verification of the query graph representing data flow between workers, which are orchestrated by an untrusted entity. This workflow is not emphasized in the VC3 paper. In Opaque, post verification is a step that can be done offline, reducing in band overhead. VC3 chooses to run integrity verification as an in band procedure. Further, the VC3 implementation and discussion is centered very much around MapReduce, unlike Opaque which is based on Spark.

Chapter 4

Threat Model

Figure 4.1: Threat Model Visualization



This work's trusted computing base is Spark and an abstract model of a hardware enclave that protects against a malicious host operating system that controls everything outside the enclave, but cannot attack the enclave memory or get information from the processor including register information while enclave code is executing or CPU information such as processor keys. The attacker, which could include the cloud provider administrators or a malicious actor who has compromised the cloud service, has full control of the cloud operating system and software stack.

The attacker can attempt to spoof or drop encrypted data being passed between Spark workers in the cloud, modify the flow of communication between Spark workers running in enclaves, and in general, perform any other operations on memory outside the protection of the hardware enclaves.

We assume security of the hardware enclave; ie. the attacker can not tamper with code execution or modify memory inside the hardware enclave, and that the attacker is unable to tamper with the enclave attestation procedure.

Side channel attacks on specific enclave technology and denial of service are out of scope of this work. In particular, timing [12], power [16], DoS [10], or other sorts of side channel attacks that violate the threat model of a powerful abstract hardware enclave are out of scope, since those are largely orthogonal to the high level ideas being presented in this work since, if for example an attack specific to Intel SGX were to be discovered, another instantiation of the enclave model could be used instead.

Memory access pattern side channel attacks are not protected against, since data obliviousness is not implemented in this work.

In conclusion, the Opaque user must trust Opaque, the version of Spark SQL that Opaque is built on, and the hardware enclave model backing Opaque in the cloud to be sure that their data is confidential and the queries being requested are the queries being computed.

Chapter 5

System Overview

The *client* refers to the user of Opaque, and is therefore trusted. We define trust from the point of view of the client.

The *server* refers to the cloud provider, which provides the machines which the client outsources computation to. We will use the term "server" also to refer to all of its machines and the machines' operating systems etc. outside the enclaves. The server is untrusted.

The *enclave* refers to the trusted execution environment/hardware enclave/Intel SGX hardware running inside the untrusted host machines in the cloud. Computation done exclusively by the enclave is considered trusted. An *ecall* refers to a specific function done by an enclave worker. The computation done by an ecall is trusted, since it is all executed by the enclave and enclave code.

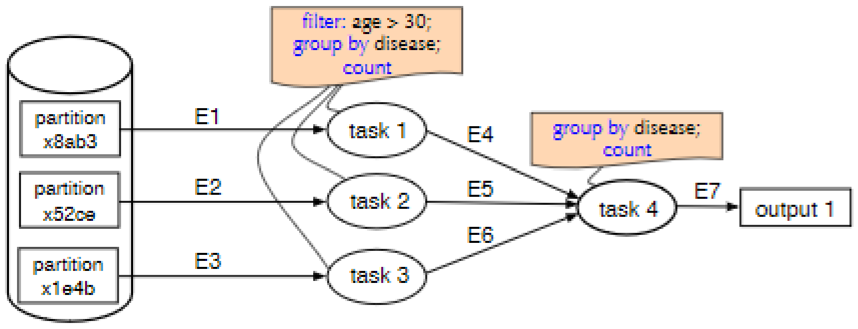
Opaque follows a client-server model. The client submits encrypted data to the server in the cloud, which contains a hardware enclave that performs Spark SQL computation on the data. The enclave then encrypts the result and sends it back to the client.

The two entities in Spark important to our discussion are the query planner (Catalyst) and the job driver. Given a query, the query planner computes a task DAG and gives it to the job driver for task delegation.

The way these entities are allocated among the client and server are important to a discussion about security. In particular, the client runs a query planner locally. The server runs both the query planner and task scheduler outside the enclave. The entities at the server are untrusted and must somehow be verified.

The main idea behind verifying that the task DAG was executed correctly is to reconstruct the flow of execution into a DAG after the result is returned to the client. The client then compares this "executed DAG" to the DAG computed by the query planner run locally at the client. If they are the same, then the flow of information had not been tampered with. If they are not, then the job driver did not delegate workers appropriately, and the flow of information has been changed in some way.

