AUDITABLE COMPUTATIONS ON (UN)ENCRYPTED GRAPH-STRUCTURED DATA

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Servio Ernesto Palacios Interiano

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2020

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Bharat Bhargava, Chair

Department of Computer Science

Dr. Jeremiah Blocki

Department of Computer Science

Dr. James V. Krogmeier

Department of Electrical and Computer Engineering

Dr. Chunyi Peng

Department of Computer Science

Dr. Xiangyu Zhang

Department of Computer Science

Approved by:

Dr. Clifton Bingham

Head of Graduate Program

Soli Deo Gloria.

In memory of Jesus Castulo Palacios Tosta.

To my wonderful mother, Aura Interiano.

To my beloved wife and son.

ACKNOWLEDGMENTS

I would like to express my appreciation and thanks to my advisor Dr. Bharat Bhargava. My deepest gratitude for his assistance, support, guidance during my Ph.D. I am thankful for allowing me an immense degree of freedom to develop my research agenda.

I want to thank Dr. Blocki, Dr. Krogmeier, Dr. Peng, and Dr. Zhang for serving in my advisory committee and for providing valuable feedback. Also, I would like to thank Dr. Ananth Grama for fascinating discussions and highly challenging problems that motivated the need to pursue the Ph.D. degree.

I dedicate this dissertation to my father, Jesus Palacios, and my mother, Aura Interiano. The significance of their perfect combination of emotional encouragement, continuous support, love, and trust cannot be conveyed in words alone. I want to express my sincerest thanks to my team members, my wife, Bessy, and my son Jesus Alberto. From eating ice cream at midnight to pizza nights on Fridays, together we got through our Ph.D. I am eternally in debt to all the people supporting my family and me, such as my beloved siblings Delfina, Jesus, Jessyka, Victoria, and my in-laws, Maritza, Cristina, and Louis.

I would like to thank the Fulbright program. Through your support, many—otherwise shattered—dreams come true. Similarly, I want to thank the OATS Center (Open AG Technologies and Systems) for giving me the opportunity and support to pursue my ideas. The OATS team provided a rich environment filled with fascinating insights and discussions. In particular, I want to thank Aaron Ault, Dr. Krogmeier, and Dr. Buckmaster.

Finally, this dissertation is dedicated to my kids (nephews and nieces) and all the underrepresented in STEM disciplines. A final thanks go to all my friends, in particular Victor Santos and my friends from the Instituto Central Vicente Caceres.

TABLE OF CONTENTS

Page

Page

Page

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

ABSTRACT

Palacios, Servio Ph.D., Purdue University, August 2020. Auditable Computations on (Un)Encrypted Graph-Structured Data. Major Professor: Bharat Bhargava.

Graph-structured data is pervasive. Modeling large-scale network-structured datasets require graph processing and management systems such as graph databases. Further, the analysis of graph-structured data often necessitates bulk downloads/uploads from/to the cloud or edge nodes. Unfortunately, experience has shown that malicious actors can compromise the confidentiality of highly-sensitive data stored in the cloud or shared nodes, even in an encrypted form. For particular use cases —multi-modal knowledge graphs, electronic health records, finance— network-structured datasets can be highly sensitive and require auditability, authentication, integrity protection, and privacy-preserving computation in a controlled and trusted environment, i.e., the traditional cloud computation is not suitable for these use cases. Similarly, many modern applications utilize a "shared, replicated database" approach to provide accountability and traceability. Those applications often suffer from significant privacy issues because every node in the network can access a copy of relevant contract code and data to guarantee the integrity of transactions and reach consensus, even in the presence of malicious actors.

This dissertation proposes breaking from the traditional cloud computation model, and instead ship certified pre-approved trusted code closer to the data to protect graph-structured data confidentiality. Further, our technique runs in a controlled environment in a trusted data owner node and provides proof of correct code execution. This computation can be audited in the future and provides the building block to automate a variety of real use cases that require preserving data ownership. This project utilizes trusted execution environments (TEEs) but does not rely solely on

TEE's architecture to provide privacy for data and code. We thoughtfully examine the drawbacks of using trusted execution environments in cloud environments. Similarly, we analyze the privacy challenges exposed by the use of blockchain technologies to provide accountability and traceability.

First, we propose AGAPECert, an Auditable, Generalized, Automated, Privacy-Enabling, Certification framework capable of performing auditable computation on private graph-structured data and reporting real-time aggregate certification status without disclosing underlying private graph-structured data. AGAPECert utilizes a novel mix of trusted execution environments, blockchain technologies, and a real-time graph-based API standard to provide automated, oblivious, and auditable certification. This dissertation includes the invention of two core concepts that provide accountability, data provenance, and automation for the certification process: Oblivious Smart Contracts and Private Automated Certifications. Second, we contribute an auditable and integrity-preserving graph processing model called AuditGraph.io. AuditGraph.io utilizes a unique block-based layout and a multi-modal knowledge graph, potentially improving access locality, encryption, and integrity of highly-sensitive graph-structured data. Third, we contribute a unique data store and compute engine that facilitates the analysis and presentation of graph-structured data, i.e., TruenoDB. TruenoDB offers better throughput than the state-of-the-art. Finally, this dissertation proposes integrity-preserving streaming frameworks at the edge of the network with a personalized graph-based object lookup.

1. INTRODUCTION

Graph-structured data is ubiquitous. Modeling and handling large-scale network-structured datasets require graph processing and management systems such as graph databases and pose significant computational challenges. Graphs[1] are data structures used to represent relationships between entities. For particular use cases —multi-modal knowledge graphs, electronic health records, finance— network-structured datasets can be highly sensitive and require controlled computation in a trusted environment, i.e., the traditional cloud computation is not suitable for highly sensitive datasets nor for applications that require low-latency and high-throughput. Further, in regulated environments, auditable computation is imperative. Many modern big-data applications work on top of graph-structured data. For instance, graph databases such as Neo4j [1], JanusGraph [2], TruenoDB [3] and others [4–7] handle large-scale graphs that model biological interactions, social networks, knowledge graphs, electronic health records, finance, etc. These systems manage large graphs and execute traversal algorithms on them. Traversal algorithms exhibit access locality in the graph structure. Although current graph databases such as Neo4j, ArangoDB, and others offer protection for data-in-transit [8, 9]—utilizing SSL/TLS— and for data-at-rest —using Bitlocker or encrypted key-value stores; those do not offer fine-grained authentication and auditability guarantees at the graph structure level (for example, subgraphs, motif).

Similarly, many modern applications utilize a "shared, replicated database" approach to provide accountability and traceability. For instance, many solutions for certifiability and traceability in supply chains using blockchain technologies have risen to prominence [10–15]. Those applications often suffer from significant privacy issues because every node in the network can access a copy of relevant contract code and data

[1]For the rest of the dissertation, the terms graph and network are interchangeable.

to guarantee the integrity of transactions and reach consensus, even in the presence of malicious actors.

Despite the importance of securing not only the content of the nodes but also the *structure* of the graph, state-of-the-art authentication measures for the graph model have been following techniques that are often used in the relational model. Further, modern systems that utilize graph data stores as backend do not provide authentication nor auditability features at the subgraph level and exploiting the access locality of traversal algorithms.

1.1 Motivation

We argue that there is a need for certification frameworks that do not rely uniquely on blockchain technologies as a building block. Instead, we propose allowing a privacy-conscious data owner to run pre-approved and certified code in their own environment on their own private graph-structured data and provide proof of correct code execution and data provenance that can be audited in the future. We examine the risks of using trusted execution environments (TEEs) in the cloud and propose alternatives on how to use TEEs differently.

We investigate the access locality exhibit by traversal algorithms on graph-structured data to present a performant graph processing model while providing authentication and auditability features. We argue that exploiting access locality can help create unique memory layouts that semantically relate individual authenticated sets with the access patterns exhibited by the users in a graph data store. We examine the use of multi-modal knowledge graphs to enhance the authentication engine and expose a richer user experience, automatically caching and pushing relevant content to the users. In particular, we generate blocked individual authenticated sets with high-locality, which serves as a building block for subgraphs authentication. Further, we argue that using a block-based structure can facilitate the encryption of graphs using AES-GCM or AES-CTR. Moreover, we examine the use of a novel mix

of blockchain technologies, trusted execution environments, and graph analytics to provide auditable and authenticated computation on top of the graph model. Finally, we integrate previous work on graph authentication with our block-based graph representation.

We argue that current graph databases offer limited functionality making the management of large networks a challenge, from points of view of analyses and presentation. To tackle this, we propose a scalable and easy to use graph database with an integrated compute engine.

Finally, we investigate stream processing systems at the edge of the network to avoid incurring costly and extensive CDN infrastructure while enabling a personalized graph-based object lookup, low latency, and high-throughput.

1.2 Contributions

1.2.1 AGAPECert

This chapter introduces AGAPECert, an Auditable, Generalized, Automated, Privacy-Enabling, Certification framework capable of performing auditable computation on private data and reporting real-time aggregate certification status without disclosing underlying private data. AGAPECert utilizes a novel mix of trusted execution environments, blockchain technologies, and a real-time graph-based API standard to provide automated, oblivious, and auditable certification. AGAPECert contributes an open-source certification framework that can be adopted in any regulated environment to keep sensitive data private while enabling a trusted automated workflow.

1.2.2 AuditGraph.io

We propose AuditGraph.io, an auditable and privacy-preserving graph processing model that performs auditable computation on graph-structured data on a trusted or

controlled node. AuditGraph.io exploits access locality of traversal algorithms; it utilizes trusted execution environments and blockchain technologies to provide authenticated access to subgraphs and provide auditable proof of correct code execution. To the best of our knowledge, AuditGraph.io is the first solution for auditable, authenticated, and privacy-preserving computation for network-structure data exploiting the access locality exhibited by traversal algorithms.

1.2.3 TruenoDB

We contribute TruenoDB, an easy-to-use scalable graph database and computation engine. TruenoDB is a novel integration of highly optimized algorithms and implementations, with distributed search engines, graph-parallel computations on top of a dataflow framework, and a rich set of drivers. TruenoDB provides a user-friendly Web UI, a simple API for developing plugins, has extensive language support, and commonly used execution engines such as Spark, and includes a library of graph analytics kernels. We validate TruenoDB outstanding usability utilizing a variety of applications ranging from computational systems biology to information retrieval. Finally, we support TruenoDB's practicality through some micro and macro benchmarks that demonstrate excellent performance, scalability, and flexibility.

1.2.4 MioStream

We present the design, implementation, and evaluation of a novel P2P service based on WebRTC (web browsers with Real-Time Communications) called MioStream. MioStream is an open-source alternative for distributed media streaming that runs on the edge of the network without incurring costly and extensive CDN infrastructure. We contribute a unique mix of algorithms using WebRTC data channels. For instance, under network degradation and high-churn environments, MioStream restructures the topology dynamically. MioStream provides authentication, privacy, and integrity of video chunks.

1.3 Impact

All the systems exposed in this dissertation are open-source. Moreover, there is a significant interest in commercializing AGAPECert. Similarly, AuditGraph.io will be used to enhance AGAPECert capabilities. The AuditGraph.io proof of concept can enhance auditability and authentication features of current graph databases. Our systems can be found at:

- AGAPECert: https://github.com/agapecert
- AuditGraph.io: https://github.com/AuditGraphDB
- TruenoDB: https://github.com/TruenoDB
- MioStream: https://github.com/maverick-zhn/miostream

1.4 Dissertation roadmap

The rest of this dissertation is organized as follows. We present an Auditable, Generalized, Automated, and Privacy-Enabling Certification Framework with Oblivious Smart Contracts in Chapter 2. We examine AuditGraph.io that leverages auditability and authentication through the exploration of access locality of the graph model (Chapter 3). Chapter 4 introduces TruenoDB and background knowledge of graph databases. We introduce MioStream in Chapter 5; we provide a background for essential building blocks. Chapter 6 concludes this dissertation.

2. AGAPECERT: AN AUDITABLE, GENERALIZED, AUTOMATED, PRIVACY-ENABLING CERTIFICATION FRAMEWORK WITH OBLIVIOUS SMART CONTRACTS

This paper introduces AGAPECert, an Auditable, Generalized, Automated, Privacy-Enabling, Certification framework capable of performing auditable computation on private data and reporting real-time aggregate certification status without disclosing underlying private data. AGAPECert utilizes a novel mix of trusted execution environments, blockchain technologies, and a real-time graph-based API standard to provide automated, oblivious, and auditable certification. Our technique allows a privacy-conscious data owner to run pre-approved *Oblivious Smart Contract* code in their own environment on their own private data to produce Private Automated Certifications. These certifications are verifiable, purely functional transformations of the available data, enabling a third party to trust that the private data must have the necessary properties to produce the resulting certification.

Recently, a multitude of solutions for certification and traceability in supply chains have been proposed. These often suffer from significant privacy issues because they tend to take a "shared, replicated database" approach: every node in the network has access to a copy of all relevant data and contract code to guarantee the integrity and reach consensus, even in the presence of malicious nodes. In these contexts of certifications that require global coordination, AGAPECert can include a blockchain to guarantee ordering of events, while keeping a core privacy model where private data is not shared outside of the data owner's own platform.

AGAPECert contributes an open-source certification framework that can be adopted in any regulated environment to keep sensitive data private while enabling a trusted automated workflow.

2.1 Introduction

Recently many solutions for certifiability and traceability in supply chains using blockchain technologies have risen to prominence. For example, to provide accountability and visibility in the food supply, blockchain solutions exist such as IBM Food Trust [10], BeefChain [11], Lowry Solutions Sonaria platform [12], ripe.io [13], OriginTrail [14], and SAP Blockchain Service [15], to name but a few.

A component of many blockchain solutions is a smart contract: a piece of code stored in the blockchain itself that is run by all (or most of) the nodes in the chain in response to some *on-chain* event: i.e., some data is added to the blockchain computing nodes which triggers processing. Nodes in the network have copies of the data and the code and achieve data integrity by checking each other's work. This approach suffers from significant privacy challenges because data and smart contract code is replicated on all nodes in the network [16] as a means to achieve varying levels of Byzantine fault tolerance: i.e., malicious actors are unable to affect data integrity.

Thus, some projects have proposed the use of trusted execution environments in the chain to provide some level of privacy-preserving computation [16] (§2.2.2). A trusted execution environment (TEE) is a private, integrity-protected, and secure computation environment built into processor hardware[1] [17, 18]. For example, Intel SGX (Software Guard Extensions) intends to supply confidentiality and integrity guarantees to computation run in environments where the hypervisor (virtualized environments), operating system, or the kernel are probably malicious/adversarial [18]. TEE's were originally intended to provide a way to keep code and data isolated from malicious actors in a shared platform, and they have had questionable (§2.4.1) success toward that goal thus far. However, our trust model simply requires that a TEE can produce cryptographically secure guarantees about which code it ran, not that it isolates data from the rest of its platform.

[1]Throughout the rest of this paper, we use the terms TEE, enclave, and **trusted black box** interchangeably.

Brandenburger et al. (2018) contributed an open-source proof of concept for TEEs within blockchain computation on top of Hyperledger Fabric [16]. Another example using blockchain and Intel SGX is SDTE [19], a data processing model implemented on Ethereum. A more general framework is the Confidential Consortium Blockchain (COCO) that aims to enable scalable and confidential blockchain networks [20]. In all these solutions, the computation is on-chain: i.e., replicated across nodes in the network.

In many of the real use cases requiring the sort of privacy provided by TEEs, participants are reticent to provide even encrypted versions of sensitive data to a public or permissioned blockchain which will enshrine the encrypted data immutably, thus giving attackers an enormous time-based attack surface to find exploits in encryption key management.

AGAPECert leverages privacy-preserving computation via TEEs but allows the TEEs to run in environments controlled by the data owner rather than on-chain. AGAPECert abstains from sending data to a public or even private blockchain network and applies restrictions to the code that runs inside enclaves: the code or algorithm that runs on the private data must be pre-approved by both data owner and regulator.

Before delving further into details of AGAPEcert and its relationship to other technologies, it is important to define a model of the problem that AGAPEcert is designed to solve.

2.1.1 Motivation and Problem Definition

Figure 2.1(a) illustrates a common activity in everyday life that we call a *trust transformation*. A regulator such as the Internal Revenue Service (IRS), an environmental agency, or even just a down-stream purchaser of a product has a form that the individual or company being regulated needs to fill out. This form contains a request for information that is derived from other sources. The source data is gener-

ally considered private by the data owner and therefore is not submitted directly to the regulator. The transformation of the source data into the fields on the regulators form can be considered a *trust transformation* from more private, fine-grained data to less private, coarse-grained data.

When the regulator has cause for increased scrutiny, such as an IRS audit or a surprise inspection, there is typically a need to reproduce the private source data and re-execute the process of the trust transformation under threat of legal action; however, this time under the supervision of the regulator or an independent third party. The source data presented under audit conditions should be verifiable as the same data that produced the responses in the original form. The auditor often has little means of verifying that the data provided by the data owner is correct. Instead, the data owner would typically sign some legal document attesting that the data they have provided is correct to the best of his/her knowledge[2].

The goal of AGAPECert is to enable the trusted service provider or data owner that performs the trust transformation in Figure 2.1(a) to be replaced or augmented with *code* agreed upon by both the regulator and the data owner as shown in Figure 2.1(b). This automates the process of data-centric certification. This process must not infringe on existing models for trust transformation that society already understands and uses, as outlined below.

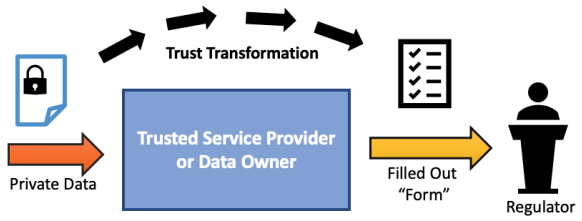
2.1.2 Design Principles

The following features must be supported in order for AGAPEcert to fit most existing certification processes:

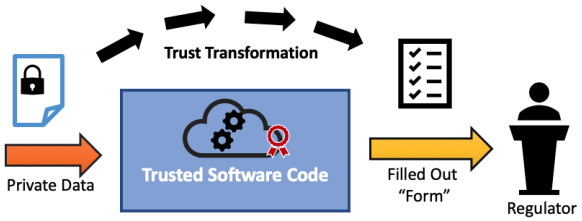
1. The data owner should be confident that private data will not be released to the regulator, even in encrypted (but decryptable) form.

2. The data owner should be able to run the code to fill out the regulators form as often as they like internally without notifying the regulator.

[2]The data owner in question is often the sole source of that information.



(a) Trust transformation today



(b) Trust transformation with AGAPECert.

Fig. 2.1.: (a) Illustration of common model of trust transformation. A person or their trusted agent will fill out a form for a regulator using private, fine-grained source data that they transform into the fields on the regulator's form. (b) AGAPECert replaces or augments the service provider or data owner with a piece of software code agreed upon by both the regulator and the data owner, thus automating the certification process.

3. The regulator must be able to verify that the private data has not changed in the event of a subsequent audit under threat of legal action.

4. The regulator should learn nothing more about the data owners information or business processes other than the exact features of the filled-out form.

5. In some cases, the regulator should be able to confirm that the appropriate code was run without requiring a full audit.

6. The process need not verify the private data beyond a legal assertion by the data owner that the data is correct. However, the process may enable better trust requirements around the integrity of private source data.

This new model has the benefit of repeatable precision enabled by its purely functional nature: when the trusted software code produces the responses in the form such as passed or properly certified, then the data owner knows that they have indeed passed the certification process, and the whims of a later human regulator or auditor cannot change that. The traditional model of data-centric automation has been to ship data to the code that uses it, thus creating privacy concerns. In the AGAPECert model, data-centric automation is achieved by shipping trusted code to the data, eliminating any needless privacy concerns, leaving only those privacy issues required by the contents of the regulator's report itself (§2.6).

2.1.3 Approach

As a key component of the AGAPECert framework, we introduce the concept of *Oblivious Smart Contracts* (OSCs) as a means to achieve *Private Automatable Certifications* (PACs). These concepts achieve the high-level flow:

1. use a piece of standardized, industry-trusted, regulator-approved code that can securely access private data using a standardized graph-based API (§2.2.4),

2. compute an aggregate resulting certification (the PAC) as a purely functional result from the input data,

3. store results back to the platform of choice for the code (i.e., a blockchain, or any standardized graph-based platform §2.2.4),

4. and hash and sign all so that these signatures and hashes can be presented in the event of a legal challenge and used to verify that the code was run faithfully, and the input data has not been changed since code execution.

No information will be leaked from the private data beyond what is produced by the pre-approved code itself. Even the produced PAC does not necessarily have to be shared by the data owner except in the case of a manual audit or legal challenge. It need not even leak the fact that the data owner has run the OSC at all if the OSC itself does not communicate with any outside platform during execution. We can consider the PAC as being produced by a *smart contract* —standard, pre-approved code shared by participants—, and consumers of the resulting PAC as being *oblivious* to all features of the underlying private dataset beyond the aggregate information in the PAC, similar to the widely-accepted use of the term *oblivious computation* [21]. These two features give rise to the name we have coined here, *Oblivious Smart Contract*.

AGAPECert utilizes a graph datastore abstraction, that is, the Trellis framework [22] which specializes the Open Ag Data Alliance (OADA) API framework [23] to provide a standard API for automated data exchange. AGAPECert uses this concept of having a known, standard API for any type of data as a foundational component to build an interoperable codebase capable of interacting with myriad individual platforms (§2.2.4.) Without the concept of a standardizable API layer with the features available in Trellis/OADA, it is not practical to write a piece of code that one would expect to work against many heterogeneous data sources. For those unfamiliar with Trellis and OADA, it is important to note that the API framework works for any kind of data or use case, not simply those within the agriculture industry or

the food supply chain. Therefore, AGAPEcert's use of Trellis and OADA enables it to be fully generalizable.

As an example of a PAC protocol utilizing blockchain as a byzantine fault tolerant[3] data storage layer, AGAPECert can leverage auditable computation through a Blockchain-Gateway that allows pluggable shared ledgers (§2.5.1) to store anonymous hashes computed during the PAC generation process.

AGAPECert integrates two crucial concepts derived from Intel's documentation [24]; *REPORT* and *QUOTE* (Table 2.1). The Intel SGX documentation defines a *REPORT* as a unique signed structure that binds a key to the enclave hardware, the signer of the codebase, the code itself, and any user-defined data. In the remote attestation process (§2.2.2), the *Quoting Enclave* verifies the *REPORT* and creates and signs the *QUOTE* with a key that is only known to trusted Intel SGX hardware. The *QUOTE* is utilized by the Intel's Remote Attestation Services to verify the identity of particular code running inside an enclave. AGAPECert can store the $Quote\_Hash$ (§2.2.1) in the PAC or a shared ledger, serving as BFT proof of the computation result timing.

AGAPECert diverges considerably from current edge computing (i.e., processing outside the cloud) literature. The AGAPECert architecture includes a graph data store node, a compute engine node, a broker, and a validator (§2.3.1). The approach is highly flexible to a wide array of situations: OSC code can interact with or be initiated by regular on-chain smart contract code as appropriate, various data components can be chosen by participants as either on-chain or off-chain to support the use case, and results and hashes can be reported directly to certifying bodies, pushed to a blockchain, or held only by the data owners. PACs can also be composed: one "meta"-PAC can be created by an OSC which simply verifies the validity of many other PACs, avoiding the need to even disclose the underlying PACs themselves.

[3]A Byzantine Fault Tolerant (BFT) network can continue operating even if some of the nodes fail to communicate or act maliciously.

Table 2.1.: Summary of Notations [18].

Notation	Explanation
OSC	Oblivious Smart Contract
PAC	Private Automated Certification
REPORT	A unique signed structure that binds a key to an enclave
QUOTE	Created by the Quoting Enclave (Signed REPORT)

2.1.4 Trust Levels

Not all use cases have the same trust requirements. AGAPECert proposes classifying OSC structures that solve various use cases into a hierarchy of three trust levels with increasing trust guarantees at the expense of increasing complexity. (Table 2.2):

- **Trust Level 1**, Owner Attested (OA): Regulator or consumer of PAC trusts the data owner to faithfully execute the OSC, and therefore does not require proof of correct execution provided by a TEE. Computation is still auditable under legal challenge. Example: one company prepares a report that utilizes data from a supplier, and they would like to automate report preparation without requiring the transfer of private source data from the supplier that the report preparation process would naturally aggregate anyway.

- **Trust Level 2**, Enclave Attested (EA): Regulator or consumer of PAC requires attestation that OSC code was executed faithfully. Computation is auditable under legal challenge, and correct code execution can be attested without access to private data. Example: government regulator would like to automate checks against data owner's digital data, so the resulting PAC contains TEE attestation.

- **Trust Level 3**, Blockchain Attested (BA): Regulator or consumer of PAC requires proof of correct execution as well as proof of event ordering for issues like double spending prevention. Contains same components as Level 2, with the

addition of a Byzantine fault tolerant shared data storage layer like blockchain. Example: Buyer of a product wants to know that it is from a certified set (i.e., purchase does not exceed available certified balance), but seller does not wish to leak information about timing and quantity of other sales.

2.1.5 Contributions

In summary, our contributions are:

- Utilizing a novel mix of blockchain technologies, trusted execution environments, and graph-based APIs, AGAPECert provides a unique, auditable, generalized, automated, and privacy-enabling certification framework for any industry.
- AGAPECert can be used as a novel privacy-preserving food-safety framework in particular.
- AGAPECert contributes auditable computation for use cases that require preserving data ownership and privacy.
- AGAPECert contributes a handshake protocol utilizing trusted execution environments and the OAuth2.0 (§2.3.3).
- AGAPECert introduces, for the first time, the concepts of *Oblivious Smart Contracts* and *Private Automated Certifications* as building blocks to automate and protect data ownership in real use cases, i.e., certification frameworks, in the supply chain.
- This work provides an opensource design, implementation, and evaluation of our solution (§2.5, §2.7).

2.2 Background

AGAPECert computes on encrypted or access-controlled data (Confidentiality), preserves the privacy and state of the original private data (Integrity), and, for some use cases, provides the latest record of the certification (sequence.)

2.2.1 Cryptographic Hash Functions and Data Integrity

AGAPECert provides integrity protection of private data and code through well-known cryptographic hash functions. Formally, hash functions map an arbitrary length input message m to a fixed-length output $h(m)$ referred to as a hash [25]. The hash has the property that it is computationally infeasible to create an input string which produces a pre-defined hash value, it is infeasible to invert (i.e., determine the original input from the hash alone), and it is deterministic (the same input string always produces the same hash).

AGAPECert uses the SHA256 hashing function to create five crucial hashes (Table 2.3) to uniquely characterize the private data, the REPORT (local attestation), QUOTE (remote attestation), PAC, and OSC.

2.2.2 Trusted Execution Environments

Trusted Execution Environments (TEEs) are an industry innovation to enhance the privacy of data and computation [16]. Through specialized and isolated execution environments (enclaves) TEEs shield applications against any malicious operating system, hypervisor, firmware, or drivers [18]. TEEs include functionality to encrypt sensitive communications, seal (encrypt) data, and verify the integrity of code and data. TEEs implementation includes specialized hardware instructions embedded in a machine's processor.

Intel SGX

Intel SGX (Software Guard Extensions) aims to supply integrity and confidentiality guarantees through a TEE [18]. Intel SGX creates a private and trusted execution region in the computer's processor called an *enclave*: a secure "virtual container" or black box that contains code and secret data [18]. The intended code and data are injected from an *untrusted* region into the enclave. Then, built-in software attestation and sealing mechanisms can provide proof that an application is interacting with the exact/correct software in the enclave and not an attacker's injected malicious code or simulator.

