Supporting Multiple Clients in Opaque

Eric Feng

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-40
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-40.html

May 11, 2021

Supporting Multiple Clients in Opaque

by Eric Feng

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:

[illegible]

Professor Raluca Ada Popa
Research Advisor

May 10, 2021

(Date)

* * * * * * *

Joseph E. Gonzalez

Professor Joseph Gonzalez
Second Reader

5/10/21

(Date)

Supporting Multiple Clients in Opaque

by

Eric Feng

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

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 Joseph Gonzalez

Spring 2021

Supporting Multiple Clients in Opaque

To my parents and my sister

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my advisor Raluca Ada Popa for giving this opportunity and her continued guidance in growing my ability as a researcher and engineer. I'd also like to thank Wenting Zheng and Chester Leung for their mentorship and direction through the project. It's been amazing developing with both of you.

A shout out to Andrew Law for finishing the MS program with me. You've been a great friend and source of inspiration. And finally, thank you to my family who believed in me when I didn't.

Abstract

Supporting Multiple Clients in Opaque

by

Eric Feng

Master of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Raluca Ada Popa, Chair

Opaque builds upon Apache Spark and utilizes Intel SGX Enclaves in order to perform oblivious, encrypted database operations. Unlike previous encrypted systems that focus on either the general database or were application specific, Opaque aims for the best of both worlds by optimizing the query layer, thus obtaining efficiency while still maintaining decent adaptability. However, the implementation of Opaque is not complete and is lacking several important security features such as obliviousness and multiparty support. For my Master's project, I implemented the multiparty aspect of Opaque.

Chapter 1

Introduction

The cloud is becoming a mainstream mechanism for companies and applications to take advantage of efficient computing and storage. However, due to inherent nature of storing sensitive information on an external service, there are a variety of security concerns when interacting with the cloud. Some of these concerns can be alleviated using encrypted databases and/or trusted executions environments [6] such as enclaves. However, recent works [4, 5, 12] have shown that these solutions are vulnerable to side channel attacks where an adversary may be able to deduce the contents of the database through access patterns. To be clear, these 'access pattern leakage' weaknesses affect both encrypted databases and enclaves.

In order to resolve these issues, access patterns must be hidden, and there have been a variety of proposed solutions [3] that implement this property of 'obliviousness' with large overheads in latency. Opaque [13] is one of the solutions that provides oblivious and confidential operations, and it aims to address the latency issue present in encrypted database systems by optimizing at the query layer. More specifically, Opaque acts as a wrapper around Apache Spark and extends the software to utilize enclaves. This is significant as previous systems, to the extent of my knowledge, primarily focused on either a database oblivious protocol or an application specific one; the former type has terrible latencies while the latter is application specific. In other words, by not focusing on the general database level or application specific level, Opaque obtains a decent balance between the two with the efficiency of application-specific implementations, but also the generalizability of the database-orientated designs.

However, the current Opaque implementation is incomplete, and several key features are missing. Among the aspects that are not implemented, most relevantly, is multiparty support. Namely, the Opaque implementation only supports a single client interaction at a time. For my Master's project, we implemented the multiparty aspect of Opaque.

The main challenges we faced in implementing multiparty was removing the inherent trust in the client present in the single client model and ensuring the resulting implementation was compatible with Apache Spark. Previously, the client was able to provide all the secret keys for enclave encryption, and it was also able to contain the Spark Driver that contained all the variables (see $2.2 for more information on Apache Spark). These actions are no longer possible in the new threat model $5.

The two solutions to address the trust issue within the context of Apache Spark, and my main contributions, are the following:

- We introduce a middle man that contains the single Spark Driver and coordinates all client queries into the same spark cluster.
- We perform a modified version of local attestation process between the enclaves so that they can determine a shared secret key for encrypted operations.

Chapter 2

Background

2.1 Hardware Enclaves

Enclaves are a trusted runtime environment. More specifically, they are regions of private memory that cannot be modified or viewed by any process running outside of the enclave. In other words, due to these confidentiality and integrity guarantees, we can run commands such that an adversary would not be able to discern sensitive, underlying data (not including side-channel attacks).