Figure 5.1: Opaque [14] Figure 3



The challenge in this approach is thinking about how to reconstruct this graph without trusting the server, which does the job orchestration. At a high level, this is done by carefully attaching logs to the encrypted output of each enclave worker, whose individual results are tagged with a cryptographic primitive called a MAC function. The MAC function takes in a secret key as input and ensures that only the enclaves, which hold a secret key shared with the client, are able to generate tags, and therefore that the logs are able to be trusted. The logs are like breadcrumbs that serve to reconstruct the flow of execution from start to finish. They hold the server accountable and ensure that tampering with the task DAG does not go undetected.

Chapter 6

System Design

To implement computation integrity, we contributed two main pieces to Opaque.

The first is the logging system mentioned in the System Overview section above. This change allows the client to effectively "see into the past" and reconstruct the flow of information that occurred when the query was executed in the cloud. This approach relies on a cryptographic primitive called a MAC function, which provides trusted "breadcrumbs" we can use to backtrack from the final output and recreate our task DAG.

The second piece is a new Scala module in Opaque's client code called the *Job Verification Engine*. The Job Verification Engine is responsible for executing various graph operations in order to verify the integrity of each query based on the logs generated by the logging system.

Each Spark worker running inside an enclave produces outputs of encrypted data that it pushes back to the host memory for further downstream processing, and can send information to as many other worker nodes as there are data partitions. Recall that Spark divides data into a specified number of partitions in order to parallelize workloads.

6.1 Breaking Down an Opaque Query

To get some more context about these DAGs and all the objects involved, let's look at an example of a typical Opaque query and break down how it can turn into a DAG of ecalls. For the sake of simplicity, assume a single data partition.

Query: `dataframe.sort($"foo").limit(10)`

(Equivalent SQL: `SELECT foo FROM dataframe ORDER BY foo ASC LIMIT 10`)

Plan (Operators): `EncryptedProject` → `EncryptedSort` → `EncryptedLocalLimit` → `EncryptedGlobalLimit`

These operators are the ones chosen by Opaque's query optimizer, to be executed in this order. The reader can consider this chain of operators the logical steps Opaque needs to execute to compute the query. Each operator is implemented by one or more ecalls, which are singular units of trusted computation done by an enclave worker. The plan of operators can also then be viewed in the context of their underlying ecalls:

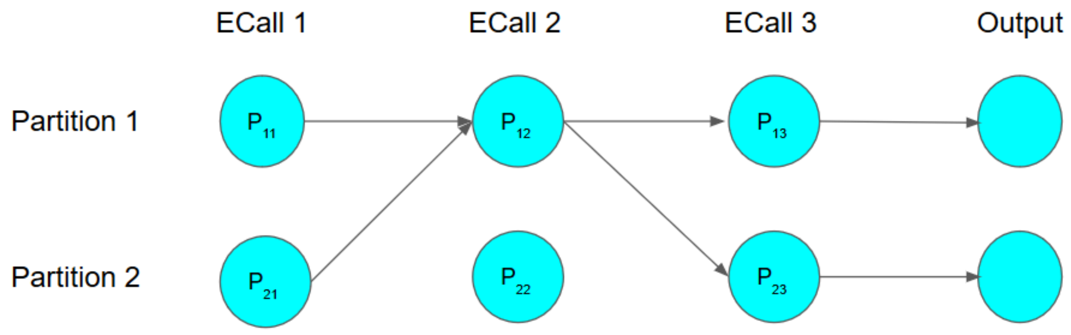
Plan (Ecalls): `project` → `externalSort` → `limitReturnRows` → `countRowsPerPartition` → `computeNumRowsPerPartition` → `limitReturnRows`

In the next section, we examine a more complex imaginary query which incorporates multiple data partitions and visualize an example DAG that might be produced by the query optimizer. As we do so, one should keep in mind the above process of breaking down a query into its operators and then its ecalls to contextualize the post verification computation.

6.2 Logging

Consider an Opaque job that splits the data into 2 partitions and performs 3 ecalls. All output from the first ecall goes to partition 1 (perhaps an aggregation operation), output from the second ecall is broadcast to all partitions (perhaps an operator done on each group of a group by operation), and output from the third ecall stays in the same partition. In visual form, this job can be represented as follows:

Figure 6.1: Task DAG of query broken into ecalls and data partitions



We want to ensure that if the Spark Catalyst query planner outputs this DAG for the imaginary query described above, the job driver run by the cloud executes this exact task DAG.