AGAPECert Trust Level 2 and above require an enclave to exist in the compute engine (§2.3.1). The data owner trusts the environment where they run the code on top of their private data, and the code they choose to run there has been pre-approved by them or their trusted service provider in advance. In addition, the regulator has also pre-approved the code and knows the appropriate *REPORT* parameters that the code will produce when executed in a TEE. Hence, the code injected in the enclave has been approved by both the *regulator* and the *data owner* — a *code trust relationship*.

Local and remote attestation

To prove that specific software code is running in trusted hardware, Intel SGX relies on local and remote attestation [18, 26]. The attestation mechanism provides *proofs*, which comprise a cryptographic signature of the enclave's content (code, data, and parameters) using the platform's secret attestation key known only to the processor. In *local* attestation, the cryptographic proof is verifiable locally by another enclave running in the same processor; this allows secure collaboration to reach a result.

In remote attestation, the cryptographic signature on the proof can be verified by a third party as having originated from a particular piece of trusted hardware using the assumption that the secret key within the processor hardware is unknown outside

of the hardware itself and the hardware never reports that key to running software. In other words, the only entity that could have produced the signature is a trusted processor because it is the only entity that knows its signing key.

AGAPECert utilizes a remote attestation mechanism as the means by which the data owner can prove to the regulator that they have faithfully executed the pre-approved code. OSC code reads private data and produces purely functional outputs from that data, including additional hashes (§2.2.1). This makes code execution both reproducible and verifiable given the same input data.

Intel Enhanced Privacy ID [27]

A critical aspect of privacy-preserving computation is attesting that compute devices have not been tampered with and are authentic. Intel's *Enhanced Privacy ID (EPID)* is an implementation of ISO/IEM 2008 that handles membership revocation and anonymity. Membership revocation exposes methods to invalidate compromised secret keys. Anonymity means that EPID will attest to the authenticity of devices without identifying the particular device, i.e., the signature was created by a key from amid a trusted group of secret keys. However, EPID cannot distinguish which particular key in the group created a given signature. AGAPECert utilizes these existing remote attestation signature schemes.

Intel SGX DCAP [28]

EPID has some limitations:

- Participants are reticent to outsource trust decisions.
- Some highly-distributed use cases require scalable verification points and need to avoid a single point of failure.
- AGAPECert can run computation in controlled environments restricting Internet access at runtime.

To tackle these issues, Intel allows the use of Data Center Attestation Primitives—Intel SGX DCAP—to build customized third party remote attestation. At this point, only servers with Flexible Launch Control (FLC) enabled Intel Xeon E Processors are supported [28].

2.2.3 Blockchain Technologies

A blockchain is an immutable, decentralized digital ledger [16, 29]. Multiple computers store ordered transactions, linked together through a series of hashes that represent all the data in the ledger up to a given block. Immutability implies that a record in a set "chained" together by hashes cannot be changed without affecting all the subsequent block hashes. A blockchain provides inherently byzantine fault tolerant [30] independent auditability capabilities typically by placing computational constraints on block content that require brute-force guessing to solve. Those constraints make it too difficult for a malicious attacker to game the system since they cannot brute-force guess solutions any faster than non-malicious participants. Adding to the concept of the immutable shared ledger is the concept of a *smart contract*. A smart contract is defined as code that resides in the blockchain itself. An event can trigger some or all nodes to execute that code. The input and output data for each run of the contract code is also typically stored in the blockchain to make code execution directly verifiable: every node uses the same inputs, runs the same code, and verifies that they produce the same output.

The simplest forms of OSC do not require a blockchain. However, some use cases require provable concepts of time or event ordering. In such cases, including a blockchain building block can be key to giving an OSC the capability to provide collaborative interaction that respects ordering of events. In such cases, AGAPECert can utilize a blockchain to store hashes when building a PAC. AGAPECert currently implements a Generalizable Blockchain-Gateway with IBM Hyperledger Fab-

ric as a building block, but can be extended to other blockchain frameworks such as Ethereum [31].

2.2.4 Real-time Graph-based API

We utilize the Trellis Framework which exposes standardized REST API semantics to interact with a user's private data store. The purpose of Trellis is to enable standardized, automated, permissioned, ad-hoc, point-to-point data connections through the use of a common REST API. It is beyond the scope of this paper to fully recount the details of Trellis[4]. However, some critical features of Trellis are important to understand AGAPECert:

- *Resource Discovery*: Filesystem-like graph schemas to define where data can be discovered. For example, catch locations for a fishing vessel for May 1, 2020 could be defined as discoverable at graph path /bookmarks/trellis/fishing/catch-locations/day-index/2020-05-01.

- *Write Semantics*: Trellis standardizes how data within a graph is written: all data changes are reduced to an ordered stream of idempotent merge operations[5]. Operation ordering is only guaranteed per-resource, not globally.

- *Change Feeds*: Clients can register for real-time change feeds for any arbitrary subgraph of data. This provides both a real-time communication channel as well as a means of concurrency-safe 2-way data synchronization. The change feed is comprised of the ordered stream of idempotent merge operations.

- *Authorization*: Trellis standardizes how any client registers and obtains authorization tokens at any Trellis platform.

[4]Refer to https://github.com/trellisfw for more information.
[5]An idempotent merge operation means that a given JSON document is produced that only affects matching keys. Keys that do not exist are created, existing ones are deep-replaced at overlapping key paths, similar to a common upsert. Applying the same merge repeatedly results in the same resource state at the mentioned key paths, hence why it is idempotent.

- *Permissions*: Trellis standardizes how data can be locally shared within a platform.

2.3 Method: AGAPECert System Model

2.3.1 AGAPECert Architecture

AGAPECert considers two primary actors: *data owners* and *regulators*. Data owners are clients that own private and sensitive data. *Regulators* are actors that would like to know some derivative of the clients private, sensitive data without requiring the disclosure of that data itself. Note that the regulator may not be only what is traditionally considered a regulator, e.g., from a government agency, but rather is used in a broader sense here as any entity looking for information that may be derived from a clients private data. By this definition, a regulator could be a direct customer of the data owner, a down-stream buyer in a supply chain, or a business partner.

We define two critical components of the certification process:

- *Private Automated Certification (PAC)*: the derivative output of the clients data (i.e., the contents of the "form" that the regulator requires the data owner to fill out).

- *Oblivious Smart Contract (OSC)*: regulator-approved code which, given access to private input data, produces the desired PAC (i.e., the "questions" on the regulator's "form").

The client (or their trusted service provider or industry consortium) ensures that the OSC obtains only the necessary data to produce a PAC, and that the PAC will not leak any unauthorized information (such as copies of the private data, or knowledge of when the OSC code was run). The client runs the approved code and provides it access to their private data to it to obtain a signed PAC. This PAC should contain, at minimum, a cryptographic hash representing the input data used in its

computation. For Trust Levels 2 and above, it must also include the cryptographic hashes of REPORT ($Report\_Hash$) and QUOTE ($Quote\_Hash$) from the Intel SGX enclave. The client then provides their PAC to the regulator upon request, at which time the regulator may validate the PAC according to the trust level for that use case. Should a subsequent legal challenge be necessary, the client can produce the private data to a legal authority, which can verify that the cryptographic hash for that data ($Data\_Hash$) matches that from the PAC, and can also re-run the OSC code to produce an equivalent PAC for comparison.

AGAPECert architecture comprises six main components, four required and two optional based on the level of trust required (Figure 2.2):

- **Compute Engine**: a compute node controlled by or trusted by the data owner that is capable of running the OSC code. A compute engine is required for all trust levels. For Trust Levels 2 and above, the compute engine must be Intel SGX-enabled.

- **Data Store**: a Trellis-conformant data storage platform that holds the private data owned by the client. This serves as the source of the data for the OSC, as the real-time communication channel for the broker and the OSC, and as the destination for the PAC produced by the OSC.

- **Broker**: a web application that initiates, authorizes, provisions, moderates, monitors, and validates OSC execution, communicating with the OSC through the secure shared Trellis connection. This serves as the bridge between the data owner and the OSC, acting as a service manager; all OSC services can be monitored through this web-app.

- **Validator**: a web application that can validate a PAC, including remote attestation for Trust Level 2 and 3, and checking a blockchain for Trust Level 3.

- **Attestation Service**: The Intel SGX attestation service or DCAP (§2.2.2). Given a particular QUOTE, this service can attest whether the QUOTE was produced by a legitimate Intel SGX enclave or not, thus attesting proper code execution. Required for Trust Levels 2 and above.
- **Blockchain Gateway**: an interface to a blockchain that is trusted by the regulator and the data owner.

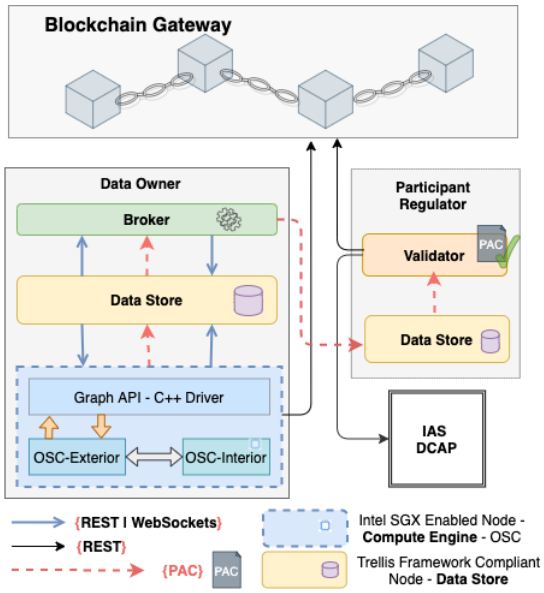


Fig. 2.2.: AGAPECert architecture for Trust Level 3. The data owner main components include a compute engine that runs the OSC (exterior and interior), a data store, a service manager for the OSCs (broker), and a blockchain-gateway to store the *Quote_Hash* and UUID (PAC.id). The regulator includes the validator and its data store. The validator queries the ledger to verify a particular PAC. Also, the validator attests correct code execution connecting to remote attestation services or DCAP.

2.3.2 Oblivious Smart Contracts

An Oblivious Smart Contract (OSC) is software code that reads private data to compute a result such as pass/fail and generate a PAC (§2.3.3). OSCs run in the compute engine node. For Trust Levels 2 and 3, this computation happens inside a TEE on the compute engine.

As explained by Intel's documentation [24] Intel SGX applications (such as Trust Level 2 and above OSC's) comprise two parts, the *untrusted part* of the application which communicates with the enclave and the *trusted part* that includes the computation inside enclaves. Note that while these terms make sense in the traditional environments where Intel SGX is intended to run, they are more confusing than helpful in our context where the environment running the OSC is assumed to be trusted by the data owner already. We will instead use the term *enclave exterior* to describe what Intel SGX terms the untrusted part, and *enclave interior* to describe the trusted part.

Trust Level 1 does not require a TEE; this simple form of OSC is simply any code capable of interacting with a Trellis platform to read data and save a resulting PAC.

For the Trust Levels 2 and 3, the OSCs include native C Bridge functions that communicate with the enclaves. The enclave exterior of the OSC connects and retrieves the private data from the Trellis data store and injects a buffer into the enclave interior of the OSC. AGAPECert includes C++ classes that implement the Trellis REST and WebSockets APIs to communicate with the Trellis data store through a shared resource located in the Trellis graph at $/bookmarks/OSC/H_k$, where H_k can be a random string generated at runtime by the OSC or a static feature of the OSC and is discovered by the Broker when initiating the connection.

The enclave interior of the OSC computes the cryptographic hashes necessary to audit the computation in the future and passes them through the enclave exterior to be stored in the PAC, which is stored back to the data owner's Trellis data store. In Trust Level 3, AGAPECert also stores the cryptographic hashes in the blockchain.

It is important to note that since AGAPECert stores only the cryptographic hashes in the blockchain, and these hashes cannot be linked to the source data using only the hash, this protects the privacy of the data owner; i.e., there is no information leakage from this process such as third party knowledge of how many times the data owner has run the OSC. However, storage of the hash in the blockchain can leak the time

when a given PAC was generated since a regulator receiving the PAC in the future can check where in the blockchain the hash was saved.

The typical OSC runs continuously, awaiting notification from the broker through the Trellis data store at shared storage location */bookmarks/OSC/*H_k to start a new PAC generation process on a new subset of data, or until a restart is submitted from the broker. Each run of the OSC produces one or more PACs deterministically, after which point the OSC returns to a listening state awaiting further signals from the broker.

2.3.3 Private Automated Certifications (PAC) Workflow

To generate a certification under Trust Level 3, AGAPECert uses the following workflow (Figure 2.3.) For other trust levels, the respective components not used by those levels are simply left out. Note that AGAPECert utilizes a set of RFCs in this process (RFC7591, RFC7517, and RFC7519) as prescribed in the Trellis authorization protocol [32].

1. *Install OSC*: The data owner installs the OSC on their compute engine. This produces a random public/private asymmetric key pair, with the public key saved as a JSON Web Key (jwk from RFC7517) in a newly generated Trellis Client Certificate (C_{osc}).

2. *Authorize Broker*: The data owner logs into their Trellis compliant node via OAuth2 to authorize a token for the Broker.

3. *Watch for OSCs*: The broker opens and maintains an active websocket connection to the Trellis data store that watches the top-level */bookmarks/OSC* document for any connected OSCs.

4. *Authorize OSC*: The data owner uses the Broker to pre-register the OSC's Trellis Client Certificate C_{osc} at their Trellis data store as an authorized OSC.

5. *Start the OSC*: The data owner starts the OSC. They can also verify that the hash of the OSC code OSC_Hash matches the one available in a private certified code repository.

   (a) The OSC Exterior performs OAuth2 dynamic client registration by exchanging its Trellis Client Certificate C_{osc} with Trellis for a Client ID. It then performs OAuth2 Client Grant flow during which it proves that it has the private key for the pre-registered certificate by creating a signed jwt_bearer token (RFC7523). This process results in a properly scoped launch token (T_l) to access the user's Trellis data store at $/bookmarks/OSC$.

   (b) The OSC Exterior generates a random string H_k that uniquely identifies this instance of OSC. The OSC uses the token received from previous step to create a resource ($/bookmarks/OSC/H_k$) in the data store. It also puts information about itself into that document.

   (c) The OSC exterior opens and maintains an active Trellis websocket connection to the Trellis data store watching for changes to the new $/bookmarks/OSC/H_k$ as the main communication channel between the broker and the OSC.

6. *Communication Channel Opens*: The brokers active Trellis websocket connection notifies it that a new OSC resource exists at $/bookmarks/OSC/H_k$.

7. *Optional: Validate OSC Quote*: In the event that the data owner requires confirmation that their platform has loaded the correct OSC code, they can load their own credentials for the Attestation Service §2.2.2 (IAS or DCAP) into the broker and it will initiate remote attestation to verify that the enclave is legitimate. In most cases the user already trusts their own platform and would not require signing up with IAS. This remote attestation workflow produces a QUOTE (§2.2.2).

8. *Provision Data to OSC*: The broker then provisions a properly-scoped data access token T_d for the OSC to use in creating its PAC. The broker writes this

token to the shared Trellis communication channel at $/bookmarks/OSC/H_k$ along with any data filtering instructions (such as restricting the PAC to only consider a particular days dataset). The active websocket connection held by the OSC exterior notifies it of the newly provisioned token and filter, triggering the OSC to begin the core PAC generation.

9. *OSC Interior Requests Data*: The OSC exterior receives the data access token and filtering instructions and notifies the OSC interior to begin PAC generation. The OSC interior uses its knowledge of the known, published Trellis semantic data structures to begin requesting data it needs. Requests for data initiated by the OSC interior are forwarded to the OSC exterior to make the actual requests over the active Trellis websocket connection.

10. *OSC Exterior Injects Data*: The OSC exterior serializes and injects the received data into the OSC Interior as it comes back from Trellis.

11. *OSC Interior Computes PAC*: The OSC interior computes its core certification result (i.e., pass/fail) from the received input data, as well as a cryptographic hash ($Data\_Hash$) of all serialized data received during the generation of one PAC. Upon completion of all received data, the OSC interior saves this hash of all input data to the PAC in the data store.

12. *OSC Interior Hashes PAC*: The OSC interior generates a hash of the entire PAC (including $Data\_Hash$ and a random universally unique identifier for the blockchain transaction) and saves this back to the PAC itself in the Trellis data store through the OSC exterior.

13. *Exterior Obtains TEE Quote*: The OSC exterior sees the hash of the overall PAC and concludes that the OSC interior has completed its work. The OSC exterior then instructs the OSC interior to obtain a QUOTE from a local quoting enclave, including the hash of the PAC as the user data for a new REPORT

from the OSC interior. The OSC exterior gathers the final QUOTE from this process and writes it to the PAC in the Trellis data store.

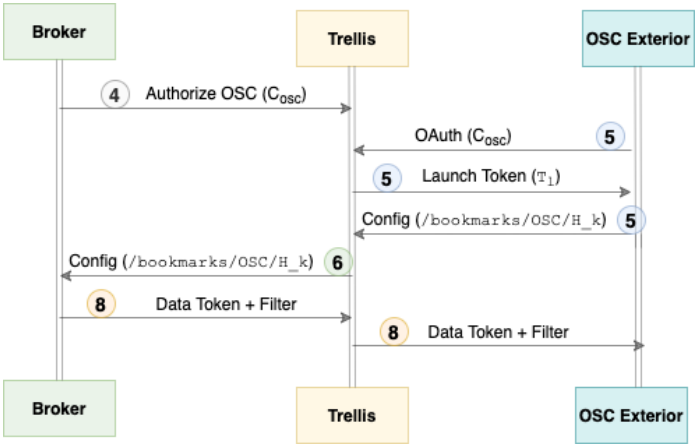
14. *OSC Interior Sends Quote_Hash to Blockchain*: For Trust Level 3, the OSC exterior will then communicate with the Blockchain Gateway to record the unique identifier and $Quote\_Hash$ in the blockchain, along with any further case-specific requirements.

15. *Data Owner Sends PAC to Regulator*: Finally, at a later time, the regulator receives the generated PAC from the data owner and uses the Validator to check it. The Validator sends the QUOTE from the PAC to an attestation service to verify that the QUOTE was indeed generated with an Intel EPID key that has not been revoked. Note that if a processors key is revoked, this either invalidates all prior PACs generated by that processor, or some outside means of providing a trusted timestamp of QUOTE generation (such as that provided by Trust Level 3 through a blockchain) must be included in this flow to maintain validity of PAC's generated prior to some known revocation time. The Validator also queries the blockchain ledger to validate the $Quote\_Hash$ and case-specific data.

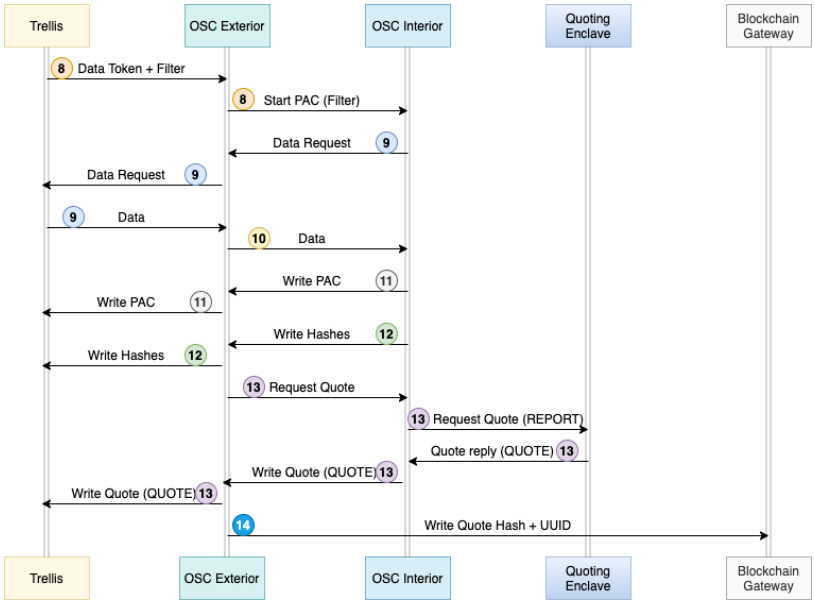
2.3.4 Blockchain-Gateway Schema

AGAPECert stores a minimal set of cryptographic hashes (§2.2.1) in the shared ledger. These hashes do not convey any identifiable private information to an eavesdropper of the transactions in the blockchain nodes. We define the methods of the smart-contract necessary to store hashes from the OSC.

The AGAPECert proof-of-concept models the business network utilizing Hyperledger Fabric (§2.5.1.) Hyperledger Fabric requires the definition of assets, participants, and transactions. AGAPECert utilizes a PAC as an asset in the blockchain that we define as a "Fabric PAC" $fabPAC$. We define two participants: an anony-



(a) AGAPECert Workflow (Before PAC Generation.)



(b) AGAPECert workflow - PAC Generation Overview.)

Fig. 2.3.: (a) Illustrates the OSC's Authorization to the provisioning of data (steps 4-8) (b) AGAPECert generates a PAC (steps 8-14).

mous participant that stores PACs in the ledger and a regulator who queries the transactions and assets in the distributed ledger.

AGAPECert defines a simple schema of at least two strings for a $fabPAC$: (unique identifier, $Quote\_Hash$) and for some use cases a One-Time-Use-Key (OTK). AGAPECert's blockchain schema is shown in Table 2.5. This forms the basis for use cases in Trust Level 3.

2.4 Security Analysis

AGAPECert does not alter the existing real-world model of requiring the regulator to trust the data provided to it by the client under threat of legal recourse, as discussed in 2.1.2. Our security analysis therefore only focuses on guarantees made about computation on the private data (which is assumed correct until audited), rather than about the private data itself.

2.4.1 Side channel attacks to the Compute Engine

TEE technology such as Intel SGX can be vulnerable to "side channel attacks:" [33–38] malicious code running on the same processor can learn about enclave computation and data via round-about methods such as tracking cache timings after an enclave is switched out of execution. AGAPECert assumes the data owner is running the OSC in an environment they already trust: the threat of a malicious entity on the same processor does not apply, hence AGAPECert is immune to traditional enclave side-channel attacks.

However, a data owner using past side-channel attacks against their own trusted enclaves (such as the enclave that provides quotes) could learn the remote attestation keys [33, 35] for their own system and use that to forge fake QUOTE's. Since such attacks have been discovered, researchers have also contributed mitigation techniques to patch those security vulnerabilities [33,35]. Additionally, Intel's security advisories

provide critical mitigation techniques [39]. For instance, some vulnerabilities require microcode level and software mitigations.

AGAPECert runs in a controlled environment in which standards and best practices are enforced; therefore, data owners and participants in the protocol must have the latest Intel SGX Software installed, the newest microcode updates through BIOS updates, and any other recommended measures such as updated operating systems and virtual machines. Microcode updates upgrade the Security Version Number (SVN) utilized in the implementation of Intel SGX [39]. Microcode updates provide new sealing and attestation keys to the enclaves on the platform [39]. Hence, we can track the SVN embedded in the PAC the regulator will assess the validity of a specific PAC via a customized attestation process using current vulnerabilities.

This means that at any given time, it is not known to be possible to forge QUOTEs that contain the latest SVN until a future vulnerability is released. In the event of catastrophic security failure of Intel SGX's architecture, data owners may need to refresh relevant PACs after updating their processor microcode to produce updated SVNs. The purely functional and auditable nature of AGAPECert's OSCs fit this model well. In addition, if the QUOTE hash was published in a blockchain prior to vulnerability discovery, regulators may consider a "likely validity" date since the blockchain can attest *when* the original QUOTE was signed.

Recall as well that the computation is auditable at any time: should the regulator question a result, they can simply trigger an audit of the private data, which could be as simple as running the OSC again with the auditor's oversight. Consider as well that it is often vastly easier for a malicious data owner to forge their private data (which they can do today without AGAPECert) than it is to attempt cracking open a CPU in an attempt to probe for highly guarded embedded attestation keys. Once such foul play is discovered during any audit, Intel EPID can simply revoke the key that the malicious data owner spent so much effort attempting to learn, providing severely diminishing returns to any such attacker.

Therefore the traditional side-channel security vulnerabilities with TEE computing have little ability to impact the security of AGAPECert in general.

2.4.2 Analyzing AGAPECert's Trust Levels

Owner Attested

The regulator trusts the data owner to correctly execute the OSC. Therefore, the Compute Engine, Broker, Data Store are assumed to be trusted. In the case of an adversarial data owner, the regulator can still validate the private data and code execution via audits.

Enclave Attested

Trust Level 2 requires the correct execution of the OSC/Algorithm. Therefore, Trust Level 2 relies on the attestation capabilities of the Intel SGX Architecture. Following our previous discussion on side-channel attacks, an adversarial data owner can steal secrets from an outdated Intel SGX-enabled node signing code and data as genuine compromising ultimately Trust Level 2. AGAPECert implementations must require up-to-date remote attestation schemes, including the latest SVN known to have reasonably uncompromisable attestation keys.

Blockchain Attested

For Trust Level 3, the discussion about faithful code execution is analogous to Trust Level 2. Ordering of events provided by a shared ledger can expose the execution timestamp of an OSC and creation of a PAC breaking our premise of obliviousness and revealing useful information for an interested party. AGAPECert utilizes a Blockchain Gateway to submit *anonymous* and *asynchronous* transactions to a shared ledger or a mix of ledgers hiding the identity of the participants. In the case of a compromised

shared ledger, no useful information is derived by an attacker solely from the global state.

2.4.3 Discussion

There exist extremely sensitive datasets and data sources—electronic health records, stock market, finance, etc. —in which even the leakage of a reduced set of bits can be catastrophic. Under adversarial environments, these use cases require stronger cryptosystems that offer semantic security [40,41] (homomorphic encryption.) AGAPECert use cases and trust model limit the capabilities of an adversary. For instance, a participant is limited by standards, regulations, and audits. Moreover, the possibility of legal action—when a deviation from the protocol is suspected—forces the participant to maintain a good reputation. Powerful adversaries (stronger than HBC[6]) that can get access to the blockchain cannot derive any useful information from the stored cryptographic hashes and random universal unique identifiers. AGAPECert does not rely solely on Intel SGX attestation and sealing primitives; instead, AGAPECert contributes a set of trust levels providing adaptability features according to particular use cases. AGAPECert's future releases can allow homomorphic encryption exposing Homomorphic and Oblivious Smart Contracts (HOSC.) Integrating SEAL [43] in AGAPECert allows a richer set of devices as compute engines.

2.5 Prototype Implementation

The AGAPECert prototype implementation components are available as open source as shown in Table 2.6, and documentation for how to install and run the entire flow can be found at https://github.com/agapecert/agapecert.

[6]"The honest-but-curious (HBC) adversary is a legitimate participant in a communication protocol who will not deviate from the defined protocol but will attempt to learn all possible information from legitimately received messages" [42].

2.5.1 Blockchain Gateway

For Trust Level 3, AGAPECert's prototype implementation interacts with a *pacContract* deployed in a blockchain network through a custom Javascript-based Blockchain Gateway[7]. The primary purpose of the Blockchain Gateway is to submit anonymous transactions to the ledger. The Blockchain Gateway is generalizable and can accommodate pluggable shared ledgers. This gateway allows the Broker, Validator, and OSC to interact with a shared ledger (or a mix of ledgers).