The process by which a third-party is able to establish trust within an enclave is called attestation. There are two types of attestation: local and remote. In local attestation, two enclaves are certifying to each other their validity. In remote attestation, a client is trying to verify the validity of some remote enclave. In both situations, the enclave being doubted generates a signature report signed by a trusted entity. The report contains proof that the enclave is indeed what it says it is, and the code it is running is valid. In the multiparty implementation, we also include a public key in the report to support further communication.

2.2 Apache Spark

Opaque is built as a wrapper around Apache Spark, a framework for quickly performing database operations. Spark handles these operations across a cluster containing a single master node and one or more worker nodes. When provided a query, a Spark driver on the master node determines a Directed Acyclical Graph (DAG) that dictates an efficient workflow for executing that query using the worker nodes. That workflow is then handled and distributed by a cluster manager provided by Spark. The role Opaque plays in this is through a modification of the engine that generates the DAG to use encrypted operations.

When starting an application with Apache Spark, the application connects to the master node in the cluster. The application contains a driver program, as mentioned above, which

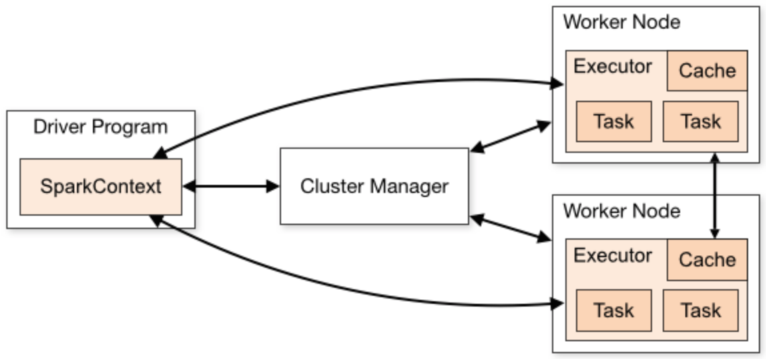


Figure 2.1: Spark Cluster Diagram [9]

in turn contains references to all the relevant variables, data sets, etc. This is significant as different Spark drivers cannot share variables or references unless they store the information in an external storage location.

Chapter 3

Related Work

Interacting with encrypted data is an active area of research. Here are some of the related works in the field.

CryptDB [8] and Monomi [10]

CryptDB and a follow-up work, Monomi, use layered encryption with each layer of encryption corresponding to some functionality and level of leakage. The "outer-most" layers of encryption are the most secure, and the "inner" layers are less secure but allow more operations. Layers are decrypted as needed, and this mechanism is controlled by a trusted proxy that also intercepts plaintext queries and transforms them into usable forms. Multiparty functionality is supported through chain encryption keys. In Monomi, the system also takes advantage of the trusted client and can send data to the client for final processing.

Conclave [11]

While not strictly a encrypted database system, Conclave uses a similar logic as Opaque when optimizing for oblivious Multiparty Computation (MPC) in intercepting queries and optimizing them. More specifically, Conclave partitions data into public and private data, so query optimization takes the form of deciding what data needs to be processed in MPC and what will not be. Overall, this splitting of the data and the query reduces the amount of data that needs to be performed in MPC and increases efficiency.

ObliDB [3]

ObliDB provides general, oblivious database workloads and follows the original Opaque paper. It takes advantage of a query processor that can dynamically choose between different oblivious implementations of the same operator to obtain competitive performance. It also provides two database formats (flat or tree-based).

Prochlo [2]

While Prochlo supports obtaining results from an encrypted database, it mainly focuses on how multiple clients can contribute data. Three of the main techniques in Prochlo include enclaves (Intel SGX), batching, and a protocol the authors call Encode, Shuffle, Analyze (ESA) that encrypts, permutes, and ensures that the data is encrypted when stored in a database. The last step of the ESA protocol, Analyze, obtains the values in a database and adds differential privacy to the results so that they can be used publically.

Cipherbase [1]

Cipherbase is an encrypted database system that supports SQL queries and provides efficiency through a client proxy and custom hardware. The proxy for each client intercepts queries from the client to the database and transforming it as appropriate. More specifically, the proxy is responsible for encrypting or decrypting queries as well as ensuring the resulting computations are efficient. The custom hardware, created from FPGAs, acts as a pseudo-enclave and processes the encrypted data from the main database.