Towards this effort, we modified the enclave code to output an additional object we call *Crumbs* along with its encrypted data. The flatbuffer schema for a Crumb object is shown below:

```
table Crumb {
    ecall:int; // Ecall executed
    input_macs:[ubyte];
    num_input_macs:int; // Number of input_macs
    all_outputs_mac:[ubyte]; // MAC over all outputs of ecall
    log_mac:[ubyte]; // MAC over the LogEntryChain
}
```

A `Crumb` object tracks metadata about a node after it is executed, such as the ecall (specific enclave function) performed at the node, as well as the node's inputs and outputs.

A list of `Crumb`s called the `LogEntryChain` is maintained for each data partition and appended to as ecalls are executed by workers allocated to each partition. At the end of the query, k lists of `Crumb`s are filled, where k is the number of data partitions, and the sum of the Crumbs over all the lists is equal to the number of Spark workers involved in the query.

Consider for example the node P_{12} in Figure 6.1. The `Crumb` object corresponding to this node would include the ecall field noting that "ECall 2" was executed. It would list two MACs in the `input_macs` field, since it takes input from node P_{11} and node P_{21}. Its `all_outputs_mac` would be a MAC over the outputs it sent to nodes P_{13} and P_{23}. Note that both P_{13} and P_{23} would include P_{12}'s `all_outputs_mac` as data in their `input_macs` field.

All objects involved in logging are MAC'd by the enclave workers at each step before being output along with the encrypted data, ensuring that the logs themselves cannot be tampered with, spoofed, or dropped by the server.

6.3 Job Verification Engine

The Job Verification Engine is the module that the client uses to do computation integrity verification. Its job is to build the "Executed DAG" from the `LogEntryChain`s, "Expected DAG" from Spark metadata, and then compare them.

At a high level, the "Executed DAG" is built by processing each Crumb object from the `LogEntryChain`s, and recreating edges based on matching `input_macs` and `all_outputs_mac` fields. For example, if node 1 sent data to node 2, node 1's `all_outputs_mac` will be one of the entries in node 2's `input_macs` list. Using this fact, it is just a matter of iterating through all the `Crumb`s to match all nodes to their "parents". Once there is an edge between each node and its parents, the graph is complete.

After reconstructing the "Executed DAG", the Job Verification Engine must compute the "Expected DAG" from the query plan. The client runs a copy of Spark Catalyst query planner for this purpose. Spark can output a DAG of Spark operators that will be used in the plan. The Job Verification Engine transforms this graph into a finer granularity DAG of ecalls in linear time; we call this the "Expected DAG".

After both graphs are constructed, the Job Verification Engine checks that the DAGs are the same. This operation involves checking the labeled nodes for their ecall identifiers in both the expected and executed DAGs, and then checking that the edges associated with each node are the same. This can be done in linear time in the number of nodes and edges.

Chapter 7

Implementation

Our changes made to Opaque to implement computation integrity in its current state include around 600 lines of Scala and 200 lines of C++. See the Opaque codebase on the MC^2 GitHub page [15] for more details on the code blocks referenced in this section.

7.1 Reconstruction of Executed Flow

We modified enclave code to create the logs used for reconstruction of the flow of data partitions between ecall workers.

In `FlatbuffersWriters.cpp`, we hooked into the internal memory writing mechanism of Opaque to ensure that when it outputs encrypted results of Spark operations to non-enclave memory, it additionally computes a MAC on the output, as well as outputting all the other logging information stored in the `Crumb` object. This mechanism can be toggled by the developer of new Opaque operations, since not all encrypted output from an enclave necessarily needs to be tracked in the task DAG, such as those from intermediary functions.

In `FlatbuffersReaders.cpp`, successor enclave workers ensure that no rows were dropped by the job driver from their respective predecessor enclave workers, and make sure the MACs output by predecessor nodes match the data.

We made minor code refactoring to integrate these changes in various other parts of the codebase including `Enclave.cpp` and `EnclaveContext.h`.

Post verification happens on the client side in `JobVerificationEngine.scala`. As mentioned before, a `LogEntryChain` object containing a list of `Crumb` objects is maintained per data partition. These `LogEntryChain`s are returned as part of the final output and can be processed by the Job Verification Engine running on the client. Once the Crumbs are given to the engine, it iterates through the LogEntryChain, and while doing so, populates a hash table mapping unique `all_outputs_mac` to `JobNode` objects. A `JobNode` is the Scala representation of an enclave worker; in other words, in the diagram in Section 6.2, each P_{ij} represents a job node, and as a result, there will be a one to one relationship between a `Crumb` object and a `JobNode` object since they represent the same thing. Iterating through the `JobNodes` again allows us to match `all_outputs_mac` fields with `input_macs` fields to recreate edges between the nodes, constructing a DAG representing the transfer of information that transpired in the query from node to node.