2.5.2 Trusted Compute Engine

We implemented the OSCs utilizing C++ for the OSC exterior. We utilized OpenEnclave [44] to implement C bridge functions for the OSC interior. The compute engine exposes a rich API (C++ driver Listing 2.1) to allow secure communication between the OSC, the graph-based API, and the Broker. The C++ driver implements secure WebSockets.

```
/// Trellis C++ Driver - Example
Trellis * objTrellis = new Trellis();
objTrellis->getPrivateData().wait();
m_private_records = objTrellis->m_private_records;
// AGAPECert C++ Driver - Compute Engine Methods
objTrellis->getUUID();
objTrellis->getToken();
objTrellis->getAuthorization();
objTrellis->getPrivateDataPath();
objTrellis->getPrivateData();
objTrellis->putPAC();
```

Listing 2.1: Example OSC Exterior usage of Compute Engine C++ Trellis driver (https://github.com/agapecert/compute-engine)

[7]https://github.com/agapecert/blockchain-gateway

2.6 Example Applications

There are an enormous number of potential applications for AGAPECert with its OSC+PAC model. This section shows a few non-trivial illustrative examples[8].

2.6.1 Trust Level 2: Certified Fishing Catch Area

The global fishing industry would like to eliminate over-fishing by requiring fishing vessels to catch fish only in approved areas. Fishermen consider their active catch areas to be proprietary to their business. The industry needs a practical zero-knowledge proof that can certify a particular fish was caught within legal boundaries without disclosing the actual catch locations. A sustainable fishing industry consortium creates an OSC which checks a given Trellis data store for a list of catch locations within a certain time period or within a certain group identifier like a lot number. The OSC pulls a set of approved geospatial catch boundaries from an industry list, intersects the catch locations with the boundaries, and produces a PAC with the time period or lot number, a true/false answer about whether all the catch locations fit within the boundaries, an identifier for the set of boundaries used, and a QUOTE from the trusted black box attesting that the OSC code was executed faithfully.

The industry would like to minimize cheating, and authorizes a set of trusted catch location recording devices that are maintained and periodically tested by approved auditors. The OSC can be augmented to verify that each catch location contains a signature from a trusted recording device manufacturer, and that the device is present in a recent active audit that is digitally signed by a trusted third-party auditor, all using the Trellis standard document integrity signature process. This additional true/false answer is added to the PAC, verifying that the catch locations themselves were attested by trusted parties other than just the fisherman. The fisherman uses the AGAPECert broker and their OSC-enabled Trellis platform to authorize and

[8] A complete description of example applications can be found at https://github.com/agapecert/osc-definitions/wiki

run the OSC, producing the certification back to their Trellis platform. Before each batch of fish can be sold at the docks, the buyer must receive the PAC for that days catch. This represents a Trust Level 2 (Enclave Attested) computation where the global fishing industry would like to know that the code was faithfully executed by the fisherman.

2.6.2 Trust Level 3: Organic Mass Balance

One of the more difficult certification problems is characterized by a mass balance. In its simplest form, there is some mass of product that has been certified to be produced (either based upon the total inputs to the process, or based upon a human auditors assessment), and the industry would like to know that the seller of a product indeed can certifiably produce that amount of that product. If an organic farmer could receive a certification that they can or did produce 10 tons of organic apples, downstream buyers of those apples would like to know that the farmer has not re-used that 10-ton organic certification multiple times with multiple buyers, thereby selling potentially non-organic apples under an organic certificate. The farmer, on the other hand, does not want to put a list of their transactions into some shared database for buyers to check for validity since this could tip off buyers about how much inventory he has or how many sales he has made recently and to whom. The buyer needs a zero-knowledge-style proof that the certified product they are buying has not been sold under this certification to someone else

An industry consortium agrees on one or more blockchain platform(s) to act as a byzantine fault tolerant shared datastore. The consortium produce an OSC which can look at a buyer's Trellis platform for a private ledger of sales. This ledger should be initiated with an additive transaction that is digitally signed by a trusted auditor (i.e., the auditor attests that the farmer has 10 tons of organic apples). The signature is verified by the OSC, and the balance is computed by subtracting any subsequent verifiable sales. The OSC finds a proposed new sales transaction in the Trellis data

store, digitally signed by the buyer and the seller. The OSC verifies that the amount of the sale does not exceed the available balance, verifies the signatures, and then saves the transaction to the end of the private ledger. The OSC produces a PAC indicating success/failure for the transaction, which can be automatically saved back to either the buyers Trellis data store, the sellers, or both. If the buyer has their own private ledger, this PAC from the seller can serve as auditable, traceable proof to include in the buyers private ledger and add to their available inventory, all without disclosing any of the buyer's sales to any outside party.

As specified to this point, the protocol suffers from a double-spending attack where the seller simply maintains multiple private ledgers, providing different ones to the OSC depending on which customer they are selling to, thus enabling them to spend the same certified product more than once. To alleviate this problem, we introduce the concept of a one-time-use asymmetric key pair generated by the OSC and verified by a smart contract on the OSCs chosen blockchain. When the OSC is evaluating a proposed transaction, it accesses the sellers Trellis datastore to retrieve the one-time-use private key from the previous transaction and an indicator of where to find the corresponding public key in the blockchain storage layer. The OSC checks the blockchains record for that key to see if has been marked as used yet or not. If it has not been used, then the OSC initiates a smart contract at the blockchain platform to mark it as used, which is only allowed by the smart contract when it verifies that the OSC can produce a signature with the private key corresponding to the public key in the chain. The OSC sees this successfully finish and then asks a different smart contract on the blockchain platform to store a new one-time-use public key and registers it in the blockchain along with a hash of the new ledger. The OSC will fail the transaction if the ledger hash for the key it used does not match the hash of the current ledger it is updating. The OSC stores the private key for this new one-time-use key pair for the next transaction. The OSC also checks that each successful transaction in the private ledger has a corresponding used key recorded in the blockchain with matching ledger hashes. Note the importance of including

the hash of the ledger in the chain with the public key. We do not want to link transactions in the chain by allowing one key to point to the next key in the chain. However, if the two are truly unlinkable, then it is possible for a malicious seller to still double-spend by simply maintaining multiple ledgers with multiple one-time-use keys. Therefore, the OSC and smart contract must allow the link to be maintained in the private ledger, both refusing to create and store a new one-time-use keypair whose full ledger hash is already present with another one-time-use key in the chain. For maximal trust, the blockchain nodes themselves should also be capable of performing remote attestation to validate a QUOTE from the OSC interior. This is an example of a Trust Level 3 (Blockchain Attested) computation.

2.7 Evaluation

In order to present empirical evidence of AGAPECert effectiveness, we develop three experiments to benchmark critical components of AGAPECert. We start by evaluating the trusted compute engine performance with a mix of Trust Level 1 and Trust Level 2. Then, we discuss the results of generating 1000 PACs for different input sizes (a complete workflow performance evaluation). Finally, we evaluate the Blockchain-Gateway with our *pacContract* and Hyperledger Fabric performance. All experiments are run 1000 times. All the code of experiments can be found at https://github.com/agapecert.

2.7.1 Experiment setup

To show AGAPECert usability and flexibility, we utilize a commodity HP Pavilion Laptop with an Intel Core i5-8250u CPU and 16GB of RAM running Ubuntu Linux 18.04 LTS 64-bit Operating System (serves as a compute engine, graph data store, and Apache Spark server). We also use a trusted edge server, Dell R340, with an Intel Xeon E-2186G and 64GB of RAM running Ubuntu Server Linux 18.04 LTS 64-bit Operating System (utilized for data center attestation primitives). The latter has

support for Flexible Launch Control (FLC) and Data Center Attestation Primitives (DCAP.) Serving as the Blockchain-Gateway is a MacBook Pro (15-inch, 2017) with an Intel Core i7 2.8GHz and 16GB of RAM running macOS Catalina Version 10.15.4 and IBM Blockchain Platform 1.0.31 Visual Studio Code Extension.

2.7.2 Trusted Compute Engine Performance

We developed a computation-intensive algorithm—the Monte Carlo approximation—as an Oblivious Smart Contract. The Monte Carlo OSC was instantiated in AGAPECert's compute engines running NodeJS (TL1), in-browser JavaScript (TL1), and C using OpenEnclace API (TL2). Besides, we developed equivalent code in Python and deployed it in Apache Spark version 3.1.0.

AGAPECert's compute engine and Apache Spark

We compared AGAPECert's compute engine against Apache Spark version 3.1.0. The goal of this experiment is to provide a context of comparison to AGAPECert; Apache Spark will scale and perform better at a massive scale. However, AGAPECert includes use cases in which preserving data ownership is critical. Figure 2.4 shows that AGAPECert's compute engine performs better than Apache Spark using this computationally-intensive Monte Carlo OSC. It is worth noting that Apache Spark's poor performance with two or four executors is due to the use of a synchronized random function in Python, which cannot scale to multiple cores. Nonetheless, a single executor (spark naive) provides a better comparison against AGAPECert's compute engine. Future AGAPECert releases can integrate Apache Spark [45] and Opaque [21] for scalability guarantees.

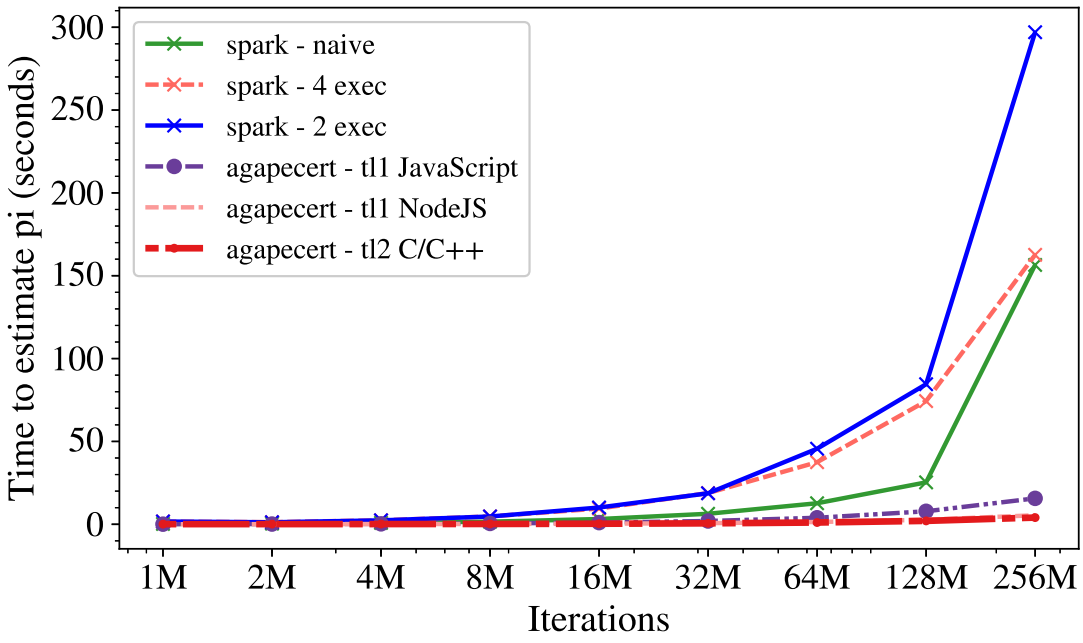


Fig. 2.4.: AGAPECert's compute engine (TL1, TL2) and Apache Spark (1,2,4 executors). We developed the Monte Carlo Approximation Algorithm for those compute engines.

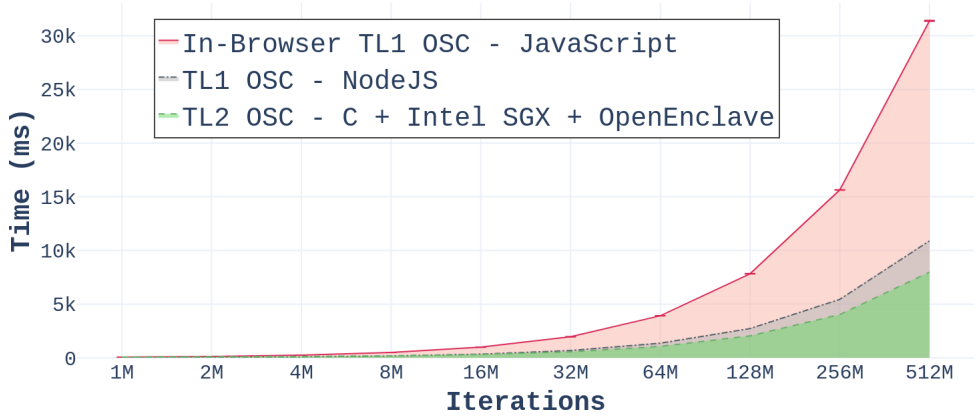


Fig. 2.5.: Compute Engine Performance. We utilize the Monte Carlo Approximation Algorithm. The *Monte Carlo OSC* is instantiated in AGAPECert for Trust Level 1 ($tl1$) for an in-browser computation, Trust Level 1 ($tl1$) NodeJS compute engine, and Trust Level 2 ($tl2$) with Intel SGX and OpenEnclave.

Trust Level 1 and Trust Level 2 comparison

In Figure 2.5, we observe that the AGAPECert (tl2) compute engine outperforms an in-browser JavaScript compute engine (tl1) with equivalent source code. Hence, AGAPECert (tl2) can offer similar performance to widely used development frameworks such as NodeJS and JavaScript. For some use cases, AGAPECert (tl2) can provide a better performance than such frameworks—even computing on top of encrypted private data. We compare the compute time exclusively. An extensive study of *ocalls* and *ecalls* performance is shown in [46].

2.7.3 Private Automated Certifications Performance

This experiment shows the total time needed to generate a PAC using the K-means clustering algorithm deployed as an Oblivious Smart Contract (OSC.) The K-means algorithm is setup with n in increments of $2^i * 1000$ where $i = 0, 1, 2, 3, 4, 5$; k is set to buckets of 250 items per cluster ($k = n/250$); k centroids are determined randomly. The total time includes enclave creation, communication with the graph-based API, computation on private-data, PAC generation, and PAC storage in the graph data store. An asynchronous blockchain access stores the PAC in the blockchain (Fig. 2.7).

The purpose of this *K-means OSC* is two-fold: (1) it presents a widely used clustering algorithm for replicability; (2) solves our example applications (2.6.1)—certified fishing catch area and similar use cases—with a straightforward modification. The regulator fixes the set of centroids, the algorithm to compute the distances, and the threshold that determines if the PAC has passed the evaluation/certification process. As mentioned before, this K-means OSC will be agreed upon by both the regulator and the data owner. The K-means OSC will contain all the semantics that allow the correct validation of data to generate an objective—according to the specification—PAC that can be audited in the future.

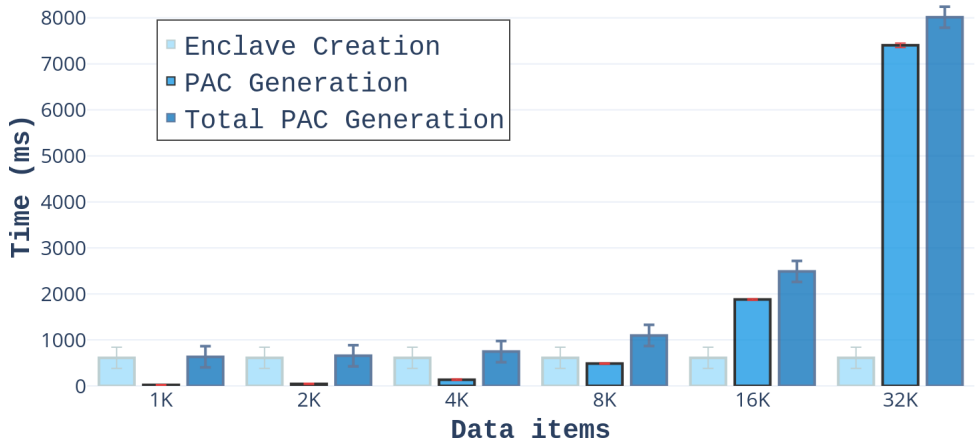


Fig. 2.6.: PACs' generation performance evaluation for Trust Level 3 (*tl*3). The K-means algorithm is instantiated as an Oblivious Smart Contract (*K-means OSC or oblivious K-means if you will*).

Figure 2.6 shows that the PAC generation is bounded by the input data retrieved from the graph data store. Additionally, the running-time in the blockchain and enclave creation is approximately constant.

2.7.4 Blockchain-Gateway Performance

AGAPECert interacts with a Blockchain-Gateway for Trust Level 3 (§2.5.1). We created a test suite to measure the Blockchain-Gateway performance using the Chai assertion library [47] and the Mocha test framework [48] (Figure 2.7). When there exist multiple blocks in the shared ledger, the Blockchain-Gateway takes around $2180.88 \pm 38\ ms$ to execute an asynchronous PAC creation in the ledger.

2.8 Conclusion

This paper presented AGAPECert, an auditable, generalized, privacy-enabling certificate framework that protects the confidentiality of data, participants, and code. AGAPECert utilizes a unique mix of blockchain technologies, trusted execution environments, and a real-time graph-based API to define for the first time Oblivious

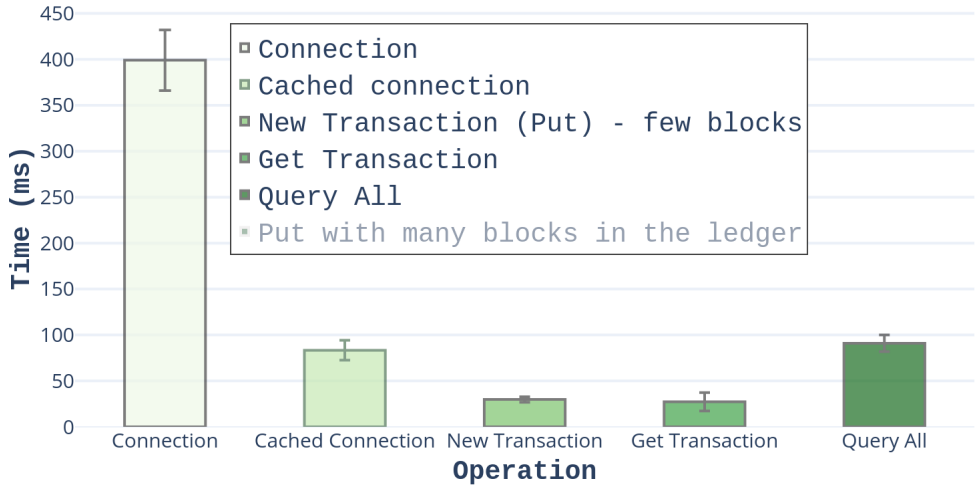


Fig. 2.7.: Blockchain-Gateway interacting with IBM Hyperledger Fabric and *pacContract* performance.

Smart Contracts (OSCs) that generate auditable Private Automated Certifications (PACs). AGAPECert offers pragmatic performance and is generalizable to many use cases and data types. AGAPECert has a significant impact providing an open source [49] framework that can be adopted as a standard in any regulated environment to keep sensitive data private while enabling an automated workflow.

2.9 Acknowledgements

Sponsorship for this work was provided by Foundation for Food and Agriculture Research (FFAR) under award 534662. The work was submitted for publication with the following list of authors: Servio Palacios, Aaron Ault, James V. Krogmeier, Bharat Bhargava, and Christopher G. Brinton.

Table 2.2.: Summary of Trust Levels requirements and security guarantees.

Level	**Explanation**	Requirements		Security Guarantees		
		SGX	**Blockchain**	auditable	independently attestated	provable sequence
Trust Level 1 $tl1$	Owner Attested	X	X	✓	X	X
Trust Level 2 $tl2$	Enclave Attested	✓	X	✓	✓	X
Trust Level 3 $tl3$	Blockchain Attested	✓	✓	✓	✓	✓

Table 2.3.: Summary of Important AGAPECert Cryptographic Hashes.

Hash Name	**Input**	**Objective**
$Data\_Hash$	$private\_data$ retrieved from a Trellis data store	Integrity of Private Data
$Report\_Hash$	$REPORT$ produced by an enclave when running the OSC	Integrity of the REPORT from the enclave
$Quote\_Hash$	$QUOTE$ produced by a Quoting Enclave	Integrity of the QUOTE from the enclave
PAC_i_Hash	PAC_i (JSON object) produced by the enclave interior	Integrity of the PAC
OSC_Hash	OSC Software Code in the Trusted Code Repository	Integrity of the OSC code itself

Table 2.4.: Summary of Components.

Components	**Explanation**
Compute Engine	Compute node controlled or trusted by the data owner. Runs the OSC.
Data Store	A Graph Data Store that holds the private data (Trellis).
Broker	A web-app that initiates, authorizes, provisions, moderates, monitors, validates OSC execution.
Validator.	A web-app that can validate a given PAC
Attestation Service	The Intel SGX attestation service or Data Center Attestation Primitives (DCAP)
Blockchain-Gateway	A Generalizable Blockchain Service to connect to a mix of ledgers as needed.

Table 2.5.: PAC business network in the shared ledger.

Property(fabPAC.)	**Type**.
id	String (UUID)
$quoteHash$	String
OTK (optional One-Time-Key)	String (Base64 encoded)

Table 2.6.: Summary of components' repositories for AGAPECert implementation.

Component	**Repository**	**Objective**
Broker	https://github.com/agapecert/broker	OSCs' Service Manager
Validator	https://github.com/agapecert/validator	PACs' verifier
Compute Engine	https://github.com/agapecert/compute-engine	Compute Engine
Blockchain Gateway	https://github.com/agapecert/blockchain-gateway	Anonymous and asynchronous shared ledger accesses

3. AUDITGRAPH.IO: AN AUDITABLE AND AUTHENTICATED GRAPH PROCESSING MODEL

Many modern big-data applications work on top of graph-structured datasets. Modeling large-scale network-structured datasets require graph processing and management systems such as graph databases. For particular use cases —multi-modal knowledge graphs, electronic health records, finance— network-structured datasets can be highly sensitive and require controlled computation in a trusted environment, i.e., the traditional cloud computation is not suitable for highly sensitive datasets. Further, in regulated environments, auditable computation is imperative.

Thus, we propose AuditGraph.io, an auditable and authenticated graph processing model that performs auditable computation on authenticated graph-structured data on a trusted or controlled node. AuditGraph.io exploits access locality of traversal algorithms; it utilizes trusted execution environments and blockchain technologies to provide authenticated access to subgraphs and provide auditable proof of correct code execution. To the best of our knowledge, AuditGraph.io is the first solution for auditable, authenticated, and integrity-preserving computation for network-structure data exploiting the access locality of traversal algorithms.

AuditGraph.io provides a tool that can be used to track computation on authenticated network-structured data or subgraphs. An enormous amount of use cases can benefit from AuditGraph.io. In particular, we integrate our solution with AGAPECert to provide auditable checkpoints of network data for certification frameworks in the supply chain.

3.1 Introduction

There is an increasing need for secure distributed computation on top of graph-structured data. Graphs[1] are data structures used to represent relationships between entities. Many modern big-data applications work on top of graph-structured data. For instance, graph databases such as Neo4j [1], JanusGraph [2], TruenoDB [3] and others [4–7] handle large-scale graphs that model biological interactions, social networks, knowledge graphs, etc. These systems manage large graphs and execute traversal algorithms on them. Traversal algorithms exhibit access locality in the graph structure [50]. Although current graph databases such as Neo4j, ArangoDB, and others offer protection for data-in-transit [8,9]—utilizing SSL/TLS— and for data-at-rest —using Bitlocker or encrypted key-value stores; those do not offer fine-grained authentication and auditability guarantees at the graph structure level (for example, subgraphs, motif).

Some solutions to compute on top of encrypted graph-data has been proposed. For instance, Xie and Xing (2014) contributed CryptGraph, which uses homomorphic encryption to compute graph analytics on top of graph-structured data [51]. Similarly, some solutions propose the use of trusted execution environments. Trusted execution environments (TEEs) are a hardware-based solution for trusted computation on an adversarial environment—where operating system, kernel, virtual machines are possibly malicious [18]. Intel SGX is one example of TEEs computing in an isolated region called enclave. Zheng, Dave, Beekman, Popa, Gonzalez, and Stoica (2017) created Opaque, a distributed analytics platform built on top of Apache Spark computing on encrypted data utilizing Intel SGX enclaves [21]. Those solutions are cloud-based and not auditable. Further, for extremely sensitive private data, bulk copying confidential graph-structured data—even in an encrypted form—to the cloud is not an option. AuditGraph.io applies authentication algorithms on top of subgraphs and provides auditable computation and checkpoints utilizing trusted execution environments to

[1]For the rest of the paper the terms graph and network are interchangeable

sign data structures that serve as a proof of correct code execution under a particular authenticated subgraph.

In this paper, we are also interested in the benefits of exploiting access locality exhibited in graph traversal algorithms. AuditGraph.io exploits access locality of network data to build a sequence of semantically related blocks that facilitate the access to neighborhoods of data while reducing read access. Some solutions proposed block-based disk layouts. Yasar, Gedik, and Ferhatosmanoglu (2017) presented a scalable block formation and layout technique for graphs that aim to reduce the I/O cost for disk-based processing algorithms [50]. They divide the graph network into a series of disk blocks. These blocks contain subgraphs with high-locality. Also, Hoque and Gupta (2012) introduced Bondhu as a technique to store a social graph on disk. Bondhu's layout scheme reduces the number of seek operations fetching multiple friends' data in a single seek [52]. Soule and Gedik (2016) introduced an adaptive disk layout, i.e., Railway [53], for optimizing disk storage for interaction graphs. Although these techniques showed significant improvement when querying network-structured data from the disk, they do not take into account graph-data from different data-owners (e.g., a variety of authenticated entities) sharing subgraphs among them. Moreover, the distribution of data is usually carried out by a possible untrusted third-party. In particular, we are interested in network-structured data that encode sensitive information, e.g., electronic health records, social networks, multi-modal knowledge graphs, finance, etc. Sensitive information necessitates confidentiality and integrity protection. AuditGraph.io validates and audits the computation on network data utilizing trusted execution environments and blockchain technologies.

AuditGraph.io includes the authentication and integrity protection of subgraphs (structural integrity). Proposed solutions for graph authentication include Kundu and Bertino (2010), which introduced schemes to verify the authenticity or integrity of graph data without leakage [54–56]. They rely on the structure of the network and aggregate signatures. However, there is no solution for disk-based processing algorithms (i.e., those processing large graphs) or implementation in a state-of-the-

art graph database. In this paper, we propose a novel auditable, integrity-protected, authenticated, and block-based graph processing model. The project includes a set of unique algorithms and cryptographic techniques (e.g., hash functions, DFS traversal, and digital signatures) that model the authentication scheme and graph structure into a series of blocks with high-locality. Further, AuditGraph.io audits the structure of authenticated network data and ensures correct graph-analysis code execution using Intel SGX remote attestation. AuditGraph.io stores a minimal set of cryptographic hashes in a shared ledger to provide time-sequence proofs that can be audited in the future.

3.1.1 Problem statement and motivation

Current tools to analyze multiple sensitive network structure datasets are limited and require bulk downloads and manual processing. This represents a risk of exposing private data in transit or during computation. AuditGraph.io aims to query and analyze a shared sensitive graph $G = (V, E)$ (§3.3.2) that contains various levels of authentication and authorization, e.g., a multi-modal knowledge graph, electronic health records, finance, etc. AuditGraph.io allows a user U_i to query a shared graph $G = (V, E)$ composed of multiple individual authenticated sets (subgraphs.) U_i can get an auditable aggregate result of the computation on the graph G (Fig. 3.1), i.e., an analysis of the graph G. Also, a derivative (crytographic hashes) of the recent analysis is stored into a shared ledger. The cryptographic hashes stored in the ledger do not convey any identifiable information about the computation or data, but the hashes serve as proof of the structure of the graph and code that can be audited in the future.