Oblix [7]

Oblix introduces efficient, oblivious encrypted search through combining an ORAM (Oblivious RAM) client and server (based on PathORAM client and servers) onto the same host and ensuring that all memory accesses are oblivious. More specifically, the client is placed in an enclave while the server is allowed to be in untrusted memory. Client internal memory accesses are also made oblivious and optimized through the computation of a fullness map for all buckets that allows for reading and writing only certain blocks. The authors name the state of oblivious client internal and external memory as doubly-oblivious.

Chapter 4

System Overview

4.1 System Architecture

In this section, we discuss the multiparty implementation and the primary actors in the system. They are (i) the clients, (ii) the Spark cluster, and (iii) the intermediate RPC listener.

Clients

A client is a party using Opaque to process data. They send queries to be executed through the RPC listener.

Each client trusts themselves, and data can be stored unencrypted on the client without being compromised. Additionally, they are able to decrypt results that are encrypted with an owned secret key.

Spark Cluster

The spark cluster, or what would nominally be considered the cloud, is where the queries are executed. The cluster is a collection of worker nodes and a single master node - every worker node has hardware enclave support. The organization of the spark cluster is detailed more in $2.2.

Query execution is dictated by the DAG physical plan generated by the Spark driver on the master node. Worker nodes have both untrusted (the host) and trusted memory (the enclave), and any communication reaching the enclave must first pass through the host. The untrusted memory can be freely tampered with and viewed, and the enclaves themselves do not establish TLS connections or other forms of direct communication with each other. However, communication remains confidential as all input and output to an enclave is encrypted under a shared secret that is generated through a modified local attestation.

As mentioned previously, the enclave is trusted and confidential, but the information inside can be leaked through side-channel attacks. Access pattern leakage can be mitigated through oblivious operators, but that is out of scope for this report.

RPC Listener

The RPC listener connects the clients and the Spark cluster and acts as a man in the middle actor. Queries are communicated from the clients to the Spark cluster, and encrypted results are communicated from the Spark cluster to the clients. The RPC listener is not trusted, and it is not able to decrypt any of the messages that pass through it.

The purpose of the intermediate orchestrator is to ensure that all clients share a central Spark driver and thus share data and memory for collaborative computing. In other words, the reason why every client cannot directly communicate with the master node on their own is because in doing so, they automatically create their own Spark driver (this is behavior of Apache Spark) and thus cannot share data.

4.2 General Workflow

This section details the various steps involved for a clients to send queries and obtain and decrypt results.

1. The RPC listener starts and connects to the Spark cluster through the master node.

2. The clients connect to the RPC Listener and, through it, performs remote attestation with the enclaves in the Spark worker nodes. Through running remote attestation, the clients are able to pass a personal shared secret key sk_i to the enclaves for result encryption.

3. The clients send queries to the RPC listener which in turn communicates them to the Spark cluster. The encrypted results are stored in a file(s) on the VM that the RPC listener is located on.

4. The clients ask for a post-verification of the query computation to ensure that the Spark driver did not tamper with the generated Spark DAG, force the enclaves to execute the wrong query, and intentionally create bad results. Post verification is out of scope for this report.

5. The RPC listener, after post-verification succeeds, reads the encrypted results from the files from step 3 and uses the enclaves on the worker nodes to re-encrypt the results under the the clients' individual keys.

6. The RPC listener sends the encrypted ciphers to the corresponding client. The client is able to then decrypt the result using their shared secret key sk_i.