7.2 Construction of Expected Flow

Construction of expected flow happens in `JobVerificationEngine.scala`.

By running a copy of Spark Catalyst with Opaque's rules on the client side, the client is able to construct the task DAG that it expects the server to execute. This is done by accessing the Spark DataFrame object's `queryExecution.executedPlan` field, which returns a recursive object representing a tree of Spark operators. Some minor additional graph processing transforms the tree of Spark operators into a tree of Opaque operators.

After retrieving this graph of operators, we need to then convert it into a graph of ecalls. This is done in the `generateJobNodes` method, which contains a cascading if else/switch case denoting an explicit mapping of each supported Opaque operator to the ecalls that implement the operator. The following code snippet from this function illustrates the logic for a few Spark operators.

```
if (operatorName == "EncryptedSort" && numPartitions == 1) {
  expectedEcalls.append("externalSort")
}
...
} else if (operatorName == "EncryptedGlobalLimit") {
  expectedEcalls.append("countRowsPerPartition",
                        "computeNumRowsPerPartition", "limitReturnRows")
}
...
```

In the above code, the `EncryptedSort` Spark operator is transformed into the one ecall that implements it, `externalSort`. The `EncryptedGlobalLimit` Spark operator is transformed into three ecalls that implement it, in the order they are supposed to be run by the server - `countRowsPerPartition`, `computeNumRowsPerPartition`, and `limitReturnRows`.

After converting the graph into a DAG of ecalls, we need to add edges based on how the ecalls communicate information downstream. This is done in the `linkEcalls` method. The following code snippet from this function illustrates some of the different kinds of logic used for various ecalls.

```
...
} else if (ecall == "externalSort") {
  // Send to same partition for the next worker
  for (i <- 0 until numPartitions) {
    parentEcalls(i).addOutgoingNeighbor(childEcalls(i))
  }
} else if (ecall == "partitionForSort") {
  // All to all shuffle
  for (i <- 0 until numPartitions) {
    for (j <- 0 until numPartitions) {
      parentEcalls(i).addOutgoingNeighbor(childEcalls(j))
    }
  }
...
} else if (ecall == "computeNumRowsPerPartition") {
  // Broadcast from one partition (assumed to be partition 0) to all partitions
  for (i <- 0 until numPartitions) {
    parentEcalls(0).addOutgoingNeighbor(childEcalls(i))
  }
...
```

In this code snippet, `parentEcalls` and `childEcalls` are lists of `JobNodes`, where each `JobNode` in the list represents a specific enclave worker performing an ecall on one data partition, and the `parentEcalls` list contains the nodes for the ecall that happens before the ecall in `childEcalls`. This method is necessary to define how ecalls communicate with one another. For example, while the `externalSort` ecall sends data from partition x to partition x in the next ecall, the `partitionForSort` ecall shuffles data in an all-to-all manner as shown in the code snippet above. These differences in ecall behavior need to be defined explicitly in the Job Verification Engine by a programmer with knowledge of how the ecall works in order for the integrity module to properly construct an the "expected flow."

7.3 Comparing DAGs

Once both DAGs are constructed, verifying integrity is a matter of comparing the DAGs and checking if they are equal, or *isomorphic*. Two DAGs are isomorphic if there exists a mapping f between the vertices of DAG1 to the vertices of DAG2 such that if any two vertices A and B in DAG1 are adjacent, $f(A)$ and $f(B)$ are adjacent in DAG2. If the "Actual

DAG" and the "Expected DAG" are isomorphic, then we can be sure that the scheduler did not tamper with the flow of execution.

In our case, the nodes are labeled with ecall identifiers, and thus the problem is easier than the DAG isomorphism problem, which does not have node labelings. With the node labelings, we can first check the list of nodes in both the expected and executed DAGs. If they match and are the same, then for each node and its edges in the expected DAG, we can check the associated node in the executed DAG to check whether the edges are also equal. An edge is equal if the parent and child have the same ecall identifier.