3.1.2 Design Principles

AuditGraph.io must support the following features to provide authentication and auditability guarantees on top of graph-structured data:

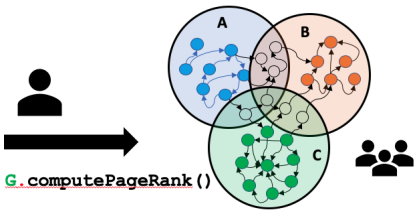


Fig. 3.1.: User U_i queries a shared graph $G = (V, E)$ (§3.3.2) composed of multiple individual authenticated subgraphs. The computation and data can be audited in the future storing untraceable cryptographic hashes in a shared ledger.

1. Users can share their graphs utilizing fine-grained access control.

2. A user U_i should be able to track the computation run on top of their shared subgraph $G_i = (V_i, E_i)$. User U_i is considered the owner for the subgraph G_i.

3. An auditor must be able to verify the correct code execution and subgraph utilized in a particular computation or process.

3.1.3 Hierarchical structure of trust

AuditGraph.io defines hierarchical layers of trust (Table 3.1) that allow a flexible framework to solve a myriad of use cases with different authentication and authorization requirements.

1. **Public**: All the users in the system can access subgraphs marked as public. Also, any algorithm is allowed to run on top of public data.

2. **Protected**: For subgraphs marked as protected, AuditGraph.io provides integrity protection. When querying protected subgraphs, the query engine returns a cryptographic hash of the structure of the subgraph as part of the result. Similarly, the compute engine checks the integrity of the subgraph.

3. **Restricted**: A query to a subgraph S_i marked as restricted includes a digital signature of S_i. When running graph analysis on top of S_i, the compute engine checks the subgraph S_i structure and the digital signature associated with S_i.

4. **Confidential**: A query to a subgraph S_i marked as confidential, must include integrity protection, a digital signature, and a record of access and computation in a shared ledger through the blockchain-gateway.

5. **Secret**: Secret subgraphs require the strongest guarantees. First, the blocks that contain secret subgraphs will be stored in different portions of the memory. The blocks can also be encrypted using 128-bit AES in GCM mode. Additional protection can include hiding access patterns using Oblivious RAM [57]. The query must return a digital signature that ensures the authenticity and integrity of the subgraph. Similarly, the compute engine will use a trusted software code (pre-approved) inside a trusted sandbox, i.e., an enclave. The compute engine will check the authenticity of the software code using remote attestation. Finally, cryptographic hashes of the subgraph structure, quote of the software code, and result will be stored in a shared ledger through the blockchain-gateway. This layer provides proofs of correct code execution and a digital signature of the structure of the graph utilized in the graph analysis.

It is important to note that it is possible to mix different layers of trust to solve multiple use cases. AuditGraph.io implements these layers of trust to improve performance when accessing less sensitive-data.

3.1.4 Contributions

In summary, our contributions are:

- The definition of the practical and formal security model of authentication for the block-based graph.
- Three new algorithms for
  - block construction with high locality,
  - disk/memory layout for authenticated and block-based graphs,

  - unique data structures and programming API for block-based graph access.

- Auditability features for the graph model.

- An *open source* implementation and evaluation of the proposed techniques (AuditGraph.io[2]).

3.2 Background

3.2.1 Data Integrity of Graphs

Graph databases often require the validation of the structure of a graph or subgraph [55]. Similarly, some applications require validating if two subgraphs are the same or if the network structure has changed since a time t. To tackle this, we utilized Arshad et al. (2013)'s contributions to provide collision-resistant hash functions for graphs that ensure the integrity of a graph [55]. In the Table 3.2, we observe the different cryptographic hashes that AuditGraph.io utilizes to contribute auditability and integrity on top of the graph model.

3.2.2 AGAPECert

AGAPECert [58] is an Auditable, Generalized, Automated, and Privacy-Enabling Certification Framework. AGAPECert provides privacy-preserving guarantees utilizing trusted compute engines and graph data stores. Also, AGAPECert exposes auditability features utilizing a generalized Blockchain-gateway and trusted execution environments. AGAPECert intends to automate the certification process utilizing Oblivious Smart Contracts that generate Private Automated Certifications. We integrate AuditGraph.io into AGAPECert to provide auditable and private checkpoints of network data and computation utilized in the certification process. In the case of an audit, the auditor will use those checkpoints to determine if the data and structure of

[2]https://github.com/AuditGraphDB

the data (subgraphs) match the specification stored in the PAC (Private Automated Certifications.)

Private Automated Certifications: PACs are defined as "the derivative output of the clients data (i.e., the contents of the "form" that the regulator requires the data owner to fill out)" [58].

Oblivious Smart Contracts: OSCs is the "regulator-approved code which, given access to private input data, produces the desired PAC (i.e., the "questions" on the regulator's "form")" [58]. We utilize a similar concept to OSC to compute on top of the graph model. AuditGraph.io utilizes a pre-approved software code to protect data ownership and data confidentiality.

3.2.3 Real-time Graph-based API

We utilize the Trellis Framework, which exposes standardized REST API semantics to interact with a user's private data store. This section is based on [58] and provides a high-level overview[3]. The purpose of Trellis is to enable standardized, automated, permissioned, ad-hoc, point-to-point data connections through a common REST API. There are important features of Trellis necessary to understand AuditGraph.io and AGAPECert [58]:

- *Resource Discovery*: Filesystem-like graph schemas to define where data can be discovered. For example, catch locations for a fishing vessel for July 25, 2020, could be defined as discoverable at graph path /bookmarks/trellis/fishing/catch-locations/day-index/2020-07-25.

- *Write Semantics*: Trellis standardizes how data within a graph is written: all data changes are reduced to an ordered stream of idempotent merge operations[4]. Operation ordering is only guaranteed per-resource, not globally.

[3]please refer to the Github project (https://github.com/trellisfw) for more information.

[4]"An idempotent merge operation means that a given JSON document is produced that only affects matching keys. Keys that do not exist are created, existing ones are deep-replaced at overlapping key paths, similar to a common upsert. Applying the same merge repeatedly results in the same resource state at the mentioned key paths, hence why it is idempotent" [58].

- *Change Feeds*: Clients can register for real-time change feeds for any arbitrary subgraph of data. This provides both a real-time communication channel as well as a means of concurrency-safe 2-way data synchronization. The change feed is comprised of the ordered stream of idempotent merge-operations.

- *Authorization*: Trellis standardizes how any client registers and obtains authorization tokens at any Trellis platform.

- *Permissions*: Trellis standardizes how data can be locally shared within a platform.

3.2.4 TruenoDB

TruenoDB [3] is an easy-to-use scalable graph datastore and compute engine. TruenoDB integrates a scalable distributed search engine, i.e., Elasticsearch [59], graph-parallel computation, i.e., Spark GraphX [60], and a set of drivers. TruenoDB implements schema-less graph storage, where vertices, edges, and properties are stored as Apache Lucene objects. TruenoDB integrates query capabilities via Gremlin traversal graph language, in-memory single host compute engine utilizing NetworkX library [61], and network visualization through Sigma.js [62]. AuditGraph.io can enhance TruenoDB providing auditable and privacy-preserving computation utilizing trusted execution environments and blockchain technologies.

3.2.5 Graph Authentication

Kundu and Bertino (2010) proposed two schemes (i.e., one for DAGs and one for a graph with cycles) to verify the authenticity of data without leakage [54–56]. Both methods rely on the structure of the network and aggregate signatures. Therefore, the security of these schemes depends on the computational Diffie-Hellman (that is, aggregate signatures) and random oracles based cryptographic hash functions. Also, they defined the *structural authenticity* as the integrity of relationships in the

network, and *content authenticity* as the authenticity of the content of nodes [54–56]. Consequently, an authenticity scheme for graph structure must preserve structural and content authenticity [54–56]. According to [54–56], confidentiality implies the following:

- A user, auditor, receiver, or prover that verifies data authenticity receives just the structural information and vertices that it is authorized to access.
- A client should not infer or receive any information about the presence of nodes, content, and structural relationships that it is not authorized to access.

The information that the user must not be allowed to infer or receive (according to the authorization level) is called *extraneous information* [54]. AuditGraph.io utilizes these definitions and algorithms to provide authentication for subgraphs in G, but it extends the definitions to include a block-based graph representation to improve access locality and diminish I/O.

3.2.6 Trusted Execution Environments

Trusted Execution Environments (TEEs) aim to enhance the confidentiality of data and computation. TEEs are hardware-based creating controlled and isolated regions called enclaves to compute securely even in the presence of a malicious kernel, hypervisor, operating system, or drivers [18].

Intel SGX

Intel SGX (Software Guard Extensions) is an implementation of the TEE model providing confidentiality for data and code [18]. Intel SGX implementation uses a private region called *enclave* [18]. Code and private data are injected into an enclave [18]. Intel SGX provides means to verify correct code execution using remote attestation.

Remote attestation: The process of remote attestation includes signing a data structure—REPORT— that contains properties of the data and code running in the isolated environment, i.e., enclave. The signed data structure is called *QUOTE*. Then, the Quoting Enclave signs the QUOTE with a key known only to the processor. This key is not revealed outside of the trusted region of the processor. The QUOTE can be used to attest that a particular graph-analysis or algorithm is running inside an enclave and not an attacker's injected code. Intel provides two methods to verify the QUOTE: Intel Attestation Services or Data Center Attestation Primitives (DCAP.) Hence, remote attestation validates the correct code execution under specific parameters. AuditGraph.io includes cryptographic hashes in the QUOTE, and then stores a derivative of the computation in the shared ledger (*Quote_Hash*, *Graph_Data_Hash*, *Graph_Structure_Hash*, *Subgraph_Signature*). We utilized a similar concept to the OSCs (Oblivious Smart Contracts, §3.2.2), in which the code that runs inside of enclaves is pre-approved by the owners of the graph-structured data. This algorithm or code runs in a trusted compute engine. AuditGraph.io provides proof of correct code execution and structure of the data utilized to obtain a result.

3.2.7 Blockchain technologies

A blockchain is a decentralized and immutable shared ledger [16, 29]. A set of nodes or peers store a chain of ordered transactions protected by an immutable set of blocks of cryptographic hashes. A cryptographic hash in a block is tied to the previous hash enabling one essential feature for auditability, traceability, and accountability features such as integrity preservation of the ordering of events in the digital ledger. A block cannot be changed without altering the output of all subsequent blocks, i.e., immutability. A blockchain provides inherently byzantine fault tolerant [30] independent auditability capabilities typically by placing computational constraints on block content that require brute-force guessing to solve, thereby making it too

difficult for a malicious attacker to game the system since they cannot brute-force guess solutions any faster than non-malicious participants.

One of the first concepts of Blockchain was introduced by Satoshi Nakamoto (2008). Utilizing a peer-to-peer network Nakamoto proposed a byzantine fault tolerant solution for the double spending problem dubbed Bitcoin [29]. The double spending problem states that an agent should not be able to register a transaction on a ledger that spends the same resource(s) twice, even if some participants in the network or the agent itself is malicious. In Bitcoin, Nakamoto constructed an immutable record that derived from a hash-based *proof-of-work*. Participants in the network would race to guess what bytes they need to add to a ledger entry in order for the resulting message to produce a hash that fits a difficult constraint, such as needing to start with a certain number of zeros. The network protocol defines the longest known chain with constraint-satisfying hashes as the correct chain. This proof-of-work protocol and longest correct chain definition provide proof of the transactions/events that happened as long as no malicious participant(s) can control more than 50% of the total number of guesses in the overall race.

AuditGraph.io utilizes a blockchain-gateway to provide asynchronous and anonymous accesses to shared ledgers. AuditGraph.io implementation includes an IBM Hyperledger Fabric, but the blockchain-gateway can be extended to connect to other popular shared ledgers such as Ethereum and many others.

3.2.8 Distance in a graph G

Let u and v be vertices in a graph G. The distance from u to v is the length of the shortest path u to v in G $d_G(u, v)$. If G is disconnected and u and v are in different components, we say that $d_G(u, v) = \infty$.

The distance is a metric space:

$d : V \times V \rightarrow Z^+ \cup \{0\}$

Properties of the distance in a graph G:

1. $d_G(u, v) \geq 0$ and $d_G(u, v) = 0$ if and only if $u = v$
2. $d_G(u, v) = d_G(v, u)$
3. $d_G(u, v) + d_G(v, w) \geq d_G(u, w)$

Eccentricity

The eccentricity of a vertex $v \in V(G)$ is defined as: $e(v) = max\{d_G(u, v)|u \in V(G)\}$

Diameter

The diameter is defined for the whole graph G. $diameter(G) = max\{e(v)|v \in V(G)\}$

Radius

$radius(G) = min\{e(v)|v \in V(G)\}$

Periphery

If $e(v) = diameter(G)$, then v is a peripheral vertex. The set of all those vertex comprise the *periphery* of a graph G.

Central vertex

If $e(v) = radius(G)$ then v is a central vertex. The set of all such vertices form the centre of G.

3.2.9 Similarity Primitives

AuditGraph.io goal is to maximize the graph locality over all authenticated sets (subgraphs). To accomplish this, AuditGraph.io utilizes similarity measures to construct semantically related blocks of graph data.

Block locality

For AuditGraph.io, we utilized the Girvan-Newman [63] algorithm based on the edge betweenness centrality to find communities. Then, we order the blocks utilizing edge betweenness centrality, and the authentication graph determines a final ordering. Similarly, AuditGraph.io can also integrate Yasar, Gedik and Ferhatosmanolu (2017) building blocks using conductance and cohesiveness [50]. Cohesiveness can find communities in graphs. Conductance is the ratio of the number of edge-cuts to the total number in the block [50].

Jaccard similarity coefficient

The Jaccard index measures the similarity/correlation between finite sample sets; the total size of the intersection is divided by the size of the union of the sample sets. When using the Yasar et al. (2017) approach, AuditGraph.io includes the Jaccard similarity coefficient and the betweenness centrality of the individual authenticated sets.

Edge betweenness centrality

Edge betweenness centrality (ebc) is the number of shortest paths that go through an edge in the graph or network [63]. An edge with high betweenness centrality score represents a bridge node or connector between two parts in a network.

3.3 Method: System and Security Model

3.3.1 Security Model

This subsection and other subsections utilize definitions provided by Kundu and Bertino (2010) [54]. AuditGraph.io extends these definitions to include a block-based graph processing model. AuditGraph.io offers structural integrity and authentication. Kundu and Bertino (2010) offered structural and content integrity [54]. For AuditGraph.io use cases, we are interested in providing authentication for subgraphs and blocks that can also be audited. Our threat model includes a trusted compute engine and graph data store.

- *Leakage-free*: a user U_i owner of $G_i = (V_i, E_i)$ cannot infer any information that is extraneous [54] to G_i (§3.2.5). Similarly, the user does not receive any blocks/subgraphs that she/he is not authorized to access.

- *Inference attack*: A user U_i with access to one or more subgraphs $G_i \subset G$, attempts to infer sensitive information about the structure—edges or nodes—from the signature and G_i (e.g., node or edge inference). Analogous, an attacker with access to one or more blocks $B \subset \mathcal{B}$ where $B = \{b_i, b_j, ..., b_k\}$ cannot infer any information about related blocks or subgraphs that she/he is not authorized to access.

- *Data tampering attack*: An attacker or adversary tampers with the structural order, the content, the relation between two nodes (edges), or a set of blocks $B \in \mathcal{B}$.

3.3.2 Data model

Authenticated block-based Graph

A block-based (blocked) connected graph $G = (V, E)$ is a data object; it consists of a collection V of nodes $V(G)$ and a collection E of edges each of which "joins" or

"relates" two of the nodes $E(G)$. A node $v \in V(G)$ represents an atomic unit of data (AuditGraph.io offers structural integrity not content integrity[5]) referred as C_v. The blocked graph $G = (V, E)$ contains a collection of subgraphs $G_i = (V_i, E_i)$ (where $i = 1, 2, ..., k$) that are shared with different users as a result of a query on the graph $G = (V, E)$.

$$G_i = (V_i, E_i) \subseteq G = (V, E)$$

Also, $V(G_i) \subseteq V(G)$ and $E(G_i) \subseteq E(G)$.

Individual Authenticated Set or Subgraph (IAS)

Each subgraph $G_i = (V_i, E_i)$ represents an individual authenticated set of the graph $G = (V, E)$. Figure 3.2 shows individual blocked sets and relationships. Each color represents a subgraph that can be represented as a block or a set of blocks in the graph G.

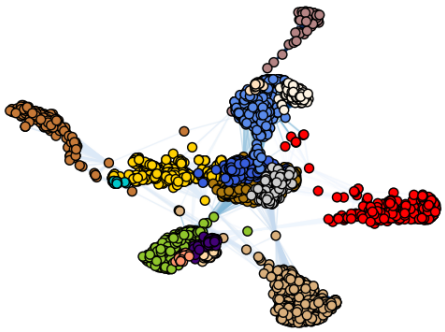


Fig. 3.2.: Authenticated-Blocked Graph example. The graph G is pre-processed into a series of blocks. Each color represents a subgraph. A subgraph is composed of at least one or a set of blocks in the graph G.

[5]AuditGraph.io does not offer confidentiality for the content C_v as in [54]; instead, AGAPECert protects the content of the nodes (property graph model.) AuditGraph.io enhances AGAPECert providing structural authentication and integrity for the graph model.

Block

In graph theory, a block is defined as a maximal nonseparable subgraph [64]. Also, every block in G is an induced subgraph of G. In the context of AuditGraph.io, a block b_i consists of one or more induced subgraphs of G, i.e., a *block* is a storage unit that stores nodes and edges of a graph G. Those nodes and edges build communities in the graph G and are structurally related. Blocks are also stored according to the authentication structure reflected in the authentication tree (§3.3.2) or the multi-modal Knowledge Graph (§3.3.2.)

Authentication on graphs [54]

A graph $G' = (V', E')$ is authenticated as a subgraph of $G = (V, E)$ if and only if

- No vertices v' or u' or edge $e'(u', v')$ has been deleted or added to $G' = (V', E')$ in a unauthorized manner.
- None of the following has been modified (i) the content C'_v of any vertex $v' \in V'$, (ii) any edge $e(u, v) \in E'$ (iii) any structural order $u \prec v$ where $u, v \in V'$, and (iv) a block $B \in \mathcal{B}$ containing node and edge information.

Authentication tree

AuditGraph.io uses an authentication tree A for use cases of authentication that follow a hierarchical structure. A Tree $A = (V'', E'')$ is the authentication graph of $G = (V, E)$ if and only if

- Any node $v \in V''$ represents individual authenticated sets—a subgraph— (§3.3.2) in the graph $G = (V, E)$.
- Any edge $e(u, v) \in E''$ represents an *edge cut* $C(u, v)$ between individual authenticated sets u and v in the blocked graph $G = (V, E)$. The edge $e(u, v)$

includes a weight $w(u, v)$ that captures the relationship between a subset of authenticated sets. We use the normalized *edge betweenness centrality* (§3.2.9).

$$w(u, v) = \sum_{(u',v') \in C(u,v)} ebc((u', v')) \tag{3.1}$$

Fig. 3.3.: Authentication Tree A for the graph G in Figure 3.2. The authentication tree is utilized for a hierarchical access structure. For more complex authentication and authorization graph data accesses the multi-modal Knowledge Graph is used (§3.3.2).

Multi-modal Access and Identity Management Knowledge Graph (AIMKG)

AuditGraph.io utilizes a multi-modal Knowledge Graph for highly evolving needs and richer semantic definition.

The multi-modal Knowledge Graph includes (Figure 3.4):

- **User Profiling**: contains historical data of user queries and access to blocks or subgraphs. Additionally, the property *frequency* of accessed blocks in the user profile can serve as a method to cache frequently accessed blocks, i.e., AuditGraph.io can implement enhancements on cache libraries [65,66] inferring useful information for similar users. Hence, relevant content can be pushed to users when analyzing a subgraph.

- **Security Clearance Graph (SCG)**: The security clearance graph is a DAG that defines the required clearance to access blocks of data or subgraphs. Similarly, the SCG determines the security clearance needs to run a graph analytics algorithm such as PageRank. For instance, more sensitive datasets will require correct code execution and auditability. However, the requirement of proper

code execution can be too restrictive for subgraphs or blocks *marked as* public. Therefore, AuditGraph.io provides rich graph processing techniques allowing a variety of algorithms to run on graph-structured data according to the security clearance.

- **Similarity Graph (§3.4.1)**: the *similarity graph* is the coarse graph representation of the semantically related individual blocks or subgraphs. The *similarity graph* is integrated into the AIMKG to relate individual blocks and groups with security clearance.

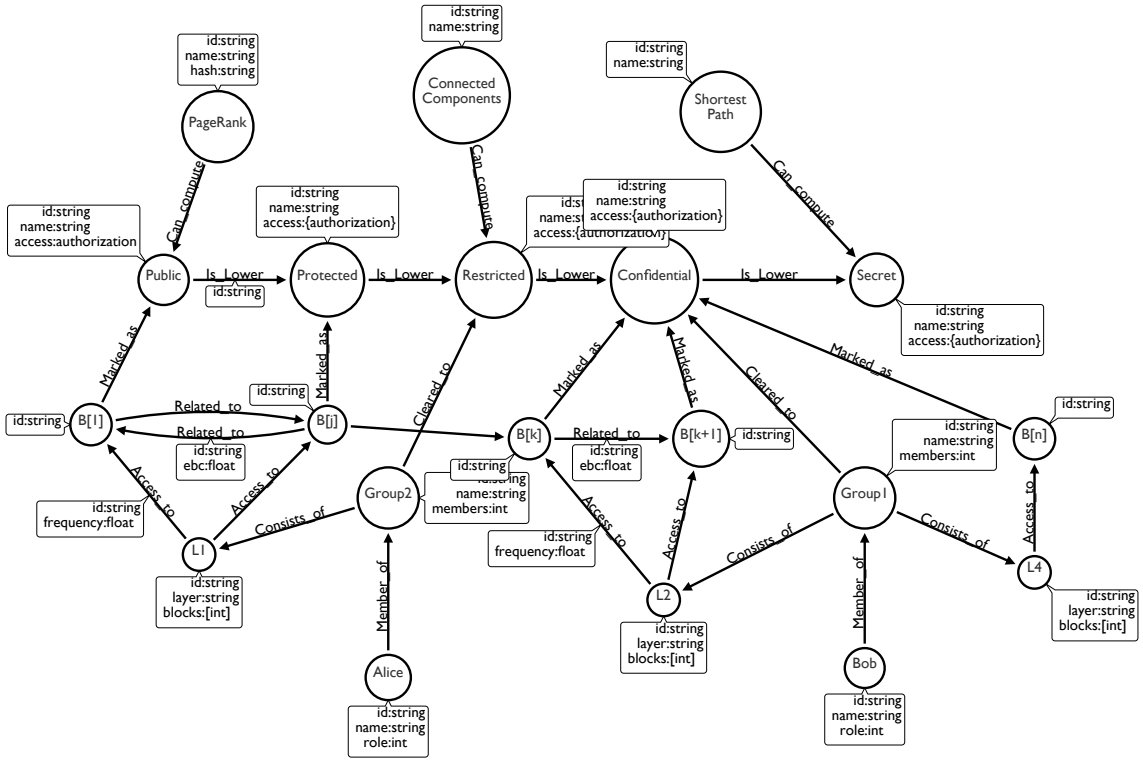


Fig. 3.4.: Multi-modal Knowledge Graph summarized data model.

3.3.3 System Architecture

AuditGraph.io includes the following components (Fig. 3.5):

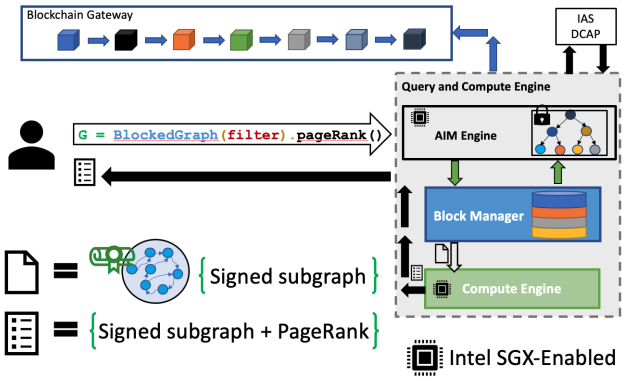


Fig. 3.5.: System Architecture for AuditGraph.io module. We implemented AuditGraph.io as a Trellis module.

- **Graph data store**: A trusted graph data store that stores the private graph data or a sensitive shared graph G.
- **Block manager**: A module that implements the algorithms described in Section 3.4.
- **AIM engine**: An authentication and authorization engine that utilizes the authentication tree (§3.3.2) or the multi-modal Knowledge Graph (§3.3.2) to provide a rich and evolving interaction with the *block manager*.
- **Blockchain-gateway**: The Blockchain-gateway provides asynchronous and anonymous shared ledger accesses. The Blockchain-gateway exposes an API that allows pluggable shared ledgers, i.e., IBM Hyperledger Fabric, Ethereum, etc.
- **Attestation service**: This is the Intel Attestation Services or Data Center Attestation Primitives (DCAP) to ensure correct code execution guarantees.

Before querying the graph G, an administrator pre-processes the graph utilizing the processing model for block creation explained in Section 3.4. A pre-defined set of authenticated sets facilitates the pre-computation of cryptographic hashes that AuditGraph.io uses as random oracles when querying the graph. This way, our method

differs from [54] pre-computing cryptographic hashes of the structure of the blocks. Moreover, we also utilize a trusted compute engine that provides proofs of correct code execution and stored minimal set of cryptographic hashes that can be audited in the future.

The process of querying and computing in the AuditGraph.io is as follows (Fig. 3.5):

1. User U_i queries the AuditGraph.io filtering the graph G.

2. The *AIM engine* will determine U_i's credentials using the Encrypted Authentication Tree (§3.3.2) or the multi-modal Knowledge Graph (§3.3.2) that resides inside an enclave. The AIM Engine queries the *AIMKG* to obtain a set of blocks that the user U_i is authorized to access or retrieves *NOT AUTHORIZED* as response to the user.

3. If authenticated, the process proceeds to obtain the blocks of graph data authorized for U_i. The block manager constructs a subgraph S_i from the set of blocks retrieved.

4. The block manager signs the subgraph S_i and sends it to the *Compute Engine*. Note that S_i can be encrypted and the *AIM Engine* determines if the algorithm for that subgraph is valid.

5. The Compute Engine instantiates the requested algorithm inside an enclave and send S_i to the enclave trusted part. The computation obtains the aggregated result or analysis of S_i.

6. Cryptographic hashes of the network data and computation are stored in the shared ledger.

7. The signed results (the result is a derivative of the graph analytics algorithms run on top of subgraph S_i) are sent back to the user U_i.

3.4 Block formation and layout

The algorithms presented in this section will serve to pre-process a sensitive graph G composed of potentially many sensitive subgraphs shared by a variety of users. Later, any user can query G according to his/her security clearance (§3.3.2 and §3.3.2).

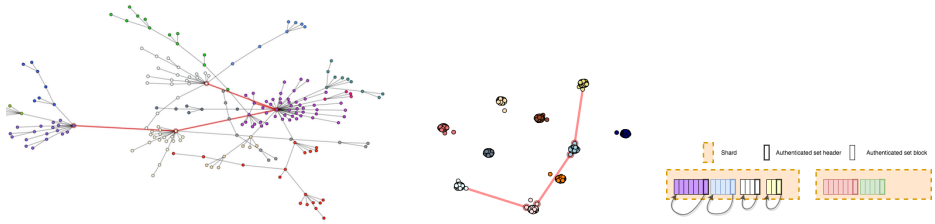
3.4.1 Authentication graph and layout ordering

Similar to Yasar et al. (2017), AuditGraph.io aims to generate locality-aware blocks. In contrast to Yasar et al. (2017), we use a novel technique that exploits the structure of the authentication tree or multi-modal Knowledge Graph (§3.3.2 and §3.3.2) to define the ordering in the disk/memory layout. According to Yasar et al. (2017), the I/O efficiency increases and the number of blocks accessed diminishes when using locality-aware blocks [50]. The authentication mechanisms can take advantage of locality-aware block formation to reproduce the authenticated access patterns on blocked-graphs.