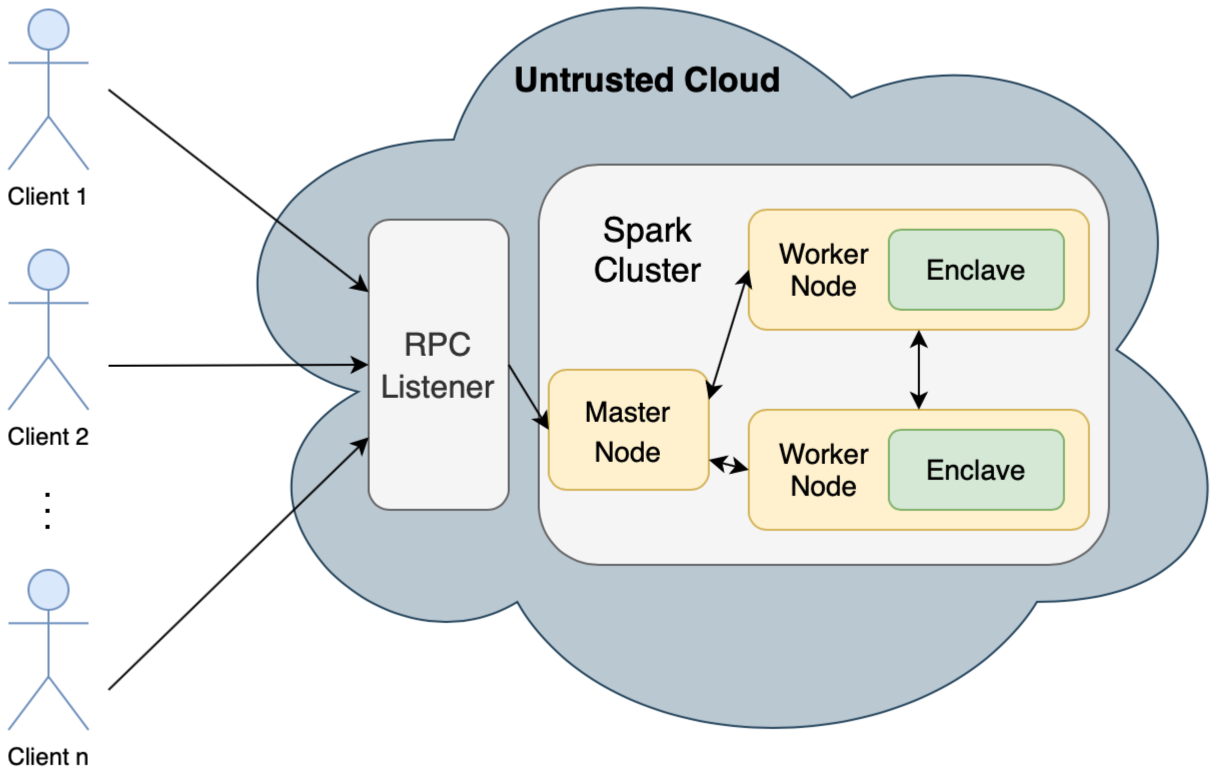


Figure 4.1: Clients send queries to the RPC listener. The RPC listener communicates to the Spark master node which takes the query, generates the DAG, and communicates with the worker nodes to execute the physical plan. Arrows only show one direction for simplicity.

Chapter 5

Threat Model

As mentioned in the previous section, there are three main actors in Opaque Multiparty: the clients, the Spark cluster, and the intermediate RPC listener. For the purpose of discussing the threat model, however, the Spark cluster and the intermediate RPC listener will be grouped into the singular actor ‘the cloud’.

Client Threat Model

A client does not trust any actor besides itself. Unencrypted client data is considered uncompromised only on the client itself and within the trusted enclaves. Malicious clients can collude with each other and the cloud and have super-user privileges, but because query results and data are always transmitted encrypted under either the client’s individual secret key or the enclaves’ shared secret key, no data is leaked to malicious actors.

Cloud Threat Model

The RPC listener and the Spark cluster combined constitute the cloud for Opaque. We assume that the entire cloud, except the enclaves, are compromised. This means that a malicious actor can freely tamper and view insecure host memory, and the clients’ data will maintain confidentiality and integrity.

We do trust the cloud for availability. We believe that this is a reasonable assumption to make as cloud providers are self-interested in ensuring cloud availability for business.

One area of compromise not addressed is the potential for the untrusted Spark driver to generate an incorrect working DAG so that the enclaves execute the wrong physical plan. Post-verification of query execution is out of the scope of this report, but this issue is currently being addressed by another person working on Opaque.

Finally, although we trust the information inside an enclave is confidential, as mentioned previously, an adversary could still compromise and infer the data through side-channel attacks.

Chapter 6

System Design

In this section, we discuss the design of Opaque multiparty. There are three main sections: (i) the initial set up before any communication occurs, (ii) local attestation without direct communication between enclaves, and (iii) establishing communication between the client, the RPC listener, and the Spark cluster.

6.1 Initial Setup

In the initial setup, I describe what each party holds.
Client: Each client holds two values at initialization.

- A 256-bit symmetric key k_i to be transferred to the enclave for encrypting data.
- A certificate that authenticates the client to the enclaves.
- The public key of a trusted entity, used to verify enclave reports in remote attestation.