7.4 Usage

The Job Verification Engine exposes a small interface of methods to use in the wider client application. Namely, to verify a job, the user calls `resetForNextJob` to flush the internal state of the JobVerificationEngine, and then after executing a query, calls the `verify` function on the query's dataframe object to verify the integrity of the query.

Chapter 8

Evaluation

8.1 Theoretical Analysis

The additional work required for verifying query integrity does not introduce any major asymptotic penalties to Opaque's operations. During runtime, the only cost is the cryptography done on each encrypted output. This implies performance and memory penalties at most linear in the size of the output that Opaque produced before integrity was implemented. The post verification step is also relatively cheap; it is a one time cost that is able to be performed offline.

Diving deep into asymptotic analysis, let P be the number of data partitions selected by Opaque's query planner. Let E be the number of different types of ecalls executed. Then, the number of Opaque workers is $O(PE)$, since at most P instances of each ecall are executed. Let N be the size in bytes of the largest output produced by any one Opaque worker, without accounting for integrity. The runtime overhead is overwhelmingly caused by the cryptography done to compute and verify MACs. The MAC function used is the HMAC-SHA256 offering of the mbedtls C++ library, which runs in time scaling linearly with the input size. Since a fixed number of MACs are computed over every output of every node, and the same number of MACs are verified at the next, each node performs $O(N)$ work towards this purpose of computing and checking MACs (with a small constant factor), for a total of $O(PEN)$ work over the entirety of the query. For storage, the MAC function outputs a fixed number of bytes no matter the input length, and so the factor of N is removed. The memory overhead thus only scales with the number of enclave operations and data partitions for an asymptotic cost of $O(PE)$. Both of these overheads have small constant factors due to the nature of the cryptography being used. In conclusion, the runtime overhead in both time and space complexity scale linearly with the system at large and incur small penalties relative to the runtime before integrity checks were done.

Post verification can be done offline and incurs a one time cost per query, and so is arguably even less impactful than the runtime penalties. Post verification is concerned with graph processing and has little to do with the actual output produced at each node, which has been verified at runtime. There are three steps to post verification: reconstructing the executed query DAG, constructing the expected query DAG, and checking for equality between the two. Constructing the two DAGs takes time linear in the number of enclave nodes, which is at most the product of the number of data partitions P and the total number of ecalls made to the enclave E, (which scales linearly with the number of Spark operators invoked by the query planner). Therefore, for both runtime and memory, DAG construction incurs a cost of $O(PE)$. Because the nodes are labeled with ecall identifiers, checking to see if the DAGs are equal is done in a linear traversal of both DAGs and thus is also computed in linear time.

8.2 Performance in Practice

To test performance, we benchmarked the system using two of Opaque's "Big Data Benchmarks." The schema and queries can be found on the GitHub repository but are listed here for convenience.

Ranking Benchmark

Schema

```
StructType(Seq(
    StructField("pageURL", StringType),
    StructField("pageRank", IntegerType),
    StructField("avgDuration", IntegerType)))
 )
```

The ranking dataset contains 1,200 rows.

Query

```
rankingsDF.filter($"pageRank" > 1000)
```

Results

Looking at table 8.1, for a single partition, Opaque (including logging overhead) incurs a 3.3x time penalty, and post verification incurs an additional 143 milliseconds, or an additional 1.75x time penalty compared to the baseline. For 5 partitions, Opaque (including logging

Table 8.1: Time Taken for Ranking Benchmark (ms)

	Spark Baseline	Opaque (With Logging)	Opaque (With Logging and Post Verification)
1 Partition	186	628	771
5 Partitions	251	715	724

overhead) incurs a 2.8x time penalty, with a similar low flat penalty when adding post verification.

User Visits Benchmark

Schema

```
StructType(Seq(
    StructField("sourceIP", StringType),
    StructField("destURL", StringType),
    StructField("visitDate", DateType),
    StructField("adRevenue", FloatType),
    StructField("userAgent", StringType),
    StructField("countryCode", StringType),
    StructField("languageCode", StringType),
    StructField("searchWord", StringType),
    StructField("duration", IntegerType)))
)
```

The user visits dataset contains 10,000 rows.