Encryption of blocks: AuditGraph.io exploits the locality of traversal algorithms to generate semantically related blocks of graph-structured data. Further, the block structure facilitates the encryption scheme using AES-GCM—authenticated encryption— or AES-CTR.

Authenticated block-based graph construction

Authenticated clients (i.e., entities) create authenticated block-based subgraphs (§3.3). Every authenticated data-graph (e.g., set) is blocked independently using the *block formation algorithm* (§3.4.2). Then, AuditGraph.io implements graph coarsening on the authenticated-blocked graph (Figure 3.6(b) shows an intermediate step). In the coarse graph, individual data-graph represent the nodes, and edges weights model the edge betweenness centrality (Formula 1) among the edge cut between data-graphs



(a) (b) (c)

Fig. 3.6.: (a) Individual authenticated subgraphs joined by high edge betweenness centrality are most likely to be stored in the same/contiguous shard/memory block. (b) Authenticated and block-based graph coarse representation. Red edges represent higher values of the edge betweenness centrality between individual authenticated sets (§3.3.2). (c) Disk/memory layout example for the Figure 3.6(a). Authenticated sets connected by a high edge betweenness centrality are stored in the same shard or memory block with high probability.

(Figure 3.6(b) edges with high betweenness centrality in red). The resulting graph is called *similarity graph*.

In the first algorithm, we modified the *Girvan Newman* partitioning algorithm [63]. AuditGraph.io builds the similarity graph removing the edges with high betweenness and keeping track of the cuts between data-graphs. Then, AuditGraph.io recalculates the betweenness centrality of the network. We proceed as long there are edges in the graph.

To generate the ordering between authenticated sets, we create a novel algorithm (Algorithm 1) that resembles the hierarchical agglomerative clustering. Also, we implement the complete-linkage clustering which provides the maximum of the pairwise dissimilarities between edge cuts (§3.3). Our algorithm stores highly related authenticated sets (i.e., individual sets that shared an edge with high edge betweenness centrality) in the same shard or memory block with high probability (as shown in Figures 3.6(a),3.6(b),3.6(c)).

Layout manager

The layout manager utilizes the layout ordering algorithm (Algorithm 3) which uses the authentication graph (or Knowledge Graph) structure to define the final ordering of blocks inside of the shards or memory blocks. Our algorithm (Algorithm 2) stores highly related individual blocked data-graphs in the same shard or memory block with high probability. For instance, we can observe in Figure 3.6(c) a disk layout representation of Figure 3.2 (we include only the four individual sets with high edge betweenness centrality in the first shard or memory block.)

Layout ordering

The layout ordering algorithm (Algorithm 3) exploits the structure of the authentication tree (§3.3.2) and provides highly related authenticated-sets ordering. This algorithm is utilized when the authentication model follows a hierarchical structure—

Data: S: Nodes represented by individual authenticated sets with their pre-computed cryptographic hashes
Data: d: Pairwise dissimilarities [edge weights represented by the edge cut $C(A, B)$ with weight edge betweenness centrality using formula 3.1]
Data: A: Authentication Graph
Result: Authentication Graph A
$N \leftarrow |S|$;
$size[x] \leftarrow 1$ for all $x \in S$;
for $i \leftarrow 0, ..., N-2$ **do**
    $(a, b) \leftarrow \arg\max_{(SxS)} d$;
    $S \leftarrow S \setminus \{a, b\}$;
    create new node label $n \notin S$;
    update d d[x,n]=d[n,x]= complete_linkage(d[a,x],d[b,x],d[a,b]) for all $x \in S$;
    size[n] $\leftarrow$ size[a] + size[b];
    $S \leftarrow S \cup \{n\}$;
    create new node label $G_{ab} \notin A.V$;
    A.E $\leftarrow$ A.E + $\{G_{ab}, a\}$ + $\{G_{ab}, b\}$;
    A.V[G_{ab}]['size'] $\leftarrow a.V.size + b.V.size$;
    A.V[G_{ab}]['hash'] $\leftarrow sha256(a.V.hash + b.V.hash)$;
end
return A;

Algorithm 1: Generating ordering among authenticated sets.

Data: mb_size: Memory Block Size
Data: S: Nodes represented by individual authenticated sets
Data: d: Pairwise dissimilarities [edge weights represented by the edge cut $C(A, B)$ with weight edge betweenness centrality using formula 3.1]
Data: A: Authentication Graph
Result: Authentication Graph A
$A \leftarrow generate\_authentication\_graph(A, s, d)$;
$layout \leftarrow layout\_ordering(A.A.root)$;
$i \leftarrow 0$;
foreach $l \in layout$ **do**
  foreach $b \in blocks[l]$ **do**
    if $b.size + memory\_blocks[i] \leq mb\_size$ **then**
      $memory\_blocks[i] \leftarrow memory\_blocks[i] + b$;
    else
      $i \leftarrow i + 1$;
      $memory\_blocks[i] \leftarrow initialize\_memory\_block()$;
      $memory\_blocks[i] \leftarrow memory\_blocks[i] + b$;
    end
  end
end
return A;

Algorithm 2: This algorithm is a layout manager utilizing the authentication tree (a similar process can include the AIM Knowledge Graph). We order a set of blocks (subgraphs) in memory blocks as an example. This layout can be extended to model disk layouts using shards instead of memory blocks.

for other use cases, AuditGraph.io uses a multi-modal Knowledge Graph. This algorithm resembles a post-order tree traversal. The running time for this algorithm is bounded by the size of the set of vertices ($|V''|$) and edges ($|E''|$), i.e., $O(|V''| + |E''|)$.

Data: A: Authentication Graph
Data: node: Authentication Graph node
Result: Layout ordering
if *node.left == NULL* **then**
  return node.id;
else
  $L \leftarrow layout\_ordering(A, node.left)$;
  $L \leftarrow layout\_ordering(A, node.right)$;
  return L + node.id;
end

Algorithm 3: This algorithm follows a post-order layout ordering for a hierarchical authentication structure using an authentication tree.

3.4.2 Block formation

AuditGraph.io can modify the Girvan-Newman algorithm [63] to get a hierarchical decomposition of the network and then organize communities in blocks according to the block size threshold.

The block formation algorithm can also use the one found in [50]. AuditGraph.io implements a hierarchical agglomerative clustering, hence the block formation follows a bottom-up fashion. First, each node is in a partition by itself. Then, the process merges pairs of partitions creating bigger partitions. The algorithm choose the pairs that maximize the distance between vertices (i.e., via Jaccard Similarity Coefficient §3.2.9). If the partition exceeds the block size threshold, AuditGraph.io creates a new block.

3.4.3 Authenticated block-based graph API

AuditGraph.io exposes a minimal but functionally rich API used to simulate graphs with high overlap and locality-aware traversal algorithms (Listing 3.1).

Authentication models

AuditGraph.io can generate individual authenticated sets through the *generate_authentication_model*() call. The *GraphType* is an enumeration that defines if the individual authenticated-sets will follow a power law distribution (i.e., using the preferential attachment model), the small-network phenomena, or networks with high–overlap with lattices (e.g., mesh, hypercube, triangular, hexagonal). Also, the API allows to call individual methods and construct personalized authentication models (Listing 3.1 line 9 uses the preferential attachment model, and 10-13 generate lattices).

Block-based traversal algorithms

AuditGraph.io API includes a rich set of traversal algorithms. These algorithms keep track of the number of visited blocks. For instance, we can call *blocked_single_shortest_path*(v) to retrieve all vertices and blocks touched when running the shortest path computation from a single source. Similarly, we include *blocked_BFS*(), *blocked_DFS*(), and *blocked_neighbors*().

```
1 /// blocked graph simulation authorization graph
2 bg = BlockedGraph() //blocked graph
3 bg.generate_authentication_model(1000, 100, 100, 4, GraphType.
    PREFERENTIAL_ATTACHMENT)
4 bg.draw_blocked_graph_manager()
5 bg.plot_values(3)
6 bg.blocked_single_source_shortest_path()
7 // independent graph generation lattices/pa
8 g = BlockedGraph() //graph
```

```
G1=g.generate_preferential_attachment(200, 1)
G2=g.generate_lattice(Lattices.HEXAGONAL, 3, 3)
G3=g.generate_lattice(Lattices.HYPERCUBE, 3)
G4=g.generate_lattice(Lattices.TRIANGULAR, 3, 4)
G5=g.generate_lattice(Lattices.MESH, 8, 8)
g.set_blocked_graph(G1)
partitions = g.partition(G1)
g.draw_blocked_graph(G1, partitions)
```

Listing 3.1: Summarized authenticated blocked graph API. Python source.

Partitioning algorithms

AuditGraph.io exposes a new set of partitioning algorithms. For instance, it can be the case that a network follows the structure of a mesh and nodes in the center are more sensitive (Figure 3.7). For such applications, novel partitioning algorithms based on the *eccentricity* (§3.2.8) of the graph are necessary. Sensitive edges must be exposed by a higher level (a more privileged) access level.

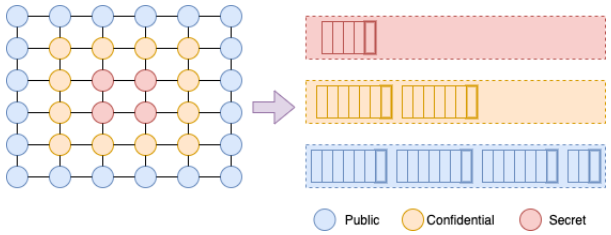


Fig. 3.7.: Coarse representation of a mesh network. Partitioning algorithms based on the eccentricity of the Graph G. Nodes in the center are considered more sensitive and therefore stored in a different shard or memory block than the other set of nodes.

3.5 Security Analysis

3.5.1 Side-channel attacks to the AIM/Compute Engine

Intel SGX can suffer from side-channel attacks [33, 34]. However, the AuditGraph.io trust model includes a trusted compute engine and a controlled environment; therefore, side-channel attacks are out of the scope for this paper. AuditGraph.io trust model targets regulated environments where users want to audit computation and network-structured data access. AuditGraph.io enforces up-to-date remote attestation services, up-to-date microcode, and best practices in place.

3.6 Prototype implementation

AuditGraph.io is implemented as an in-memory module for the Trellis platform. Trellis consists of many docker containers that handle the authentication, authorization, query requests, graph lookup, and many other core components. Trellis is based on OADA and ArangoDB multi-model NoSQL database. AuditGraph.io enhances Trellis providing an authenticated and auditable graph processing system. In particular, we integrated our solution with AGAPECert.

3.6.1 Compute Engine

We implemented the compute engine utilizing a mix of React components and JavaScript APIs for the service manager. For the compute engines that require Intel SGX, we implemented the drivers using C++ and OpenEnclave.

3.6.2 AIM Engine

This AIM Engine includes a connection to a graph database in which the Multimodal Knowledge Graph is stored. We utilized Neo4j's Docker container 4.0.4. Some parts of the knowledge graph are created in the pre-processing stage when initializ-

ing the system. Then, the User Profile interface updates the user's preferences and improves the capabilities of the AIM Engine. The AIM Engine can infer what information or blocks are essential for some users and cache the appropriate information even before the user queries the graph.

3.6.3 Block Manager

AuditGraph.io implements a Python-based block manager utilizing NetworkX 2.4 [61]. The Block Manager provides a richer interface to simulate authentication graphs and block-based traversal algorithms.

3.6.4 Graph pre-processing and integrity

As explained in [67], including timestamps in the Depth-first search (DFS) reveals essential information about the graph's structure. Kundu and Bertino (2010) utilized this notion of timestamps, and they used pre-order and post-order numbers [54, 56]. AuditGraph.io utilizes a DFS traversal to obtain the post-order, and pre-order numbered list of nodes. Then we compute an HMAC-SHA256 derived from the post-order and pre-order numbers that describe or determine graph's structure. The key k for HMAC-SHA256 is determined by the initialization step of AuditGraph.io query processing. This key k can be associated with particular users or groups, and a mixture of keys can be used to provide a final cryptographic hash for a subgraph or a set of subgraphs. A trusted node stores the key or set of keys, and the pre-computation of cryptographic hashes happens inside an enclave. The DFS traversal can reveal the type of edges—tree edges, back edges, forward edges, cross edges; however, AuditGraph.io's trust model assumes a trusted compute engine and graph data store; therefore, we only offer *structural integrity and authentication* for the graph that can be used to audit computation. The pre-computation facilitates the creation of digital signatures when querying the graph. We bound the input size for the digital signature algorithm to a fixed string (the HMAC-SHA256.)

3.6.5 Digital signatures

Since we pre-computed the HMAC-SHA256 of the DFS pre-order and post-order lists, we bound the size of the input (e.g., a fixed string) for the digital signature algorithm improving performance considerably. AuditGraph.io uses the ECDSA signature scheme with the SHA384 hash function.

3.6.6 Blockchain-gateway

When the data is confidential or secret, then AuditGraph.io utilizes a blockchain-gateway to interact with different shared ledgers. For the AuditGraph.io's Blockchain-Gateway we utilized an IBM Blockchain Platform 1.0.31 Visual Studio Code Extension.

3.7 Evaluation

We implemented the AuditGraph.io Block Manager using Python 3.8.3, NetworkX 2.4 [61], METIS 5.1.0 [68]. The trusted compute engine exposes C++ drivers to connect to the graph data store. AuditGraph.io utilizes OpenEnclave 0.9 to provide secure enclaves and remote attestation. Blockchain-Gateway contracts are built using JavaScript.

3.7.1 Experimental setup

AuditGraph.io's simulator and experiments were run on top of a MacBook Pro (15-inch, 2017) with an Intel Core i7 2.8GHz and 16GB of RAM running macOS Catalina Version 10.15.5.

Synthetic datasets We generate data graphs through the preferential attachment model [69] where new vertices attach preferentially to already well-connected nodes (i.e., richer gets richer phenomena). This model reproduces the property of large networks in which connectivities follow a scale-free power-law distribution.

Moreover, the AuditGraph.io API exposes methods to generate small-world networks. We also simulate extreme overlap between data graphs using lattices (e.g., meshes, hypergraphs, hexagonal, and triangular).

3.7.2 Graph Pre-Processing and Integrity

In this experiment, we evaluate the total time exposed by the DFS traversal for various datasets. The DFS traversal pre-order and post-order lists are used as input datasets to compute the cryptographic hash HMAC-SHA256 (§3.6.4). As shown in Figure 3.8, it takes around $80ms$ to compute the pre-order and post-order lists for the anonymized Facebook dataset.

In Figure 3.9, we analyze the total overhead incurred by the pre-computation of cryptographic hashes for the output datasets generated in the experiment shown in Figure 3.8. Analyzing Figures 3.8 and 3.9, it is clear that generating the HMAC-SHA256 depends on the size of the pre-order and post-order lists $O(|V|)$, in which V is the vertex set for those lists.

3.7.3 Graph Integrity and digital signatures

This experiment evaluates the overhead to compute digital signatures derived from the structure of the graph (§3.6.5). Since we pre-computed the HMAC-SHA256 of the DFS pre-order and post-order lists, we bound the size of the input (e.g., a fixed string) for the digital signature algorithm improving performance considerably. AuditGraph.io uses the ECDSA signature scheme with the SHA384 hash function. According to our findings (Figure 3.10), computing a digital signature for a subgraph or block is cheap, i.e., around 2551μs with the P256 curve and the SHA384 hash function.

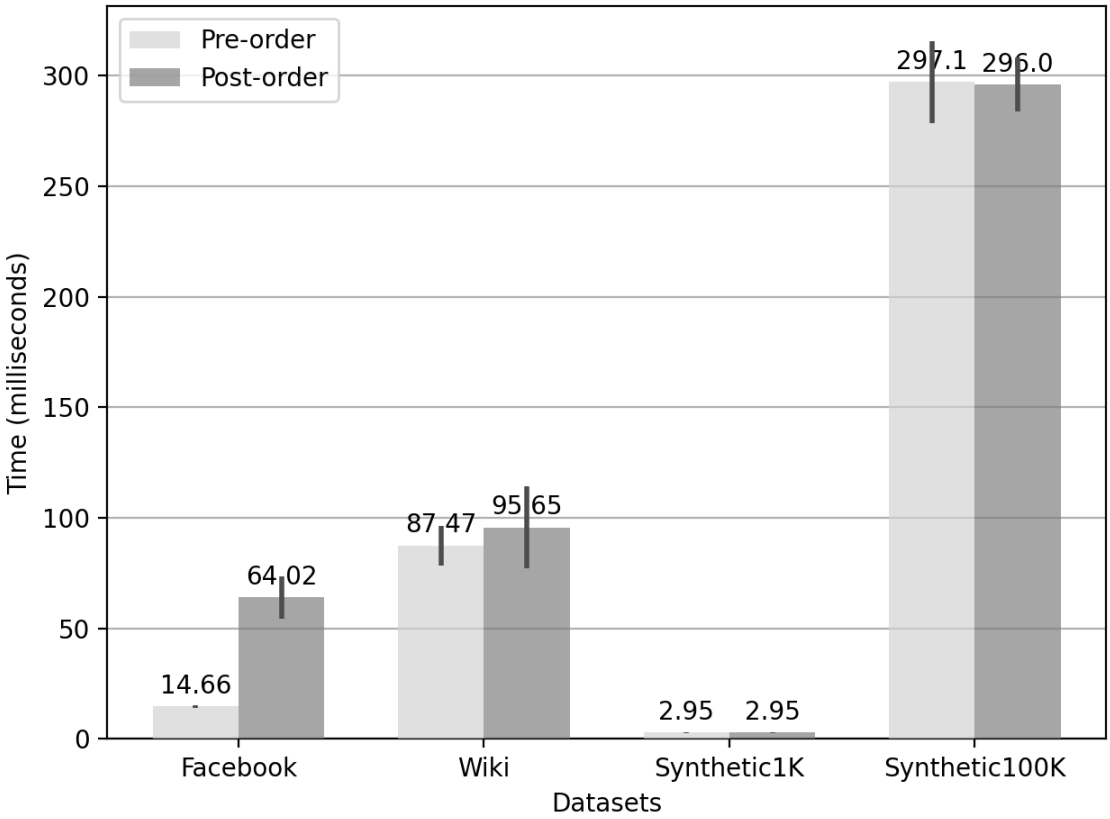


Fig. 3.8.: DFS traversal algorithm in various datasets. The pre-order and post-order lists generated by this algorithm provides essential information related to the graph structure.

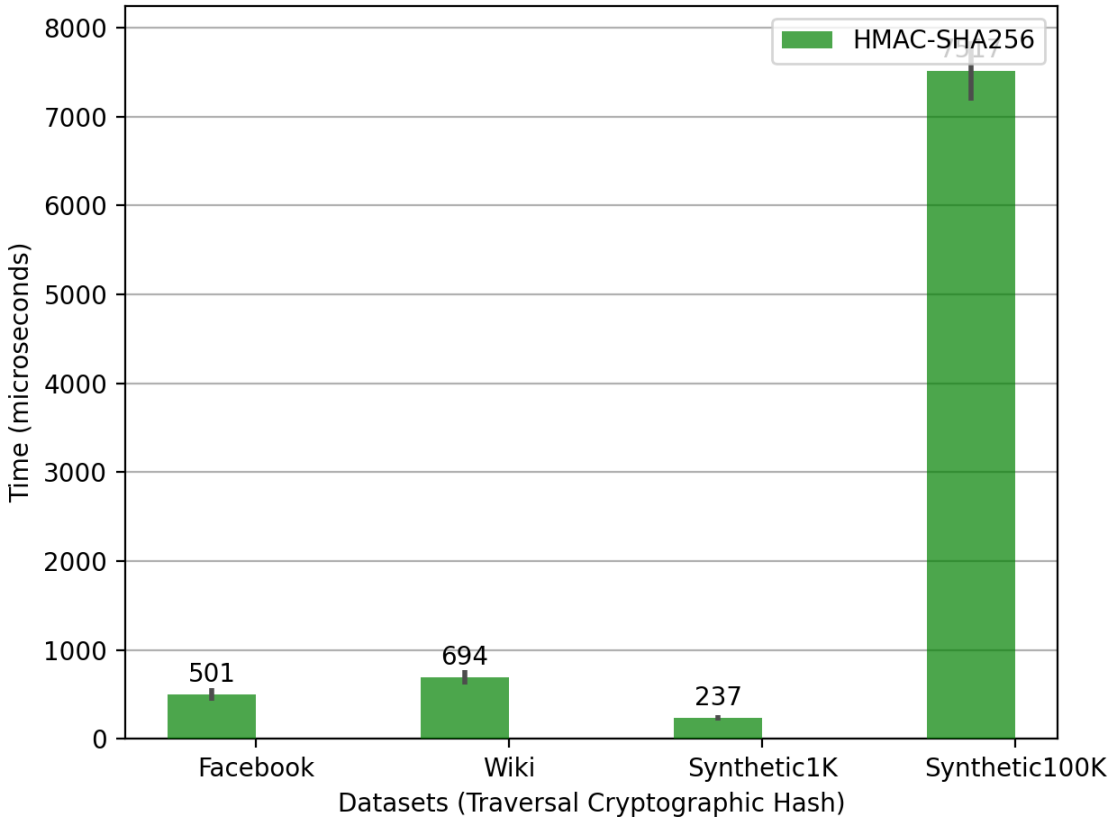


Fig. 3.9.: This experiment measures the overhead of generating cryptographic hashes for the datasets and pre-order and post-order lists generated in Figure 3.8.

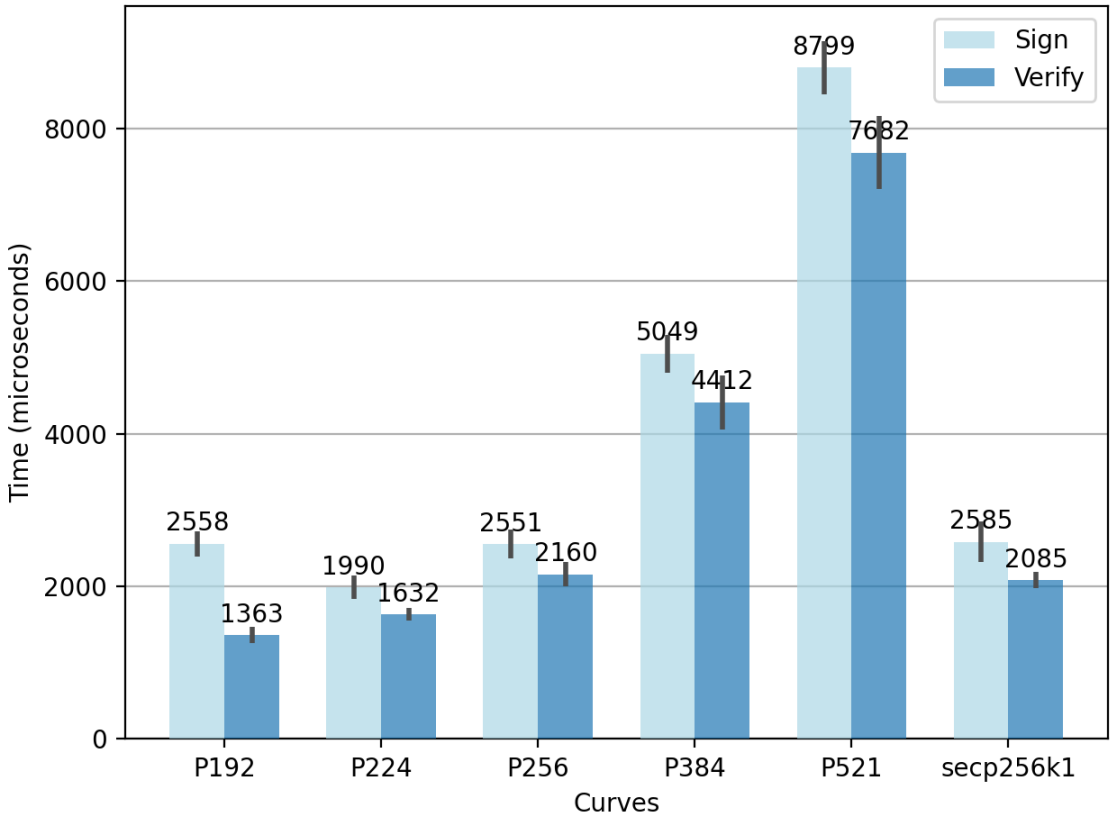


Fig. 3.10.: Evaluation of the digital signature scheme overhead (ECDSA with SHA384) on top of the Facebook dataset.

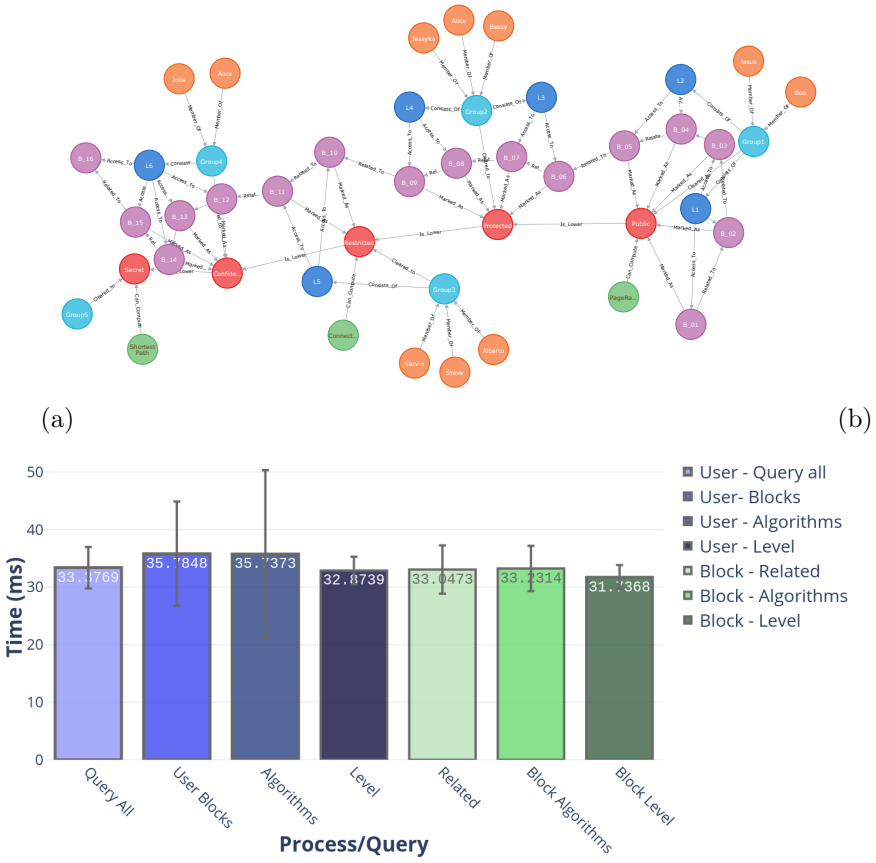


Fig. 3.11.: a) Summarized multi-modal Knowledge Graph Representation. The data model described in Figure 3.4 is materialized using Neo4j graph database. The *AIMKG* includes the *similarity graph*. Also, the edge betweeness centrality is included as a property in the edges. We queried this *AIMKG* to obtain the results in Fig 3.11(b). b) AuditGraph.io's *AIMKG* Multi-modal Knowledge Graph performance. We perform the most common traversal queries—using Cypher—in the Knowledge Graph in Figure 3.11(a).