Spark Cluster: The untrusted host (excluding the enclaves) on both the master node and worker nodes do not handle decrypted data and do not hold any items. The enclave holds two items at start up.

- A 2048-bit RSA public/private keypair (pk_j, sk_j). This key pair is used to communicate with other enclaves in local attestation as well as the clients in remote attestation as explained in sections 6.2 and 6.3.
- The public key of a trusted entity, used to verify other enclaves.

RPC Listener: The RPC listener acts as an intermediate actor in the communication between the clients and the Spark cluster. As such, it does not contain any items.

6.2 Local Enclave Attestation

In order for the enclaves to determine the same shared key, they need to be able to trust one another and communicate. However, in Spark, direct communication between the worker nodes outside of the evaluation of a physical plan isn't supported, and so we have to use the Spark driver as an intermediate actor to relay communications. To be specific, the steps for local attestation are as follows:

1. The Spark driver gathers attestation reports from all the enclaves. These reports are from OpenEnclave's evidence SDK, and they contain the public key of the enclave.
2. The reports are sent to a single, randomly picked enclave e.
3. Enclave e verifies all the attestation reports using the public key of a trusted entity and obtains the public keys of all other enclaves.
4. Enclave e generates a random 256-bit symmetric key and encrypts that key under each of the other enclaves' public keys. All of the ciphers are concatenated into a single message c and returned to the Spark driver.
5. The Spark driver sends c to each of the enclaves.
6. Each enclave goes through the list of ciphers and attempts to decrypt each one. One is guaranteed to decrypt successfully and provide the enclave the shared key.
7. At the end of the process, all enclaves obtain the shared key, and the Spark driver learns no information.

6.3 Communication between Clients and the Spark cluster

Remote Attestation

In order for the client to trust the remote enclaves, they perform remote attestation with them. Because the clients do not communicate directly with the Spark cluster (and therefore the enclaves inside them), they receive the attestation report through the RPC listener. This is okay, because the client can verify and integrity check the report using the public key of a trusted entity. After verification, the client will send back their shared key encrypted with the enclaves' public key. This shared key will be used for future communication between the client and the enclaves.

Spark-shell Sub-process

One of the primary difficulties with enabling multiparty support for Opaque was Sparks inherent limitation on one active spark-context per application. More specifically, if there were no RPC listener, then each client would connect to the spark cluster individually with their own spark context, and they would not be able to communicate and work together. The solution to this problem was to introduce the central shell that would be able to take all client queries and direct them to the same spark context. Hence, the RPC listener instantiates a single spark-shell sub-process and redirects all queries it receives to that sub-process.

Result Encryption and Decryption

To ensure that the results of the query are not known to the driver, the query results remain encrypted until post-verification. More specifically, until the clients actively send a post-verification request, the result will remain encrypted on disk. Once post-verification occurs however, then the result will be read by the driver, sent to the enclave for decryption and re-encryption using client keys, and then sent back to the corresponding clients.