Query

```
uservisitsDF
    .select(substring($"sourceIP", 0, 8).as("sourceIPSubstr"), $"adRevenue")
    .groupBy($"sourceIPSubstr").sum("adRevenue")
```

Results

Table 8.2: Time Taken for User Visits Benchmark (ms)

	Spark Baseline	Opaque (With Logging)	Opaque (With Logging and Post Verification)
1 Partition	477	6400	7346
5 Partitions	475	20974	21820

Looking at table 8.2, for a single partition, Opaque (including logging overhead) incurs a 13x time penalty, and post verification incurs an additional 946 milliseconds, or an additional 2x time penalty. For 5 partitions, Opaque (including logging overhead) incurs a 40x time penalty, with a 1000 millisecond penalty when adding post verification.

Chapter 9

Limitations and Future Work

9.1 Implementation Pitfalls

Although the techniques described in this work can be generalized at a high level to other similarly designed systems, the actual implementation is brittle; the job verification engine's logic works on a case-by-case basis for each Spark operator and could change depending on the ecalls used. Thus, a programmer would have to keep this in mind when extending Opaque to support any new user defined functions or adding additional Spark operators.

9.2 Future Work

In terms of functionality, this work has fully supported query integrity verification for Opaque. Future work could take two different directions: it could refactor code implementing the job verification logic for further time and memory gains, or improve Opaque in an orthogonal manner. For instance, the original paper's framing aimed to support data obliviousness to prevent side channel attacks on Intel SGX enclaves. As of right now, this functionality is not merged in the master branch of the Opaque code base, and further work needs to be done to do so while integrating with the recent developments in Opaque.

Chapter 10

Conclusion

In this report, we explored the integrity limitations of the open source software of Opaque, outlined new design changes, and discussed an implementation of computational integrity that adds a needed layer of security to Opaque. This work reflects a core component of the secure analytics platform, and ensures to the client that the untrusted driver located in the cloud schedules tasks among Opaque workers correctly, and in addition, does not tamper with encrypted data passed between Opaque workers. While the implementation is Opaque specific, the high level ideas are widely applicable to any distributed secure analytics platform that operates in a similar fashion to Opaque.

Bibliography

[1] Einar Mykletun, Maithili Narasimha, and Gene Tsudik. “Authentication and Integrity in Outsourced Databases”. In: (2006), pp. 107–138.

[2] Wei-Shinn Ku et al. “A query integrity assurance scheme for accessing outsourced spatial databases”. In: *GeoInformatica* (2012), pp. 97–124.

[3] Qingji Zheng, Shouhuai Xu, and Giuseppe Ateniese. “Efficient Query Integrity for Outsourced Dynamic Databases”. In: Association for Computing Machinery, 2012, pp. 71–82.

[4] Andrew Baumann, Marcus Peinado, and Galen Hunt. “Shielding Applications from an Untrusted Cloud with Haven”. In: USENIX Association, 2014, pp. 267–283.

[5] Felix Schuster et al. “VC3: Trustworthy Data Analytics in the Cloud Using SGX”. In: IEEE Computer Society, 2015, pp. 38–54.

[6] Sergei Arnautov et al. “SCONE: Secure Linux Containers with Intel SGX”. In: USENIX Association, 2016, pp. 689–703.

[7] Tyler Hunt et al. “Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data”. In: USENIX Association, 2016, pp. 533–549.

[8] Matthias Schunter. “Intel Software Guard Extensions: Introduction and Open Research Challenges”. In: 2016, p. 1.

[9] Matei Zaharia et al. “Apache Spark: A Unified Engine for Big Data Processing”. In: (2016), pp. 56–65.

[10] Yeongjin Jang et al. “SGX-Bomb: Locking Down the Processor via Rowhammer Attack”. In: 2017.

[11] Shweta Shinde et al. “Panoply: Low-TCB Linux Applications with SGX Enclaves”. In: Internet Society, 2017.

[12] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. “CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management”. In: USENIX Association, 2017, pp. 1057–1074.

[13] Chia-che Tsai, Donald E. Porter, and Mona Vij. “Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX”. In: USENIX Association, 2017, pp. 645–658. ISBN: 978-1-931971-38-6.

[14] Wenting Zheng et al. “Opaque: An Oblivious and Encrypted Distributed Analytics Platform”. In: USENIX Association, 2017, pp. 283–298.

[15] RISELab. *MC2 Github*. `http://github.com/mc2-project/opaque`. 2019.

[16] Kit Murdock et al. “Plundervolt: Software-based fault injection attacks against intel SGX”. In: Institute of Electrical and Electronics Engineers, May 2020, pp. 1466–1482.

[17] Rishabh Poddar et al. “Visor: Privacy-Preserving Video Analytics as a Cloud Service”. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020, pp. 1039–1056.