3.7.4 Multi-Modal Knowledge Graph Performance

In this section, we evaluate the Multi-modal Access and Identity Management Knowledge Graph (*AIMKG* §3.3.2) performance. This experiment includes the anonymized Facebook dataset [70]. The block formation and layout algorithms (§3.4) generated *16* blocks of semantically related blocks (similarity graph §3.4.1.) We then created the *AIMKG*, including the *similarity graph*, the user profiling subgraph, and authentication and authorization subgraph. We materialize the data model (Fig-

ure 3.4) utilizing a Neo4j's Docker Container 4.0.4. Additionally, we integrate this $AIMKG$ module with AGAPECert's broker. The test suite includes the Chai assertion library, the Mocha test framework, and the Neo4j JavaScript driver. The queries are run 1000 times using the bolt driver to connect to Neo4j with a basic authentication scheme[6] (Fig. 3.11(b)).

3.7.5 Neighbors Algorithm Performance

Now we analyze the blocked-based neighbor's algorithm in a power-law graph model (Fig. 3.12). For this experiment, we modified the neighbors' algorithm. We include a block property that serves to divide the nodes into different buckets; then, we count the number of blocks/buckets touched by the neighbors' algorithm. The AuditGraph.io API exposes methods to perform block-based traversal algorithms. In Figure 3.12, we can observe different schemes of graph partitioning and disk/memory layout. The primary benefit of AuditGraph.io is when accessing authenticated graph datasets. AuditGraph.io exploits the access patterns of the authentication tree (§3.3.2) or multi-modal Knowledge Graph (§3.3.2) to produce ordered blocks in shards or memory blocks with high-locality. AuditGraph.io stores blocks with high-locality in the same shard with high probability. The non-blocked partitioning performs poorly compared to our solution.

3.7.6 Blockchain-Gateway Performance

For the AuditGraph.io's Blockchain-Gateway we utilized an IBM Blockchain Platform 1.0.31 Visual Studio Code Extension. Analogous to AGAPECert's experiment, we use Chai assertion library and the Mocha test framework (Figure 3.13). When there exist multiple blocks in the shared ledger, the Blockchain-Gateway takes around $2173.12 \pm 45\ ms$ to execute an asynchronous object creation.

[6]The AIMKG graph construction script and queries can be found at http://github.com/AuditGraphDB

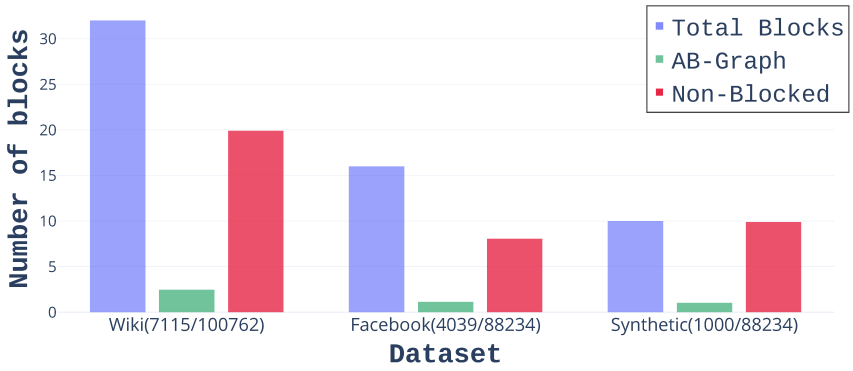


Fig. 3.12.: Neighbors' algorithm performance. This experiment tested three different datasets (Wiki, Facebook, and a synthetic dataset.) We measure the total number of blocks touched by the algorithm on those datasets. AuditGraph.io touches fewer blocks and retrieves more related blocks with high probability.

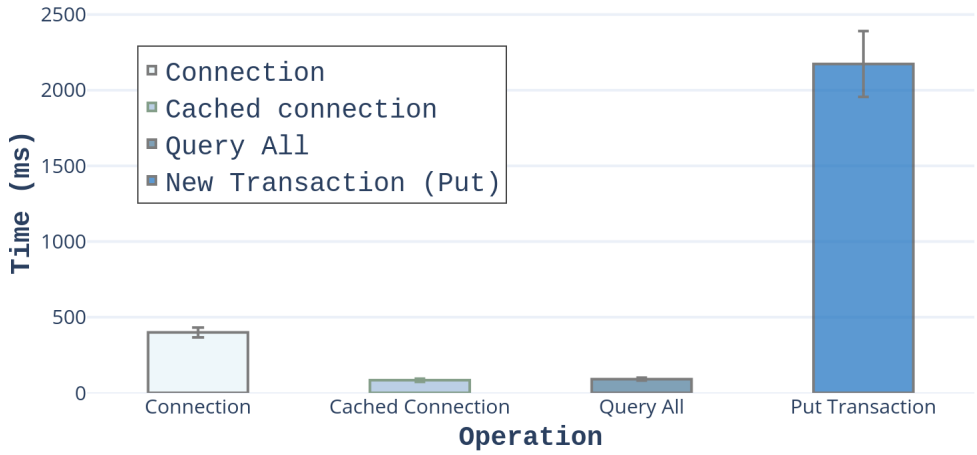


Fig. 3.13.: AuditGraph.io's Blockchain-Gateway component interacting with IBM Hyperledger Fabric and *auditContract* performance.

3.7.7 Discussion

We also ran experiments utilizing AuditGraph.io's API methods to generate small-world networks and the preferential attachment model [69]. Using AuditGraph.io's simulator, we compare the number of buckets or blocks touched by traversal algorithms; AuditGraph.io reduces the number of blocks touched when executing traver-

sals on the graph. These experiments offer empirical evidence of the feasibility of our ideas.

Limitations: Our solution is in-memory only, future work includes testing these techniques using a combination of disk and memory.

3.8 Conclusion

We have presented AuditGraph.io, an auditable, authenticated, and blocked graph processing model. AuditGraph.io exploits the structure of the network to generate blocked individual authenticated sets with high-locality. AuditGraph.io utilizes the structure of the access hierarchy (authentication graph) to define the ordering of blocks and entities in the disk or memory layout. A *similarity graph* with edges representing the relationship between authenticated sets (edge betweenness centrality as weight) determines the final ordering of blocks in shards or memory blocks.

AuditGraph.io provides network-structured data integrity and digital signatures using previous work and our block-based model. Also, AuditGraph.io contributes auditable computation on top of the graph-model utilizing trusted execution environments and blockchain technologies. Our preliminary results and simulation show that AuditGraph.io can reduce the total number of visited blocks and potentially reducing the total I/O cost for disk/memory processing algorithms. Through a unique mix of digital signatures schemes, blockchain technologies, and trusted execution environments, AuditGraph.io provides an auditable, integrity-preserving, and block-based graph processing model. AuditGraph.io contributes a tool that can be used to track computation on authenticated network data or subgraphs.

3.9 Acknowledgements

The research was partially carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration (80NM0018D0004). Also, this work was partially supported by the

National Science Foundation Grant number CCF 1533795, Division of Computing and Communication Foundations under the XPS program. The work is in preparation for publication with the following list of authors: Servio Palacios, Bharat Bhargava, Arun Viswanathan, and Jeremy Pecharich. We also thank Dr. Ananth Grama for fascinating discussions and advice.

Table 3.1.: Summary of the hierarchical structure of trust, requirements, and security guarantees.

Level	**Explanation**	Requirements		Security Guarantees				
		Intel SGX	**Blockchain**	integrity	auth	auditable	ind. attested	sequence
Public	Open	X	X	X	X	X	X	X
Protected	Integrity	X	X	✓	X	X	X	X
Restricted	Authenticated	X	X	✓	✓	X	X	X
Confidential	Audited	X	✓	✓	✓	✓	X	✓
Secret	Attested	✓	✓	✓	✓	✓	✓	✓

Table 3.2.: Summary of Important AuditGraph.io Cryptographic Hashes.

Hash Name	**Input**	**Objective**
Graph_Data_Hash	*private_data* retrieved from a Trellis data store	Integrity of Private Data
Graph_Structure_Data_Hash	*private_data* structure retrieved from the block manager	Integrity of Private Data structure
Report_Hash	*REPORT* produced by an enclave when running the OSC	Integrity of the REPORT from the enclave
Quote_Hash	*QUOTE* produced by a Quoting Enclave	Integrity of the QUOTE from the enclave
OSC_Hash	*OSC* Software Code in the Trusted Code Repository	Integrity of the OSC code itself

Table 3.3.: Datasets.

Datasets		
Wiki	Vertices	7,115
	Edges	100,762
Facebook	Vertices	4,039
	Edges	88,234
Synthetic	Vertices	1,000
	Edges	88,234
Synthetic100K	Vertices	100,000
	Edges	1,000,000

4. TRUENODB: THE SCALABLE GRAPH DATASTORE/COMPUTATIONAL ENGINE HYBRID

Handling large and complex dynamic graph datasets poses significant computational challenges. Existing graph databases such as Neo4j, Titan, OrientDB, and others, offer limited functionality making the management of large networks a challenge, from points of view of analyses and presentation.

In this paper, we present TruenoDB an easy-to-use scalable graph database and computation engine. TruenoDB is a novel integration of highly optimized algorithms and implementations, with distributed search engines, graph-parallel computations on top of a dataflow framework, and a rich set of drivers. TruenoDB provides a user-friendly Web UI, a simple API for developing plugins, has extensive language support, interfaces with commonly used execution engines such as Spark, and includes a library of graph analytics kernels.

We validate TruenoDB outstanding usability utilizing a variety of applications ranging from computational systems biology to information retrieval. Finally, we support TruenoDB's practicality through some micro and macro benchmarks that demonstrate an excellent performance, scalability, and flexibility.

4.1 Introduction

Recently, there is an increasing importance towards analyzing graph relationships and interactions of modern datasets. In particular, integrated interaction data (e.g., Protein-Protein interactions, Protein-DNS interactions, microRNA-mRNA interactions, social networks, and others) can have millions of edges and vertices. Moreover, networks[1] can represent dynamic systems in which the graph structure is continuously updated. Analysis of massive graph datasets is often performed via specialized

[1]For the rest of the paper, the terms graph and network are interchangeable.

systems incurring in bulk downloads[2]and in-house processing. Running stages of the analytics pipeline in separate systems is cumbersome and inefficient. For instance, Neo4j [1] and Titan [71] offer graph queries via pattern matching (Cypher [72] and Apache TinkerPop Gremlin [73]), but allow limited support for graph algorithms (e.g., connected components, PageRank). Visualization tools such as Cytoscape [74], NetworkX [61], and Gephi [75] provide methods to visualize small in-memory networks. Nonetheless, these visualization tools are not suitable means to explore a large dataset or perform distributed computation. Further, Biological network databases usually offer a constrained query interface.

Most existing workflows are based on an ETL (i.e., Extract, Transform, Load) pipeline. We can summarize a typical integrated analytics process with the following steps: (1) Individually install all the components needed (e.g., graph database, Apache Spark, Hadoop, distributed indexing system or search engine). (2) Extract and transform raw data into a graph representation. (3) Apply functions to the graph representation and write to a persistent storage. (4) Load the graph into a graph processing system such as GraphLab [76], compute a graph algorithm (e.g., PageRank), and persist results. (5) Load results into dataflow processing engine (e.g., Apache Spark, Hadoop), apply transformations, and persist results.

We present TruenoDB [3], an easy-to-use scalable graph datastore and compute engine. TruenoDB integrates a scalable distributed search engine (i.e., Elasticsearch [59]), graph-parallel computation (i.e., Spark GraphX [60]), and a rich set of drivers (e.g., JavaScript, Java, Scala, C++, and Python). TruenoDB implements a schema-less graph storage, where vertices, edges, and properties are stored as Apache Lucene objects. TruenoDB integrates query capabilities via Gremlin traversal graph language, in-memory single host compute engine utilizing NetworkX library [61], and network visualization through Sigma.js [62]. To support a complete graph analytics pipeline, TruenoDB includes a library of graph analytics kernels, and provides a con-

[2]Bulk download refers to moving/copying significant amounts of data from specialized systems. For instance, when copying data from Neo4j into HDFS to run graph analysis using *mazerunner*. Then, the results are written back to Neo4j in large amounts.

nector (e.g., *trueno-elasticsearch-spark-connector*) to extend graph analysis from Spark GraphX. To enhance the distributed read scalability, TruenoDB implements the Elasticsearch *Sliced Scroll API*.

We validate TruenoDB outstanding functionality through a variety of applications ranging from computational systems biology to information retrieval. Finally, we include a number of micro and macro benchmarks to demonstrate the excellent performance, scalability, and flexibility of TruenoDB.

In summary, our contributions are:

- We alleviate the difficulty of current graph analytics pipelines when handling enormous datasets. Through a novel consolidation of scalable indexed storage, graph-parallel computation, and a diverse set of drivers; TruenoDB offers a straightforward highly scalable platform to process distributed graph analytics and queries.

- TruenoDB implements an efficient *trueno-elasticsearch-spark-connector* which interacts with execution engines such as Apache Spark (e.g., GraphX, GraphFrames, MLlib, Spark SQL). This connector provides an uncomplicated API for developing plugins (i.e., Scala based).

- We offer an *open source* implementation and evaluation of TruenoDB[3].

The outline of this paper is as follows. After presenting TruenoDB architecture in §4.2, we discuss two scenarios that shows flexibility and potential of TruenoDB in §4.3. Then, in §4.4 we provide a comprehensive evaluation of the graph database, including micro and macro benchmarks. Finally, in §4.5, we discuss some related research and existing solutions and include our conclusions in §4.6.

[3]https://github.com/TruenoDB

4.2 System Architecture

TruenoDB implements a fast scalable graph store supported by Apache Lucene indexes, and with distributed computation capabilities. Fig. 4.1 depicts the system architecture.

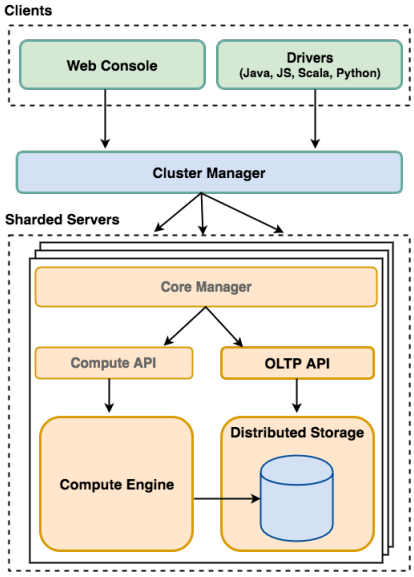


Fig. 4.1.: TruenoDB System Architecture

4.2.1 Cluster Manager

TruenoDB provides a distributed, resilient, and highly optimized graph store. TruenoDB can be easily configured as a cluster or single instance installation. In a cluster setup, TruenoDB deploys a node manager which acts as coordinator. The node manager keeps track of all the servers that are part of the deployment, balance the workload among the cluster, and restarts nodes in case of failures. Further, when a node fails, the graph data is synchronized from the storage master node.

The nodes can be configured either for storage, computation or both. The storage is replicated and sharded on Elasticsearch, while the computation is distributed using

Apache Spark. All storage nodes form an Elasticsearch cluster; likewise, computation nodes form an Apache Spark cluster.

4.2.2 Core Manager

TruenoDB exposes two different methods of communication for clients to interact with the graph store: (1) using the RESTful interface from the Web Console; (2) through any of the available drivers. Either method has access to the same features. The Core Manager handles and resolves clients request using the corresponding API.

In the case of traversal queries or graph updates, requests are handled using the OLTP[4] API, which interacts with the distributed graph store. TruenoDB integrates the Gremlin language to support traversal queries. By contrast, distributed graph analytics are handled by the Compute Engine through the Compute API, as described in §4.2.5.

4.2.3 Web Console

TruenoDB's Web User Interface (Web UI) offers capabilities for graph processing, analytics, visualization and graph management. The Web UI is connected to the database and processing engine using a RESTful interface, which enables a fully interactive workspace that allows graph queries or structure updates via the Gremlin traversal language. Query results are presented in a visual canvas powered by Sigma.js framework, which allows the user to graphically interact with the graph (or subgraph). Moreover, the Web UI offers graph analytics capabilities either locally, using the NetworkX library; or distributed analysis, supported by Apache Spark.

[4]Online Transaction Processing

4.2.4 Graph Store

The graph store is supported by Elasticsearch [59], a full-text search and analytical search engine based on Apache Lucene, and serves two purposes. First, it stores the graph data in a distributed and durable platform. The data is distributed in shards, and replicated along the cluster. Second, the graph store allows the analysis of a large amount of data quickly and in near real time. TruenoDB provides a schema-less graph store, where elements (i.e., vertices, edges) and their properties are stored as Apache Lucene objects. Hence, all the data is automatically indexed without the need to deal with any index management.

TruenoDB does not support ACID transactions, which makes the graph database not a suitable solution for transactional applications. Nevertheless, TruenoDB was designed to provide a highly scalable platform to support analytic computation and high-throughput for online query.

4.2.5 Computation Engine

TruenoDB integrates an efficient graph processing system (e.g., to execute graph algorithms *Spark GraphX* [60, 77]) and a RESTful interface for submitting and managing Apache Spark jobs (i.e., *Spark Job Server* [78]). Spark Job Server maintains the Spark Job Context which allows sharing the *Spark Context* and the *Resilient Distributed Datasets* (i.e., RDDs). Furthermore, it provides a clean and secure method of submitting jobs to Spark, avoiding the complex setup requirements of connecting to the Spark Master directly. Figure 4.2 shows the Compute Engine architecture.

TruenoDB's Compute Engine provides the sharing of *RDD*s in a Spark application among multiple Spark Jobs. Hence, the compute engine enables the recurring utilization of shared RDDs for low-latency data access across multiple graph analysis or queries.

GraphX is a graph processing system built on top of Spark to efficiently process iterative graph algorithms [60, 77]. The dataflow operators *join-map-groupby* form the

core of GraphX (e.g., graph computation on top of collections [60,77]). In GraphX the graph is represented as horizontally partitioned collections [60, 77]. GraphX includes a set of graph algorithms to facilitate graph analytics [79]. In addition, TruenoDB's Compute Engine introduces a few more (available from TruenoDB's Web UI see section §4.2.3):

- PageRank
- Personalized PageRank
- Triangle Count
- Shortest Paths
- Label Propagation
- Connected Components
- Strongly Connected Components
- Minimum Spanning Trees
- Global Clustering Coefficient

TruenoDB's Compute Engine can be extended to include other graph algorithms (e.g., shortest paths with weights, merging graphs, etc.)

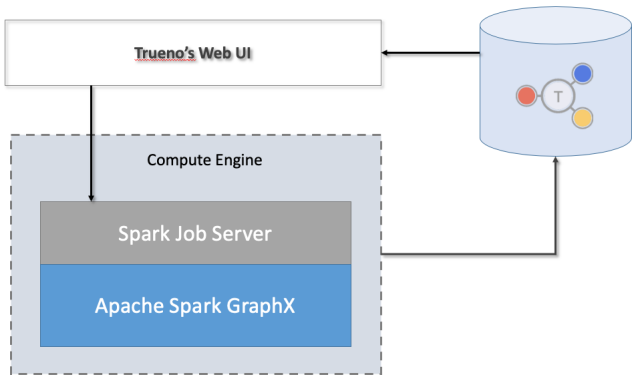


Fig. 4.2.: TruenoDB's compute engine architecture.

TruenoDB supports native integration between Elasticsearch and Apache Spark through *elasticsearch-hadoop*[5]. This connector creates *RDD*s that can read data from *Elasticsearch*. Native integration is the recommended method to retrieve/store documents from/to *Elasticsearch*. Listing 4.1 shows how to connect to the Elasticsearch cluster and retrieve *vertices* and *edges*.

[5]https://github.com/elastic/elasticsearch-hadoop

```
/* Creating Spark Context */
val sc = new SparkContext(conf)
/* Loading Vertices */
val verticesESRDD = sc.esRDD(index+"/v")
/* Loading Edges */
val edgesESRDD = sc.esRDD(index+"/e")
```

Listing 4.1: Connecting to TruenoDB - Scala Source

TruenoDB offers a connector called *trueno-elasticsearch-spark-connector*. TruenoDB implements the Elasticsearch *Scroll API* which can retrieve a large number of results from a single search request. We split the results into partitions using the *Sliced Scroll API* that allows us to improve *read* scalability (as shown in the evaluation §4.4.4). Listing 4.2 shows how to load data from Elasticsearch using partitions and slices.

```
import org.apache.spark._
/* Importing Trueno ES and Spark Connector */
import org.trueno.elasticsearch.spark.connector._

val conf = new SparkConf().setAppName(appName).setMaster(master)
val sc = new SparkContext(conf)
/* Loading Vertices from TruenoDB */
val verticesRDD =
  sc.parallelize(1 to slices, slices)
    .map
    { i =>
      val tc = new ESTransportClient(index,host,port)
      tc.getVertexRDD(i,slices+1)
    }
```

Listing 4.2: Loading vertices from TruenoDB using *trueno-elasticsearch-spark-connector*. Scala Source.

4.2.6 Drivers

TruenoDB provides a rich set of drivers (e.g., Java, JavaScript, Python, C++, and Scala). Drivers connect to the TruenoDB's engine that exposes the graph storage and graph analytics engine. The drivers are completely asynchronous (e.g., *Promises* in Javascript, and *CompletableFuture* in Java API). Moreover, drivers include support for bulk operations.

Creating a Graph: Listing 4.3 shows how to connect to TruenoDB and create a graph g with two vertices and one edge (JavaScript driver). First, it connects to the TruenoDB's engine (line 3-5). Then, it creates a graph object (line 8). Similarly, the driver implements $addVertex()$, $addEdge()$ to create those types of objects (lines 12, 13, and 26). TruenoDB allows customized properties in nodes and edges (lines 15-18). Finally, the drivers persist results into the backend (lines 20 and 29).

```
1  /const Trueno = require('../lib/trueno');
2  /* Instantiate connection */
3  let trueno = new Trueno({host: host, port: port});
4
5  trueno.connect((s)=> {
6    console.log('connected', s.id);
7    /* Create a new Graph */
8    let g = trueno.Graph();
9    /* Set label: Graph Name */
10   g.setLabel('citations');
11   /* Create Vertices */
12   let v1 = g.addVertex();
13   let v2 = g.addVertex();
14   /* Adding properties */
15   v1.setProperty('name', 'alice');
16   v1.setProperty('age', '25');
17   v2.setProperty('name', 'bob');
18   v2.setProperty('age', '35');
19   /* Persist v1, v2 */
20   v1.persist().then((result) => {
```

```
21      console.log('Vertex successfully created');
22    }, (error) => {
23      console.log('Error: ',error);
24    });
25    /* Create an edge between v1 and v2 */
26    let e1 = g.addEdge(1, 2);
27    e1.setProperty('relation', 'knows');
28    /* persist e1 */
29    e1.persist().then((result) => {
30      console.log('Edge successfully created');
31    }, (error) => {
32      console.log('Error: ', error);
33    });
34  }, (s)=> {
35    console.log('disconnected', s.id);
36  });
```

Listing 4.3: Creating Graph - JavaScript Source.

Computing PageRank: Listing 4.4 demonstrates how to connect to the TruenoDB and compute PageRank on top of graph g. TruenoDB's compute engine requests a job to the Spark Master (i.e., we use Spark Job Server RESTful API) First, the JavaScript driver needs to establish a connection (lines 5, 6). Then, we create a graph object in our API (graph g). Next, we define the algorithm (e.g., PageRank) that will be computed on g (line 13). Additionally, the PageRank algorithm requires some parameters such as *tolerance* and *reset probability*. The API allows to persist the result in the backend or to retrieve the values to the client.

```
1  //* Including Libraries */
2  const Trueno = require('../lib/trueno');
3  const Enums = require("../enums");
4  /* Creating new Trueno object */
5  let trueno = new Trueno({ host: host, port: port});
6  trueno.connect((s) => {
7    /* Create a new Graph */
```

```
  let g = trueno.Graph();
  g.setLabel(destinationGraph);
  /* Create Compute Object */
  let computeObject = g.getCompute();
  /* Defining Algorithm Type */
  computeObject.setAlgorithm(Enums.algorithmType.PAGE_RANK);
  /* Algorithms Parameters */
  let parameters = {
    schema: {string: destinationGraph},
    tolerance: {string: 0.000000001},
    alpha: {string: 0.85},
    persisted: {string: "false"},
    persistedTable: {string: "vertices"}
  };
  computeObject.setParameters(parameters);
  /* Get the JobId of the algorithm */
  computeObject.deploy().then((jobId) => {
    console.log('JobId: ', jobId);
    computeObject.jobStatus(jobId)
      .then((status) => {
        if (status == Enums.jobStatus.FINISHED) {
          computeObject.jobResult(jobId)
                       .then((ranks) => {
            console.log('Ranks: ', ranks.result);
          });
        }
      });
  });
}, (s) => {
  console.log('disconnected', s.id);
});
```

Listing 4.4: Computing PageRank - JavaScript Source

4.3 Applications

We provide an open source implementation and evaluation of TruenoDB. Additionally, the project includes an extensive documentation of TruenoDB components[6]. We evaluated TruenoDB usability in a wide range of datasets to demonstrate its flexibility and simple Web UI. We support TruenoDB's practicality via micro and macro benchmarks (Refer to Section §4.4.4).

Top ranked gene in the Parkinson's neighborhood: The analysis and visualization of interaction datasets are increasingly important. Some tools have been developed to provide the biological and biomedical research communities with genetic and protein interaction data (e.g., Biogrid [80,81]). However, to analyze and visualize this data, bulk downloads and processing are necessary[7]. TruenoDB provides a simple filtering and analysis of the full Biogrid dataset. The visualization and analysis are straightforward using TruenoDB Web UI. For larger datasets, the Web UI integrates the distributed compute engine (Apache Spark) via Spark Job Server. TruenoDB Web UI incorporates NetworkX to analyze small networks in memory.

TruenoDB provides a graph import tool[8] to load networks from JSON-formatted files. For instance, it is possible to load the complete Biogrid dataset in a graph representation using:

```
1 /trueno import biogrid
```

Listing 4.5: Importing the Biogrid dataset.

We study the Parkinson neighborhood using two publicly available databases (e.g., Biogrid[9] and OMIM[10]). For instance, we filtered the genes of Parkinson disease (retrieved from OMIM) in the Biogrid dataset (e.g., from the TruenoDB Web UI). Listing 4.6 shows the integrated Gremlin query.

```
1 /g.V().has('name', within('GBA','ADH1C','TBP','ATXN2','MAPT','GLUD2'))
```

[6]https://truenodb.github.io/documentation/latest/
[7]For example using Biogrid and Cytoscape dedicated plugin.
[8]https://github.com/TruenoDB/trueno-javascript-driver
[9]Biogrid is a repository of protein, chemical, and genetic interactions.
[10]https://www.omim.org/

```
2  .as('genes')
3  .both()
4  .as('neighbors')
5  .bothE()
6  .as('network')
7  .select('genes', 'neighbors', 'network')
```

Listing 4.6: Filtering the Parkinson Neighborhood.

TruenoDB can store intermediate results in the graph store (e.g., we can save *parkinson_network* as a graph available in the Web UI). Further, TruenoDB Web UI facilitates loading and analyzing stored networks. For example, we use TruenoDB compute engine[11] to obtain the top ranked genes in the Parkinson disease[12]. Figure 4.3 demonstrates top ranked gene[12] neighborhood.

Fig. 4.3.: TruenoDB Web UI Visualization - Top ranked gene neighborhood in the Parkinson's Network.