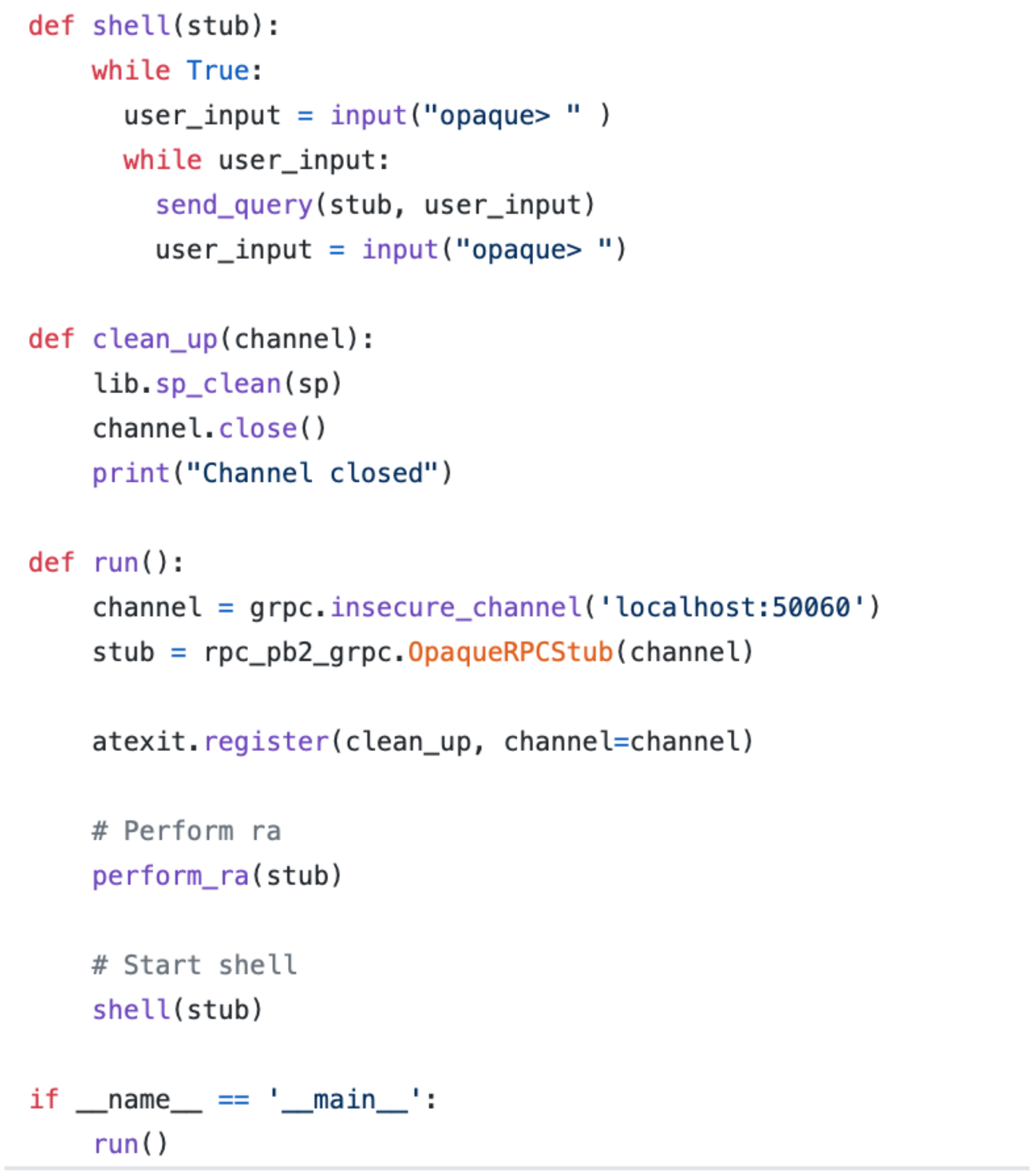
Chapter 7

Implementation

The implementation is built on the original Opaque codebase written in primarily C++ and Scala. In total, the added code was approximately 2000 lines of C++, Scala, and Python code combined.

The Python client and RPC listener use gRPC for communication and call underlying Opaque functions which can be either in C++ or Scala. For client side cryptography, we used OpenSSL, while for enclave cryptography and interfacing we used MbedTLS and Open Enclave respectively.

The multiparty implementation was tested on Intel SGX enclaves on Azure Secure Confidential Computing instances. The code is currently open source at `https://github.com/mc2-project/opaque`.



```python
def shell(stub):
    while True:
      user_input = input("opaque> " )
      while user_input:
        send_query(stub, user_input)
        user_input = input("opaque> ")

def clean_up(channel):
    lib.sp_clean(sp)
    channel.close()
    print("Channel closed")

def run():
    channel = grpc.insecure_channel('localhost:50060')
    stub = rpc_pb2_grpc.OpaqueRPCStub(channel)

    atexit.register(clean_up, channel=channel)

    # Perform ra
    perform_ra(stub)

    # Start shell
    shell(stub)

if __name__ == '__main__':
    run()
```

Figure 7.1: Sample code for client shell. This code contains calls for attestation and sending queries

Chapter 8

Evaluation

8.1 Tests and Setup

In the experiments, we ran two common learning algorithms: logistic regression and K-means. We also ran the first 22 queries of the TPCH benchmark.

Logistic Regression: Provided n points and classifications, logistic regression attempts to match a logistic function to the points in order to obtain a binary classifier.

K-means: Provided a cluster of n points, k-means attempts to partition them into k clusters such that each point is located in the cluster with the closet mean.

TPCH Benchmark: The TPCH benchmark is a common benchmark for any data-processing software and completing it details complete coverage. Opaque does not support all TPCH operations yet, and for this evaluation, I only ran the first 22.

The experiments include three systems: the original Opaque implementation with no division of trust, an Opaque implementation with division of trust and local attestation, and Opaque multiparty (queries being sent over gRPC). For gRPC, we had both the client and the server on the same host.

The experiments were run on Microsoft Azure Confidential Computing Virtual Machines. The specs of the machines were DC4s_V2 machines with 4 vCPUs and 16 GiBs of memory. The spark configurations of the cluster that the tests were run on was 3 executor instances each with 4096 MB and with spark.sql.shuffle.partitions = 3. The configurations of the gRPC server was 4 executor instances and 8 GB total.

8.2 Results

Logistic Regression: We measured the training time on 1000, 10000, 100000, 250000, and 500000 randomly generated classified 10D points. We trained for 5 iterations. K-means: We measured the training time to classify 1000, 10000, 100000, 250000, and 500000 randomly generated 2D points into 3 clusters. TPCH Benchmark: We measured the time it took to complete the first 22 queries of the TPCH benchmark.

Num. Points	Trusted Client (s)	Untrusted Client (s)	gRPC (s)
1000	6	7	7
10000	7.5	9	10
100000	17	27	27
250000	34	52	54
500000	64	99	100

Table 8.1: Times for Logistic Regression

Num. Points	Trusted Client (s)	Untrusted Client (s)	gRPC (s)
1000	6.5	8	8
10000	6.5	21	22
100000	19	125	123
250000	34	296	290
500000	66	498	494

Table 8.2: Times for K-Means

Trusted Client (s)	Untrusted Client (s)	gRPC (s)
126	125	128

Table 8.3: Times for first 22 Queries of TPCH

In general, after removing trust from the client, the more the workload increased, the more encryptions are performed, and the greater the time taken. The TPCH times were the same across all methods as the additional encryption for each query was low. For these benchmarks, transitioning from an untrusted client to communicating through gRPC did not increase latency as the final result in each case was not large and could be transmitted in one trip on the single host (i.e. a single array for both Logistic Regression and K Means). However, we expect that for workloads where the final result is many rows long, the corresponding overhead would increase accordingly.