High energy physics citation network analysis [82]: We implemented a wrapper (i.e., GraphX algorithms wrapper) integrated with TruenoDB's Web UI. In this application, we analyze the citations SNAP dataset [70][13] (nodes represent papers[14]). First, we built a personalized PageRank algorithm to obtain a rank relative to the "source" node (e.g., V_s) in the graph G. We were able to find the most important paper from the perspective of the source paper (i.e., V_s). We obtained the

[11]We request a job to the Spark cluster via Spark Job Server.
[12]Top ranked gene is MAPT.
[13]https://snap.stanford.edu/data/cit-HepPh.html
[14]If a paper i cites paper j, the graph contains a directed edge from i to j.

top ranked node[15]. We found the most important paper[16] from the *top ranked paper point of view*.

We extended the analysis using the gremlin query language. We traversed the graph and provided the following information.

- We obtained the paper titles and citations count.
- We listed the paper titles of the papers with more than 100 citations.
- We showed the subgraph of Vertex V_1 (we showed the vertices who cite V_1).

Figure 4.4 exhibits the uncomplicated Web UI interface interacting with the top ranked paper neighborhood.

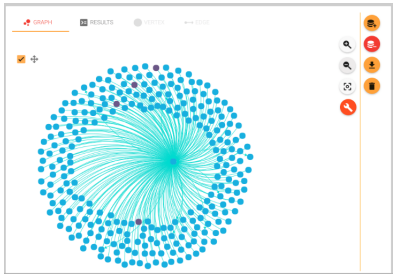


Fig. 4.4.: TruenoDB Web UI Visualization - Top ranked paper neighborhood in the Citation's Network.

4.4 Evaluation

4.4.1 Experimental Setup

Single-machine experiments were run using a machine with Intel Core i5 (4 cores @2.7GHz, 8MiB cache) with 8GiB of RAM.

Distributed experiments were run on a cluster of 16 machines with Intel Xeon X3430 (4 cores @2.4GHz, 8MiB cache), 8GiB of RAM, and spinning disks. The nodes connect over a shared Gigabit Ethernet connection. We configured the nodes

[15]The top ranked paper "Noncompact Symmetries in String Theory" [83]
[16]"An algorithm to generate classical solutions for string effective action" [84]

Table 4.1.: Datasets

Datasets		
Biogrid	Vertices	15,034
	Edges	301,685
Citations	Vertices	29,554
	Edges	167,103
LDBC Graphalytics (Scale 1)	Vertices	34,379
	Edges	1,010,631
LDBC SNB (Scale 1, filtered by **knows** relationship)	Vertices	9,152
	Edges	180,832
Pokec	Vertices	1,632,803
	Edges	30,622,564

with CentOS Linux release 7.2.1511 (Core), Spark 2.1.1 built for Hadoop 2.7.3, and Elasticsearch 5.3.2.

4.4.2 Query and Traversal Performance

First, we evaluated TruenoDB performance on a single machine installation. The performance was compared against a well known graph database, Neo4j, using three different datasets (as shown on table 4.1): Biogrid [81], Citations [70], and Pokec [70]. Both, Trueno and Neo4j v3.2 were installed on identical hardware, and tests were conducted using the Javascript driver. To evaluate the performance we measure the throughput of each graph database while handling a workload between 5K to 50K requests, depending of the dataset and the test. The workload consists of reads of randomly selected vertices, write operations (creates vertices not using transactions), and a mix of reads and writes (90% of the workload correspond to read operations).

Throughput: Table 4.2 shows the throughput of Trueno compared to Neo4j. The read performance between both databases is similar on the smaller datasets, but as the data grows TruenoDB scales better than Neo4j, which outperforms by a 1.2× factor. Furthermore, TruenoDB outperforms Neo4j on write operations by two orders of magnitude.

Although we are not declaring a transaction explicitly on Neo4j, every update to the graph store is wrapped on a transaction. Therefore, the main overhead on writes operations correspond to the transaction management. By contrast, TruenoDB by not supporting ACID transactions offers a higher throughput.

Both Neo4j and TruenoDB use Apache Lucene to index the data. However, in the case of Neo4j only the unique keys are indexed. As discussed in §4.2.4, TruenoDB by being supported on Elasticsearch, all data is indexed.

Table 4.2.: Throughput Statistics

Datasets		Requests (records/s)	Trueno (records/s)	Neo4j (records/s)
Biogrid	Read	15,034	4,450.58	4,629.14
	Write	5,000	4,752.48	56.46
	Mix	15,034	3,442.05	2,566.54
Citations	Read	29,554	4,336.03	5,187.62
	Write	5,000	4,697.75	30.94
	Mix	29,554	4,279.64	3,217.22
Pokec	Read	50,000	3,243.29	2,619.41
	Write	5,000	6,853.29	30.76
	Mix	50,000	2,841.42	1,228.29

4.4.3 Scalability

In this section, we demonstrate that, for distributed reads, TruenoDB scales according to the number of executors, partitions, and slices of data (Figure 4.6). Going from 4 to 32 partitions (a factor of 8) yields an approximately 4-fold increase in throughput (records/second). We utilize the *trueno-elasticsearch-spark-connector* to take advantage of the Sliced Scroll API provided by Elasticsearch (Section §4.2.5). In this experiment the *Spark master* is set to *Cluster Master* and we increase the number of executors, partitions, and slices by a factor of 4. All *executors* have 3GB of memory and 2 cores.

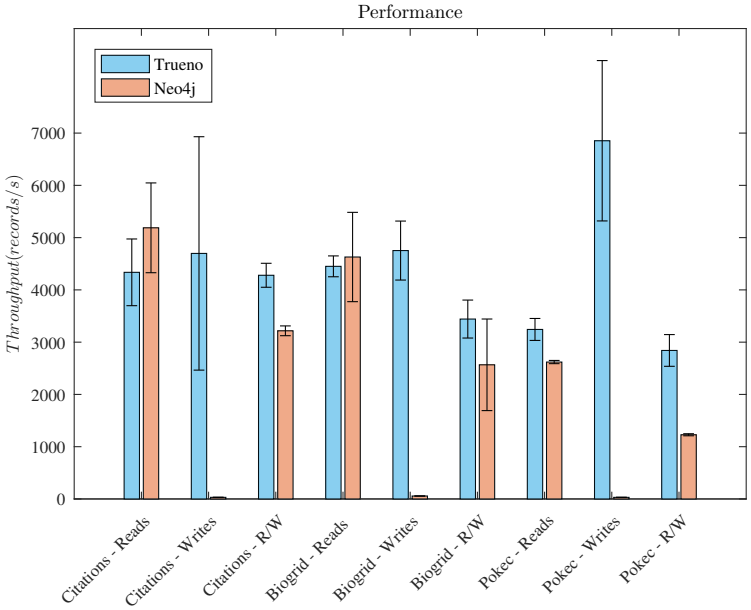


Fig. 4.5.: Performance

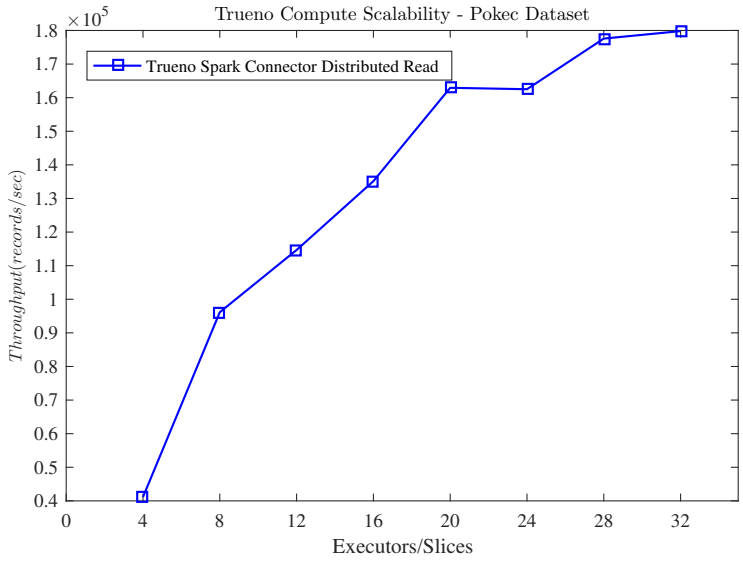


Fig. 4.6.: TruenoDB distributed read on the Pokec dataset.

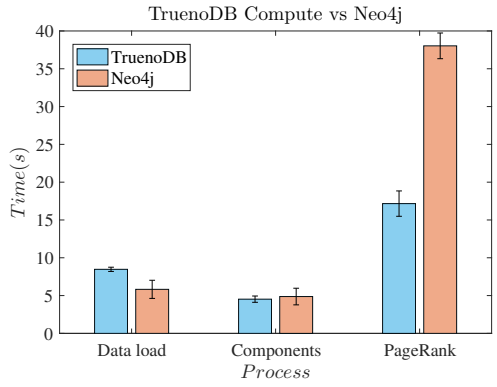
4.4.4 Computation Engine

In these experiments, one machine is reserved as the *master* while the others are *workers*. We evaluated TruenoDB's Compute Engine with the same datasets used for the §4.4.2 (see table 4.1). For each dataset, we measure the total read time from the TruenoDB's backend. In addition, we create the *VertexRDD* and *EdgeRDD*. Then, we analyze the GraphX graph creation time from a collection of vertices and edges in an RDD (e.g *VertexRDD* and *EdgeRDD*). Finally, we estimate the runtime of two graph algorithms (e.g., PageRank and Connected Components). PageRank runs 10 iterations as in [60].

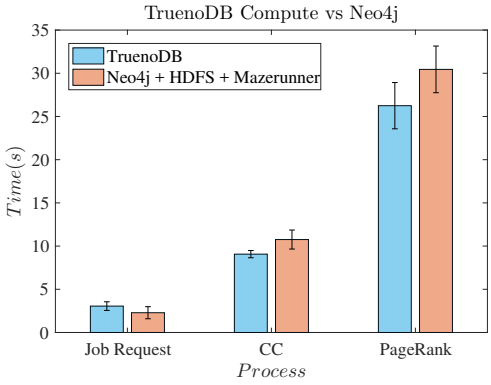
We begin by studying the compute engine behavior on a single machine. TruenoDB and Neo4j implement a *Spark connector*[17] which uses GraphX as the building block. Figure 4.7(a) shows the comparison between TruenoDB Compute Engine workflow and Neo4j (e.g., Neo4j Spark connector). TruenoDB performs similarly to Neo4j when loading data and computing the *Connected Components* algorithm. However, the *PageRank* algorithm is notable faster in the TruenoDB engine. TruenoDB creates the VertexRDD and EdgeRDD using the documents retrieved from TruenoDB's backend (i.e., Elasticsearch).

We evaluate the Neo4j Mazerunner[18] Service which adds a REST API that allows requesting jobs to Apache Spark GraphX (Figure 4.7(b)). For this experiment, we install all the components in a virtual machine[19]. TruenoDB excels in the easy installation and configuration compared to the setup of three Docker containers required by Mazerunner. Neo4j is slightly faster when requesting a job to the Spark cluster; however, algorithms run faster when using TruenoDB (i.e., Mazerunner needs to copy the results back to HDFS then Neo4j has to copy those into the GraphDB engine). Not to mention, Neo4j Docker container only supports the version 2.2.1.

[17] https://github.com/neo4j-contrib/neo4j-spark-connector
[18] https://github.com/neo4j-contrib/neo4j-mazerunner
[19] We setup a virtual machine (VMWare) with an Intel Core i7-4700MQ (4 cores @2.4GHz, 6MiB cache) 4GiB of RAM and a spinning disk. Algorithms run until convergence and the runtime includes loading the data from TruenoDB and Neo4j. The virtual machine includes the three docker image deployments (i.e., Hadoop HDFS, Neo4j Graph Database, and Apache Spark.)



(a) TruenoDB vs Neo4j.



(b) TruenoDB vs Mazerunner.

Fig. 4.7.: Computation time (PageRank and Connected Components algorithms) on the Biogrid dataset. TruenoDB vs Neo4j.

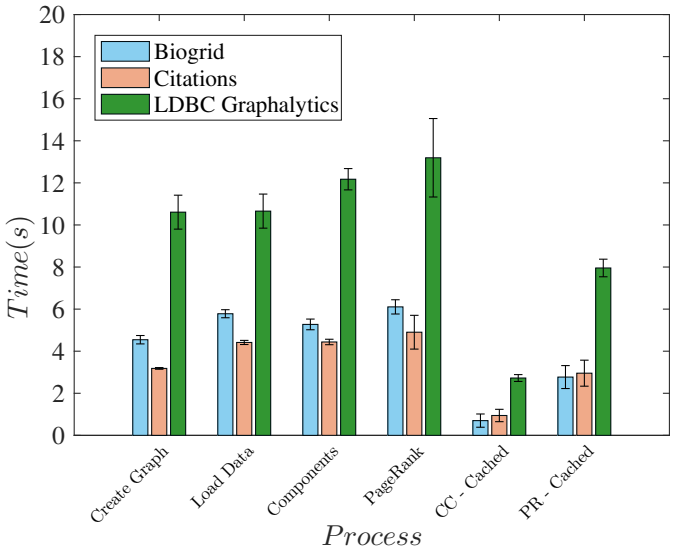


Fig. 4.8.: TruenoDB Compute on Biogrid, Citations, and LDBC (Scale 1) datasets.

In Figure 4.8 we show the computation time on the *Citations*, *Biogrid*, and LDBC Graphalytics[20] datasets. We also include the runtime when all data structures are cached in memory (i.e., the algorithms finish/converge faster). In this test the *Spark master* is set to *local[*]* and we use two *executors* with 3GB of memory and 2 cores.

Finally, we evaluate TruenoDB against a bigger dataset (e.g., *Pokec*). This experiment utilizes 34 *executors* with 3 GB of RAM and 2 cores each. In Table 4.3 and Figure 4.9 we observe that TruenoDB distributed read[21] took 271.47 seconds to retrieve 32 million records from the backend (i.e., 118K records/second). The *PageRank* computation took roughly 10.8 minutes. Similarly, the *Connected Components* runtime was 4.92 minutes. Therefore, we exploit the strong scaling performance guarantees provided by GraphX [60].

[20]https://github.com/ldbc/ldbc_graphalytics
[21]We use *elasticsearch-hadoop* due to its maturity.

Table 4.3.: Compute Statistics

Compute Statistics (s)		
Biogrid	Load data	5.781
	Graph creation	4.546
	Connected Components	5.273
	PageRank	14.981
	CC - cached	0.701
	PageRank - cached	6.770
Citations	Load data	4.416
	Graph creation	3.182
	Connected Components	4.438
	PageRank	11.951
	CC - cached	0.944
	PageRank - cached	6.955
Pokec	Load data	271.472
	Graph creation	270.119
	Connected Components	295.918
	PageRank	651.907
	CC - cached	37.698
	PageRank - cached	350.487

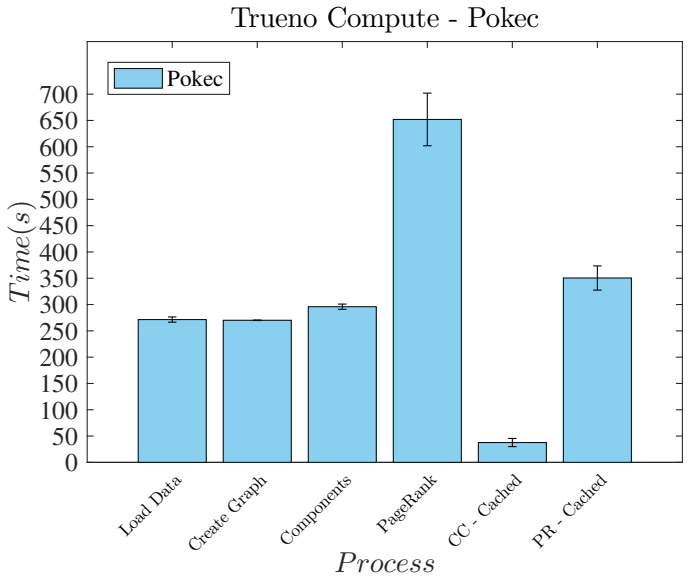


Fig. 4.9.: Distributed read and computation on the Pokec dataset.

There is an extensive comparison between graph analysis platforms in [85]. They introduce LDBC Graphalytics which consists of six deterministic algorithms and standard datasets that enable an objective comparison of graph analysis platforms.

4.5 Related Work

4.5.1 Graph Databases

Titan [71] and Neo4j [1,72] have a limited support for graph algorithms (e.g., connected components, PageRank). These graph databases offer mostly graph queries via pattern matching (i.e., Cypher [72] and Gremlin [73]). TruenoDB utilizes the Spark GraphX capabilities to provide additional graph processing features (e.g., constructing the graph g from *Elasticsearch* documents and running graph algorithms on top of g).

TAO is an eventual consistent geographically distributed graph store optimized for reads [86]. TruenoDB relies on the eventual consistent model of Elasticsearch. For

instance, write operations wait for the primary shards to be active before proceeding (i.e., $wait\_for\_active\_shards = 1$). This default value can be modified by setting $wait\_for\_active\_shards$. TAO was designed specifically to serve the Facebook's social graph [86]. TAO does not provide an advanced graph processing API (i.e., analysis jobs do not execute within TAO requiring off-line graph processing systems). On the contrary, TruenoDB integrates *Spark Job Server* and *Spark GraphX* to provide an advanced customizable API.

In their promising paper, Dave et al. (2016) discussed GraphFrames as a single API that includes graph algorithms, pattern matching, and relational queries [87]. They built *GraphFrames* from previous work in graph analytics using query optimization for pattern matching and declarative APIs for data analytics [87]. Previously, the implementation of those applications required multiple systems increasing the complexity and overhead. GraphFrames promises to write a complete optimized processing pipeline [87]. TruenoDB aims to provide a user-friendly (i.e., easy to use) scalable graph datastore and compute engine to solve the common analytics pipeline of current graph databases. TruenoDB integrates GraphX and Spark Job Server to include graph analytics, a traversal API (e.g., Gremlin [73]) and visualizes the result through a Web UI.

4.5.2 Visualization and Analysis Tools

With the aim of understanding complex graph interactions, graphic representations facilitate and expedite the analysis of data. For instance, nodes are more comfortable or easier to identify when they have a higher number of interactions, or a higher ranked node (e.g., a common protein, person, paper that interacts with many others). Consequently, many tools have been developed to allow the visualization of network data, Cytoscape [74], NetworkX [61], Gephi [75], Sigma [62]. Cytoscape is an open-source platform for analyzing and visualizing molecular interaction networks [74]. Cytoscape does not provide tools to explore a large dataset. Although

Cytoscape offers an interactive visualization tool (i.e., Cytoscape.js), Cytoscape focuses on desktop-based use cases. In [88], they introduced PINV (Protein Interaction Network Visualizer) that applies prefilters on the data to visualize and analyze protein interactions. HTML5, JavaScript, SVG (Scalable Vector Graphics), and canvas are the building blocks of PINV. On the other hand, TruenoDB integrates Sigma [62] and NetworkX [61]. Sigma is a JavaScript library dedicated to graph drawing. Sigma is a highly customizable rendering engine (e.g., renders on WebGL or Canvas). Moreover, TruenoDB includes NetworkX as a single host in-memory compute engine. NetworkX is a Python language software package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks [61].

4.5.3 Graph Processing Frameworks

We briefly describe graph processing frameworks similar to GraphX [60]. For instance, Trinity [89] is a graph engine built on top of distributed memory storage infrastructure (i.e., memory cloud). PEGASUS [90] is a graph mining package built on top of Hadoop. Apache Giraph [4] is an open-source project that utilizes an iterative vertex-centric programming model similar to Google's Preguel [91]. Giraph is built on top of Apache Hadoop's MapReduce. To overcome the drawbacks of MapReduce approaches to process large networks, many solutions based on vertex-centric computation arose (e.g., Pregel [91], GraphLab [76], PowerGraph [89], and GraphChi [92]). PowerGraph [89] utilizes a programming model known as Gather-Apply-Scatter (GAS). Preguel programming model utilizes message passing between vertices in a graph [91]. GraphLINQ is a graph processing framework using a general-purpose dataflow framework [60].

4.6 Conclusion

We presented TruenoDB a user-friendly (i.e., easy to use) distributed and scalable graph datastore and computational engine. We demonstrated TruenoDB flex-

ibility and usability through a variety of applications. TruenoDB provided facile graph analysis and visualization via the Web UI. The integrated distributed graph computation (i.e., Spark GraphX) is extremely useful when handling large datasets. We validated TruenoDB excellent performance providing evidence through comparisons against state-of-the-art commercial graph database and a variety of datasets (e.g., biology datasets, social networks, citations networks, and the LDBC benchmark). In particular, TruenoDB outperforms Neo4j on write operations by two orders of magnitude. We demonstrated the desirable TruenoDB scalability guarantees using the simple cluster installation and large datasets. We showed that *trueno-elasticsearch-spark-connector* provides outstanding distributed read performance. TruenoDB and Neo4j provide Spark connectors that allow a practical interaction with GraphX and GraphFrames. TruenoDB aims to provide a straightforward highly scalable platform to process distributed graph analytics and queries. TruenoDB uncomplicated installation and accessible Web UI make this engine a suitable candidate for a diverse set of applications ranging from computational science biology to information retrieval.

4.7 Future Work

We have a long roadmap of features that will be part of forthcoming releases of TruenoDB. First, we plan to include optimizations and improvements in the drivers. Then, we will include additional algorithms in the integrated compute engine (e.g., Machine Learning algorithms using MLlib, Merging Graphs, Graphframes based algorithms). We plan to improve the traversal API. Next, the automated Spark Cluster installation and configuration will incorporate options for cluster management tools (e.g., Mesos). TruenoDB will include integrated authentication and access control mechanisms. Additionally, upcoming releases include support for dynamic graphs, compressed querying, a full versioning API and storage optimization, and support for trusted computations on graphs.

Acknowledgments

This work was partially supported by the National Science Foundation Grant number CCF 1533795, Division of Computing and Communication Foundations under the XPS program. The list of authors of TruenoDB includes Victor Santos, Servio Palacios, Edgardo Barsallo, Miguel Rivera, Chih-Hao Fang, Pen Hao, Ananth Grama, Tyler Cowman, and Mehmet Koyuturk. TruenoDB's project started in Fall 2015; experimental results were obtained in Fall 2016. The principal investigator (PI) and Victor Santos granted permission to use TruenoDB's write-up for this dissertation.

5. MIOSTREAM: AN INTEGRITY-PRESERVING, PEER-TO-PEER, DISTRIBUTED LIVE MEDIA STREAMING ON THE EDGE

The work in this chapter has been previously published in Multimedia Tools and Applications International Journal. DOI: https://doi.org/10.1007/s11042-018-6940-2

The typical centralized cloud model is poorly suited to latency-sensitive applications requiring low-latency and high-throughput. This paper proposes an integrity-preserving serverless framework for live-video streaming that runs on the edge of the network. We present the design, implementation, and evaluation of a novel P2P service based on WebRTC (web browsers with Real-Time Communications) called MioStream. MioStream is an open-source alternative for distributed media streaming that runs on the edge of the network without incurring in costly and extensive CDN infrastructure. We contribute a unique mix of algorithms using WebRTC data channels. For instance, under network degradation and high-churn environments, MioStream restructures the topology dynamically. MioStream provides authentication, privacy, and integrity of video chunks.

This paper exposes a set of micro-benchmarks to measure the quality of service under network degradation and high churn environment (inducing failures). The Mesh topology offers the highest goodput per peer; the stalled playback on a node equals 1.8% of the total video play. Our results show the feasibility of this proof of concept under high-churn environments. The total stream interruptions in the topology are not longer than one second under a binomial distributed series of failures. The integrity check applied to each package includes a considerable overhead and impact the quality of service.

5.1 Overview

Peer-to-Peer (P2P) systems constitute the backbone of a diversity of distributed implementations (e.g., video streaming, file sharing, and many others) due to their performance, resiliency, and scalability. In particular, we are interested in an emerging technology called WebRTC that allows Real-Time Communication (RTC) on top of the World Wide Web (i.e., through web browsers such as Chrome, Firefox, Opera, and Edge). In their paper, Rhinow et al. (2014) provided an analysis of the feasibility of implementing live video streaming into web applications [93]. Although they offered performance measures, they did not include an open-source repository, nor they studied the security implications and quality of service under failures. Lopez et al. (2016) introduced Kurento Media Server, an open-source WebRTC media server that offers features such as group communication, recording, routing, transcoding, and mixing [94]. They provided APIs that facilitate the development of web-based video applications. Lopez et al. did not present an analysis of the quality of service under failures or security attacks—i.e., an adversarial environment. Similarly, Garcia et al. (2017) introduced NUBOMEDIA, an open-source cloud platform-as-a-service (PaaS) designed for WebRTC services [95]. NUBOMEDIA exposes a set of APIs that facilitate the development of WebRTC applications.

In this paper, we propose *MioStream* [96] a new distributed peer-to-peer video streaming system. MioStream takes advantage of emerging technologies to accomplish the desired operability. MioStream offers media streaming capabilities on top of WebSockets, WebRTC, and JavaScript. We identified challenges analogous to the ones exposed in [97, 98]. For instance, how to deal with peer selection in high churn and heterogeneous network. MioStream offers authentication and integrity capabilities. For instance, We authenticate every peer against a centralized server, supervisor, using an authentication protocol based on TLS and DTLS cookies. Once authenticated, peers are monitored, including their connection channels using WebSockets heartbeats and WebRTC data channel features. In the case of network degradation,

the system is capable of restructuring the topology dynamically (i.e., changing active receivers or bridges) to obtain the performance in the overall network similar to [97, 98]. MioStream is an open-source alternative for distributed media streaming that runs at the edge of the network without incurring costly and extensive CDN infrastructure.

In summary, our contributions are:

- A personalized object-lookup utilizing WebSockets and WebRTC.
- A novel communication and security layer implemented through the *supervisor* as Certification Authority (CA).
- An open source implementation and evaluation of the proposed techniques [99].
- An analysis of the quality of service in the presence of failures and attacks.
- We enhanced previous work on integrity validation of video chunks [100]. We contribute a unique method applied to live-video streaming.

Through a set of micro benchmarks, our evaluation demonstrates MioStream high-quality video streaming in the presence of failures.

A brief explanation of the building blocks of the system is in Section 5.2. The key concepts and architecture of the system are in Section 5.3. We provide an extensive set of experiments demonstrating the feasibility of our system in Section 5.4. Section 5.5 provides an insight into challenges experienced throughout the development of the system and future work. We include a discussion of related research and existing solutions (Section 5.6). Finally, we present our conclusions in Section 5.7.

5.2 Background

In this section, we introduce the essential concepts of the WebRTC stack utilized to build MioStream. Our discussion is based on [101–105][1].

[1]We refer the reader to this useful books. You will find a detailed explanation of WebSockets and WebRTC.

5.2.1 WebSockets

WebSockets allow a client and a server exchange message-oriented streams of text/binary data in a bidirectional way [104]. According to [104], the API provides the following services:

- WebSockets exposes a message-oriented communication API and allows useful message framing.
- WebSockets comprises an extensible API and sub-protocol negotiation.
- WebSockets enforce the same-origin policy and connection negotiation.
- WebSockets can interoperate with current HTTP infrastructure.

MioStream utilizes SocketCluster framework [106] to provide WebSocket services. SocketCluster allows communitation using the client-server model (similar to *socket.io*) and group communication via *pub/sub* channels. Also, this library exposes features such as automatic reconnects, heartbeats, timeouts, and multi-transport fall-back functionality as recommended in [104, 105]. SocketCluster design scales vertically—across multiple CPU cores—and horizontally—across multiple nodes/instances/machines.