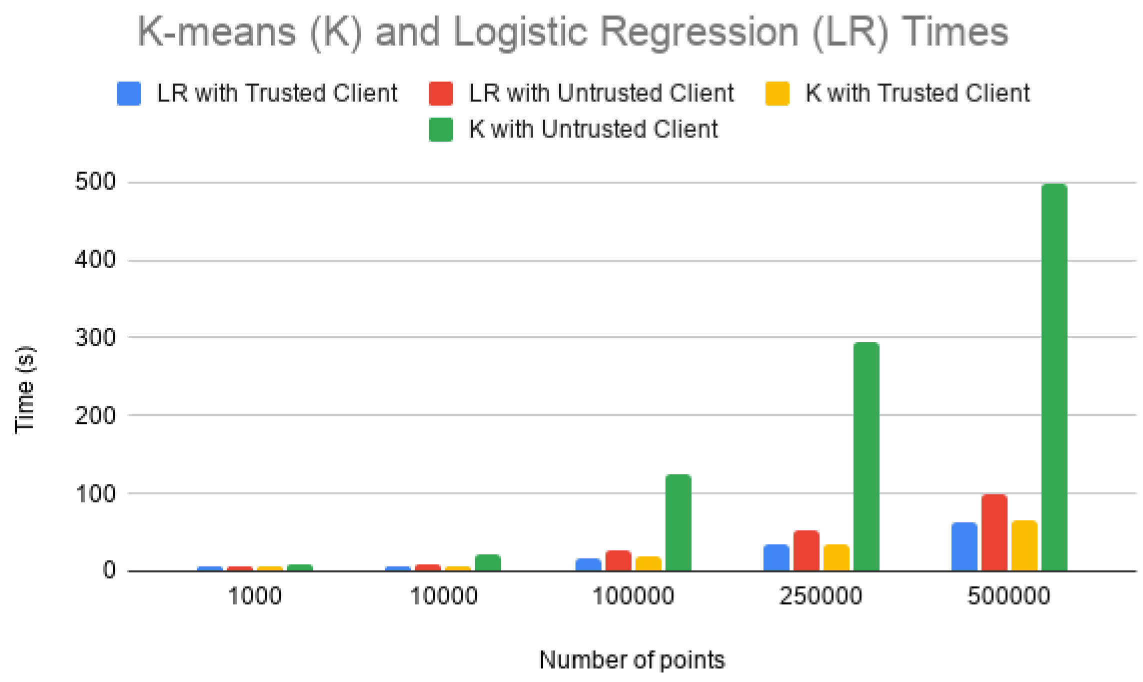


Figure 8.1: The time increases as the amount of data that needs to be encrypted grows.

Chapter 9

Next Steps

The current multiparty implementation creates a client shell for interaction, but the shell is limited in that it can only send single line queries that must return a value. In other words, Scala functions, classes, and objects need to be defined externally and then imported into the shell using an 'import' command. Additionally, the shell has other issues including being unable to use arrow keys to navigate current and previous input. The next immediate step would be to improve the client shell to support these features.

Beyond the client shell improvements, post-verification and client authentication are also currently not supported. The former is important in order to make sure that the Spark driver does not maliciously tamper with the generated DAG and trick the enclaves into executing the wrong query. The latter is important to make sure that only trusted parties are allowed to connect to the spark cluster and process sensitive data. One possible solution to this is to embed the trusted usernames into the enclave on start up and only allow those users (identity would be signed by a trusted entity) to establish an initial connection. Another possible solution is to add a digital signature onto every query and ensure that the enclave checks that the query signer is trusted.

Finally, oblivious operators has also not been implemented yet. This is important in order to ensure that the enclave is not vulnerable to side-channel attacks.

In summary, future work can focus on client shell improvements, the integration of the post-verification engine, authentication of clients, and the implementation of obliviousness.

Chapter 10

Conclusion

For this Master's report, I contributed a design and implemented Opaque multiparty functionality. The implementation involved removing the trust from the client and providing a mechanism for clients to share a Spark driver.

Bibliography

[1] Arvind Arasu et al. "Secure Database-as-a-Service with Cipherbase". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, New York, USA: Association for Computing Machinery, 2013, pp. 1033–1036. ISBN: 9781450320375. DOI: `10.1145/2463676.2467797`. URL: `https://doi.org/10.1145/2463676.2467797`.

[2] Andrea Bittau et al. "Prochlo: Strong Privacy for Analytics in the Crowd". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: Association for Computing Machinery, 2017, pp. 441–459. ISBN: 9781450350853. DOI: `10.1145/3132747.3132769`. URL: `https://doi.org/10.1145/3132747.3132769`.

[3] Saba Eskandarian and Matei Zaharia. *ObliDB: Oblivious Query Processing for Secure Databases*. 2019. arXiv: `1710.00458 [cs.CR]`.

[4] Paul Grubbs, Thomas Ristenpart, and Vitaly Shmatikov. *Why Your Encrypted Database Is Not Secure*. 2017.

[5] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. *Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation.*

[6] Frank McKeen et al. "Innovative Instructions and Software Model for Isolated Execution". In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '13. Tel-Aviv, Israel: Association for Computing Machinery, 2013. ISBN: 9781450321181. DOI: `10.1145/2487726.2488368`. URL: `https://doi.org/10.1145/2487726.2488368`.

[7] Pratyush Mishra et al. "Oblix: An Efficient Oblivious Search Index". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 279–296. DOI: `10.1109/SP.2018.00045`.

[8] 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`.

[9] *Spark Cluster Mode Overview*. URL: `https://spark.apache.org/docs/latest/cluster-overview.html`.

[10] Stephen Tu et al. *Processing analytical queries over encrypted data*. 2013.

[11] Nikolaj Volgushev et al. "Conclave: Secure Multi-Party Computation on Big Data". In: *Proceedings of the Fourteenth EuroSys Conference 2019*. EuroSys '19. Dresden, Germany: Association for Computing Machinery, 2019. ISBN: 9781450362818. DOI: `10.1145/3302424.3303982`. URL: `https://doi.org/10.1145/3302424.3303982`.

[12] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. *Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems*. 2015.

[13] 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`.