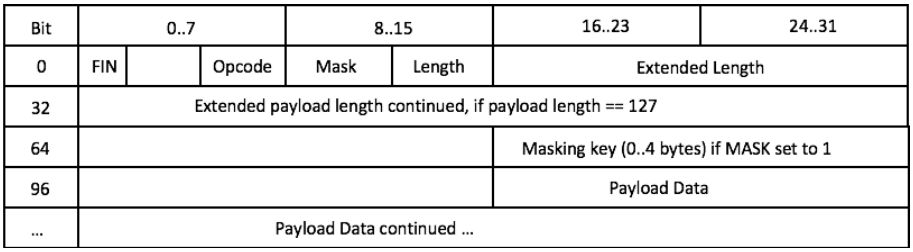


Fig. 5.1.: WebSocket frame [104, 107].

WebSocket frame

Figure 5.1 shows a map of a typical WebSocket frame for the RFC6455 [107]. In detail, the first bit FIN indicates if the fragment is the last or final fragment of

a message. To indicate the type of frame that is being transferred, the Websockets frame includes the 4-bit *opcode* to represent (1) text, (2) binary, or (10) for connection liveness. For more detail please check [104].

Deploying WebSockets Infrastructure

Since we do not have control over the policy of the client C_i network, MioStream has to use TLS tunneling over a secure end-to-end connection. Therefore, WebSockets traffic can circumvent firewalls and intermediate proxies [104]. It is possible to send security parameters to *SocketCluster* to allow secure connections.

The following discussion is motivated by [104, 105, 108].

Performance objectives

- MioStream leverages reliable deployments through secure WebSocket (WSS over TLS).
- To minimize transfer size, MioStream optimizes the binary payloads.
- To minimize transfer size, MioStream considers compressing UTF-8 content.
- MioStream avoids *head-of-line blocking* splitting long messages.
- MioStream observes buffered data on the client.

Performance notes

MioStream takes into consideration the performance criteria based on WebSockets' architecture [104]. For instance, MioStream considers in-order delivery of WebSockets messages associated with the order of the client's queue. According to [104], large messages or queued messages will delay the delivery of messages queued behind it (e.g., this is known as *head-of-line blocking*). MioStream's application layer can divide large

messages into shorter pieces or implement its customized priority queue [104]. As a result, considering that the application is delivering latency-sensitive data, MioStream takes care of each message's payload size.

5.2.2 WebRTC

Web Real-Time Communication (WebRTC) is a mixture of JavaScript APIs, standards, and protocols, which enables peer-to-peer data, video, and audio sharing between browsers (peers) [101, 102, 105, 109]. WebRTC transports data over UDP because latency and timeliness are crucial [101, 105].

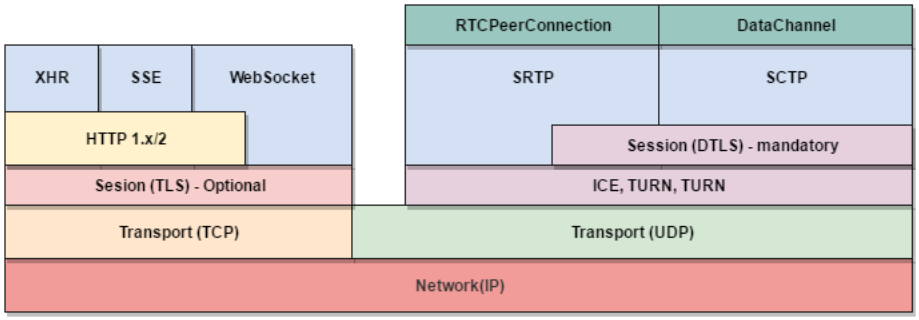


Fig. 5.2.: WebRTC Protocol Stack [101, 104].

WebRTC protocol stack components

Figure 5.2 shows the protocol stack [101, 104].

- ICE: Interactive Connectivity Establishment [110]
- STUN: Session Traversal Utilities for NAT [111].
- TURN: Traversal Using Relays around NAT [112].
- SDP: Session Description Protocol [113].
- DTLS: Datagram Transport Layer Security [114]

- SCTP: Stream Control Transport Protocol [115]
- SRTP: Secure Real-Time Transport Protocol [116]

To establish and maintain a peer-to-peer connection over UDP, ICE, STUN, and TURN are required. MioStream includes secure data transfers between peers through WebRTC; therefore, DTLS is used to encrypt data transfers [104, 105]. SCTP and SRTP expose multiplexing of different streams. Also, SCPT and SRTP provide partially-reliable delivery on top of UDP and congestion control.

RTCPeerConnection API

RTCPeerConnection RTCPeerconnection wraps the connection setup, management, and state [104].

DataChannel

The peers can exchange arbitrary application data through the data channel API [104]. MioStream utilizes the data channel API to provide edge creation in the personalized object lookup. According to [104], each data channel can provide the following:

- Out-of-order or in-order delivery of messages.
- Partially reliable or reliable delivery of messages.

When setting a time limit or a maximum number of retransmissions the channel can be configured to be partially reliable [104]. Also, the WebRTC stack will handle the acknowledgments and timeouts [104].

MioStream deals with the NAT traversal problem using WebRTC capabilities. Also, MioStream wrapped the PeerJS [117] library to provide WebRTC basic functionality and functions (e.g., Communication Layer). For instance, the built-in ICE [110] protocol performs the required routing and connectivity checks. MioStream's unique

communication and security layer handle the delivery of notifications (*signaling*) and *initial session management*. MioStream integrates the PeerJS library with the communication and security layer.

Session Description Protocol [113]

WebRTC utilizes the Session Description Protocol (SDP) to determine the parameters of the peer-to-peer connection. SDP describes the session profile; that is, SDP does not deliver media directly. Also, the Session Description Protocol describes the properties of a session through an uncomplicated text-based protocol. Finally, the JavaScript Session Establishment Protocol [118] (JSEP) encapsulates the Session Description Protocol (SDP); therefore, WebRTC applications do not have to deal with SDP directly.

Interactive Connection Establishment (ICE) [110]

When establishing a peer-to-peer connection, the peers need to route packets to each other. In addition, the peer-to-peer connection can fail because the peers can be in different private networks. To combat this, ICE agents manage this connectivity complexity. As shown in Figure 5.2, each **RTCPeerConnection** object contains an *ICE agent*. The ICE agent collects the candidates' port tuples and local IP addresses. Also, the agent transmits connection keep-alive. Finally, the ICE agent makes connectivity checks among peers.

5.3 Design and Implementation

5.3.1 Architecture

MioStream is composed of daemons installed with just one command via the NPM (Node Package Manager [119]). The system has a peer-to-peer architecture

which comprises three main components, as shown in Figure 5.3. We describe the components as follows.

- **Broadcaster**: The *broadcaster* is the source of a particular media stream. This is a multiplatform [120] application which leverages cutting edge web technologies such as WebRTC, Media Source Extensions [121], and all experimental components included in the Chromium [122] browser. With all Node.js native capabilities, the *Broadcaster* extracts binary data either from a camera/microphone hardware or a pre-encoded WebM [123] container, which can be audio or video. Finally, the *Broadcaster* streams the source media via WebRTC data channels (See §5.2) to the neighbors (e.g., one-hop apart connected peers).

- **Receiver**: The *receiver* is an *HTML5* web application which receives the desired media stream. Using several technologies present in the *Broadcaster* (WebRTC and Media Source Extensions), the *receiver* consumes a stream and **relays** the stream to its neighborhood (neighbors peers).

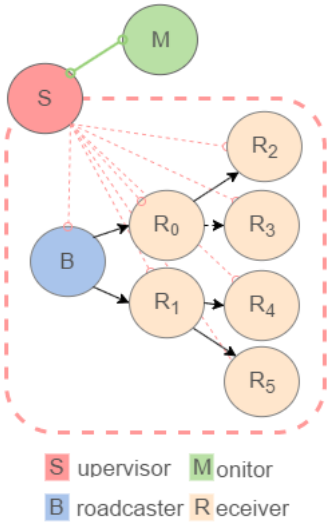


Fig. 5.3.: MioStream Architecture Diagram.

- **Supervisor**: The *supervisor* registers each peer (*broadcaster* or *receiver*). The *supervisor* monitors every connection to the components in real-time and notifies every relevant change to the corresponding nodes, i.e., *disconnection of a*

peer. The supervisor includes the authentication protocol explained in §5.3.4. That is, we use the *supervisor* as a CA (Certification Authority) to take care of the authentication of every peer facing challenges similar to [124]. The system enhances the privacy and integrity of the video chunks allowing only authenticated peers to transmit/receive data and connect to streams. We assume that the peers have a valid public key that the server can validate in the authentication protocol. Also, the **authenticated** *broadcaster* generates keyed SHA-256 (HMAC) to ensure the integrity of video chunks. Therefore, the system allows the detection of malicious chunks in the broadcast.

After a *Broadcaster* creates a stream, the *supervisor* registers and announces the existence of this new stream to the *Receivers*. When a *receiver* chooses to join the stream, the *supervisor* registers the new component in the stream topology. After the insertion of a new *receiver* in the topology, signaling events and connection setup connect this new peer to its neighbors directly. This new data channel creates a path where the stream will flow. Currently, we have implemented three topologies: **Linked List**, where viewers are chained one after another in the order of arrival. **Binary Tree**, where viewers connect into a tree structure. Also, we implemented a **Mesh** topology. The tree and mesh topologies have the advantage of the scalability and low content delivery latency. The stream only takes the *depth* of the tree to reach the last node in the topology, and the scalability of the overall topology follows the geometric series: $2^{n+1} - 1$ where n is the depth of the tree.

The following additional component, not part of the architecture, is used for administrative and experimental purposes.

- **Monitor**: The *monitor* is an *HTML5* web-tooling interface used to visually emit commands to the *receivers* and evaluate the effectiveness of a particular streaming topology. The monitor is conduct experiments on a streaming topology according to the following parameters: network topology for the receivers,

number of induced failures per second, number of simulated attacks per second and probabilistic distribution to generate attacks or failures. Some of the features of the *monitor* include to measure stalled stream events, disconnections, apply different failure distributions (see §5.4), and visualize the shape of the topology (usually a binary tree).

The system relies on WebSockets and WebRTC data channels. We use the SocketCluster [106] library for the real-time signaling communication from which we leverage convenient features such as heartbeats, connection state, and event-driven communication. The *supervisor* is continuously listening for connections from *broadcasters*, and *receivers* (e.g., peers). At each disconnection, the supervisor updates the data structure to balance the load among peers. Also, the supervisor serves as a low-latency fault-tolerant messenger and certification authority.

5.3.2 Virtual Topologies

One of the key aspects of MioStream is the virtual topologies. The main role of the virtual topology is to provide a deterministic way of arranging the connections between nodes (connected peer machines) in the system. When the supervisor component is deployed, it arranges the broadcasters and receivers in the desired topology data structure. In the case of a list, it uses a linked list to store each broadcaster following a receiver which in turn broadcast the packages to the next node. The flow of data in the topology is managed by monitoring delays and retransmissions in each section of the topology. The following is the description of each implemented topology and how they work:

1. **List Topology**: The simplest of the topologies is a linked list virtual overlay where each broadcasting node is connected to a receiver, which in turn is a broadcaster to the following node. Intuitively, the scaling of this topology is linear, and failures can be handled by creating connections to successive nodes. For instance, we can let specify that each node will connect at least to 3 nodes

ahead of its current position, this way if a subsequent node fails, the stream can continue to flow uninterrupted or with little delay. Stalled flows of data due to asymmetric connection speeds between nodes in the list can be mitigated by keeping the topology in descending order using connection speed and latency as the ordering metric.

2. **Mesh Topology**: The mesh topology is represented as a 2D matrix where each neighboring cell is a receiver node which consequently transmits the flow of data to neighboring nodes. This topology offers low latency and scalability by start the streaming the data from several broadcasters across the 2D matrix. These broadcasters are selected with the main criteria of creating a full coverage of topology at the moment of transmitting. For example, a broadcaster starts transmitting the package to all its 8 neighboring nodes(horizontal, vertical, and diagonal cells of the 2D matrix) and the data will start propagating like a wave caused by a water droplet. When a packet arrives at a node which has already received the data sequence, it drops the packet and doesn't propagate any further. Intuitively, we can guess that resilience is improved by how well the broadcaster are distributed across the topology and how well the packages propagate until they are dropped by overlapping 'waves'. Bottlenecks and asymmetric propagation speed is mitigated by arranging the broadcaster and neighboring nodes in a circular shape, i.e. the fastest node in the center(broadcaster) and slower nodes at the edges of the propagation wave. A drawback of this topology is the high complexity to maintain when nodes fail and the propagation coverage needs to be calculated and rearranged.

3. **Tree Topology**: The three topology is represented as a binary tree and has the best scalability of the three used topology in the study. Similar to the list topology, resilience can be increased by creating preventive connections to successive nodes. The dual connection of the binary tree makes it convenient to handle faults since connections can be created crossing the branches and follow

protocols to reroute the traffic through those branches. Stalled flow can be mitigated to implement a max-heap instead of an unordered binary tree, this way preventing bottlenecks in slow link connected nodes.

These three topologies are a starting point to future and more complex structures such as a self-balancing directed graphs where the topology is ruled by a tradeoff of optimality in the data flow in exchange of maintainability in the presence of failures. Another possibility is to apply machine learning to weight the probability of failure in certain nodes by inspecting feature vectors of physical, software, and connection links features, this way optimizing the backup links among the topology.

5.3.3 The Communication Layer

We now explain the communication components that provide signaling features during a media streaming session. The communication layer abstracts all the burden of WebRTC, SocketCluster and the Security Library. This layer provides application transparency to the operations being executed under the hood.

1. *Collision Resistant Unique Identifier*: Since MioStream wraps the PeerJS library inside of the communication library, it needs to provide a collision resistant function to generate unique names with negligible probability of collisions. We use a randomly generated UUID which has 128 bits. The chance that 128 bits of having the same value can be calculated using probability theory (i.e., birthday problem), which can be shown to be negligible.

   $p(n) \approx 1 - e^{\frac{-n^2}{2*x}}$ where n is the number of bits.

   We use this *PeerID* to ensure unique WebRTC data channels between peers.

2. *Public and private key*: Every peer has its own RSA private and public key. We provide a parameter to configure the level of security enforcement among peers (e.g., key bit length $\geq$ 2048). These keys are used for the Authentication

[1]Extending our library to include Elliptic Curve Cryptography is straightforward.

Protocol described in the Security Layer. Our security model assumes a trusted *supervisor*.

3. *Connected Peers*: Every peer keeps its neighborhood set similar to [125, 126]. We then forward packets received and/or send to the application layer, the *video chunks* received.

4. *Signaling Capabilities*: Every peer connects to the *supervisor* using the communication library. Consequently, this layer handles all the signaling, heartbeats, and disconnections.

5. *WebRTC Data-channels*: We want to be able to build a variety of topologies using our implementation. Hence, the communication library provides a capability that is the equivalent to creating an *edge* $e(u, v) \in E$ in a graph $G = (V, E)$ where u and v are authenticated nodes in our network[2]. Consequently, we can create a rich set of topologies using the basic building blocks of *edges* (e.g., data channels see §5.2) and *vertices* (e.g., peers, *receivers* or *broadcasters*).

5.3.4 Security Layer

In this subsection, we discuss the security guarantees that our system offers.

1. *Authentication Protocol*: MioStream implements a handshake protocol, as shown in Figure 5. Although we took some ideas from TLS and DTLS [124]; MioStream includes the *componentType* in the topology as a new property.

    - MioStream receives a socket connection to its SocketCluster server (*supervisor*).
    - The *supervisor* sends the message `COLLECT_PEER_INFO_REQUEST`.

[2]In this paper, we use the terms nodes, vertices, and peers interchangeably.

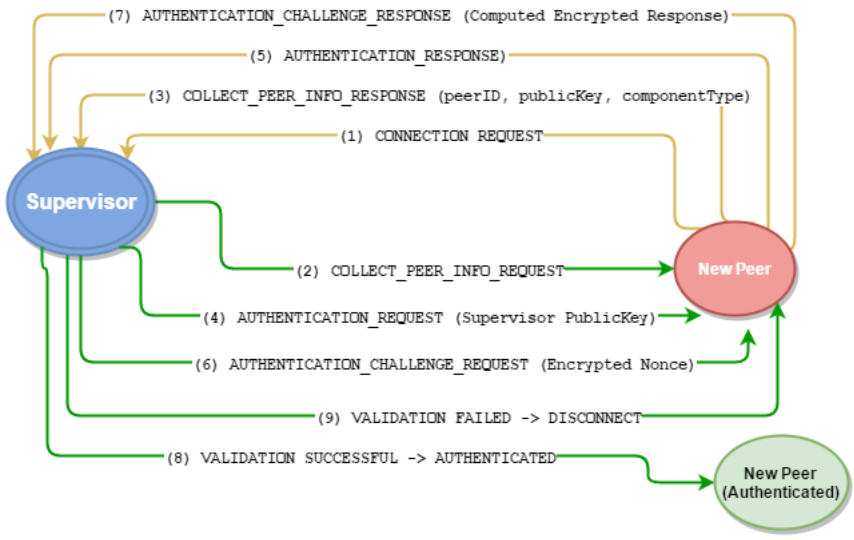


Fig. 5.4.: Authentication Finite State Machine.

- The peer replies with a `COLLECT_PEER_INFO_RESPONSE`, including its PeerID (used in the PeerJS connection), the Public Key, and a *componentType* (*Broadcaster* or *Receiver*).
- The *supervisor* sends its public key to the peer using `AUTHENTICATION_REQUEST` message[3].
- The peer *ACKs* with the message `AUTHENTICATION_RESPONSE`.
- The *supervisor* now computes a nonce including two random numbers and encrypts those numbers using the peers public key. We then send the nonce using `AUTHENTICATION_CHALLENGE_REQUEST`.
- The Peer receives the challenge request, decrypts the nonce, and computes a number based on two random numbers included in the nonce. The peer then proceeds to encrypt the response using the supervisor's public key. Finally, the peer answer with `AUTHENTICATION_CHALLENGE_RESPONSE`.

[3]Our future work will include a Certification Authority to deal with digital signatures, public keys, and tokens.

- The supervisor now checks the response given by the peers, if matches the one computed on the server, then the peer has now the state of **AUTHENTICATED** and is ready to initiate or receive streams.

2. *Integrity Validation of Video Chunks*: Habib et al. [100] proposed a probabilistic packet verification protocol. In particular, their *One Time Digest Protocol* (OTDP) utilizes a keyed hash to generate digests. Also, they divide the media file into segments or blocks; each block contains many packets. The server provides a secret key $K_i \in K$ for each segment i. Each supplying peer will generate digests using the provided keys and segments. Habib et al. motivate the integrity verification on video chunks presented in this project; therefore, to provide data integrity during peer-to-peer video streaming, MioStream adopts a similar approach but with some significant differences and contributions. First, this project only has a *broadcaster* for a specific stream. Second, we have implemented a personalized object (stream) lookup. Third, our implementation is live-oriented; hence we do not know all video-chunks in advance. The security model does not assume that connecting peers are trustworthy, so MioStream enforces authentication through the *supervisor* including a Public Key Infrastructure (PKI). Every *authenticated* peer is allowed to be part of our streaming sessions. The compute message digests are computed using a keyed SHA-256 ($HMAC$) and NodeJS Crypto library instead of digital signatures as in [100], but we use video chunks VC instead of segments. Therefore, we derived the following digest:

$$D_{ij} = h(K_i, VC_j)$$

where D_{ij} is the computed HMAC for the video chunk (VC) j and K_i is the *supervisor's* generated key for stream i.

Our implementation modifies the steps of One Time Digest Protocol (OTDP) to adapt it to our model as follows:

- The peer P_0 authenticates itself against the *supervisor*.
- P_0 requests a list of media streams to the *supervisor*.
- The *supervisor* provides a set of keys based on the search results.
- When the *broadcaster* (P_i) starts the streaming, it sends an initialization packet to P_0.
- The *broadcaster* (P_i) sends both data and digests to P_0.
- P_0 verifies random video chunks. Furthermore, P_0 can verify every packet depending on the *threshold* defined as a parameter.

The *broadcaster* computes the hash of every video chunk and sends the packet via the data channels connected to its *receivers*. Next, the *receiver* checks for the integrity of the packet and compares with the digest sent by the *broadcaster* or source of the media stream. In contrast to [100], we do not know all chunks of media in advance. Therefore, MioStream cannot compute random message digests and send those to every peer in our topology. We compute message digests of the chunks generated by the application layer. The *receiver* can verify at random one message digest to identify forgeries. MioStream includes a probabilistic approach using a uniform random number generator and comparing those generated numbers against a *threshold* defined as a parameter. The size of the message digests are 44 bytes (Base64 encoded).

5.3.5 Discussion

The system has several limitations that are mainly due to the client-server nature of web technologies. Currently, we can use two execution points: Any browser's ECMAScript run-time with WebRTC support, where users go to a website and automatically setup all the execution environment, and a Node.js application (*supervisor*) that currently runs on top of the V8 engine with I/O and OS's system call capabilities. The browser run-time is limited to its intrinsically high failure rate, especially

in mobile devices. Although MioStream lacks fault tolerance capabilities, the implementation is flexible enough to include more SocketCluster instances that can provide redundancy among *supervisors* (i.e., the communication layer can be extended to accommodate fault tolerance features). MioStream offers authentication and data integrity protocols; however, security poses a significant challenge because of the usually heterogeneous and high-churn network. Still, we can alleviate the attack surface utilizing trusted hardware in the *supervisor* and *relay points*.

5.4 Experiments

The experiments evaluate two main aspects of the system: *video streaming quality* under failures and *integrity verification of video chunks* overhead. First, we injected failures into the network to simulate how users would abandon the video stream (i.e., we use three distributions Binomial, Uniform, and Poisson). Hence, P2P connections between nodes in the network topology are aborted and packets get lost, ultimately causing stalled playbacks on other peers. The quality of the services was evaluated in terms of stalled playback and goodput metrics, as we will refer later. Stalled playbacks refer to delays or pauses in the viewer. Goodput measures the number of bytes received by the application layer (video chunks). Topology simulates the connections among the peers (e.g., linked list, tree, mesh). Second, we evaluated the performance of our the integrity verification of video chunks implemented in the security layer.

5.4.1 Experiment Setup

The experiment setup consisted of 83 machines with different specifications, ranging from four to eight 2.0 GHz cores, 8GB of RAM, connected to a 1Gbps local area network. All 83 machines act as *receivers*. In another machine, we set up the *supervisor* and the *monitor*. Finally, we stream the media from a separate machine running

Table 5.1.: Media Information of the WebM video used for the experiment

Media Information		
General	Format	WebM
	Format version	Version 2
	File size	24.0 MiB
	Duration	2mn 53s
	Overall bit rate mode	Variable
	Overall bit rate	1159 Kbps
Video	Format	VP8
	Width	1920 pixels
	Height	1080 pixels
	Display aspect ratio	16:9
	Frame rate mode	Constant
	Frame rate	29.970
Audio	Format	Vorbis
	Bit rate	128 Kbps
	Channel(s)	2 channels
	Sampling rate	44.1 KHz

the *broadcaster*. Hence, we use a total of 85 machines for all experiments. In table 5.1, we can see the technical specification of the video streamed for all experiments.

Each experiment is repeated on three different network topologies: *linked list*, *tree* and *mesh*. The *monitor* component has the capability of visualizing how nodes connect throughout the whole experiment. Basically, in the first set of experiments, we are particularly interested in observing the quality of service of the system under a non-reliable network (we use a pessimistic method of inducing failures).

5.4.2 Method of injecting failures

We define a failure as a disconnection of a *viewer* from the network. Since each *viewer* also forward the video to another peer, the flow of the stream is interrupted with every failure. When a *viewer* disconnects from the stream, that same node is reconnected to the network as a new incomer *viewer*. Hence, the network size is kept constant during the whole experiment. The rate of failures varies from one to five.

To better evaluate the behavior of the system in different failures scenario, we induce the failures based on three probability distribution: binomial, uniform, and Poisson. The *monitor* use the distribution to choose the failure peer from the network topology. For each distribution and network topology, we ran five experiments, varying the failure rate from one to five failures per second. The experiment consist on stream one minute of video across all the receivers. We took four samples in order to rule out spontaneous network congestion and processing load fluctuations.

The metrics used to evaluate the behavior of the system were:

- *Stalled:* Overall stalled playbacks in the stream during the experiment.
- *Total Stalled Time:* Overall playback stalled time during the experiment.
- *Average Stalled Time:* Overall average stalled time during the experiment.
- *Goodput.* Average goodput, or bytes received per second, on each peer during the experiment.

These metrics all together will describe the quality of the stream on different failure rates and scenarios.

5.4.3 Results

Figures 5.5-5.10 show the results for the stalled counter and stalled time for each distribution and network topology, varying from one to five failures per second. Under the **uniform** distribution, every peer in the topology has the same probability to fail during the experiment. As we can see in the figures, the stalled counted and the total stalled time experiences a gradual increment as the failure rate increases.

The **binomial** distribution induces failures mostly at the center of the topology. The failure peers were selected using 20 trials with a 80% of success probability. As we can observe, in the Figures 5.5-5.10, both stalled counter and total stalled time are very low compared to those values in the uniform distribution experiments. The low

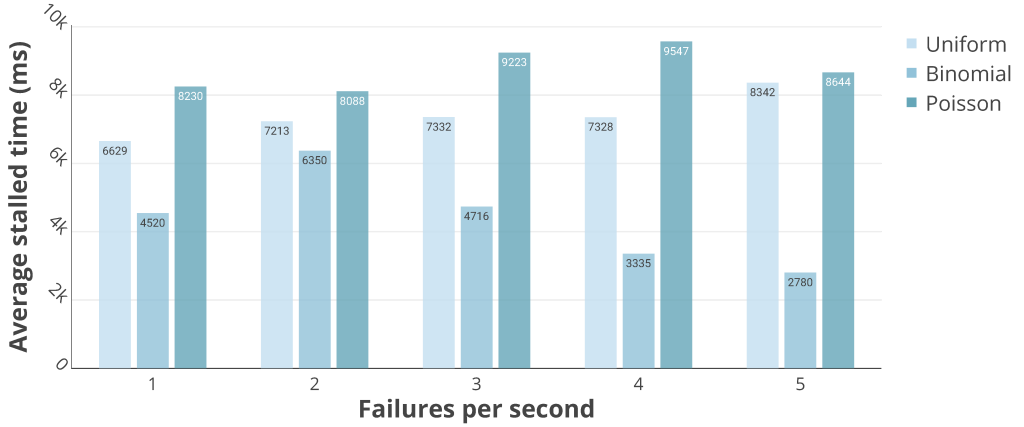


Fig. 5.5.: Linked list topology under induced failures (Uniform, Binomial, and Poisson distribution). Average stalled time of playback per peer.

stalled counter can be explained by analyzing the following scenario: A broadcaster sends a chunk of the media through all the topology at time t_1, and let say that this media chunk reaches the leaf nodes in the topology after n milliseconds of latency. If the binomial failure experiment induces failures at the center of the topology, it may be common that the chunks traveling the topology are in transit before the center, or after the center at the moment of failure. Since the chunk, if already delivered to the leaves or are before the center, there won't be stalled playbacks.

Finally, with a **Poisson** distribution the stalled counter and stalled time also gradually increments with the number of failures. In the sense of the location of the induced failure in the topology, this distribution behaves similarly to the binomial but closer to the root or broadcaster instead of the middle of the topology. Therefore, the values are higher than the other two distributions, since failures closer to the broadcast (root) impact more peers in the topology. Each faulty peer was chosen with a mean of 83/2 (half number of nodes in the topology).

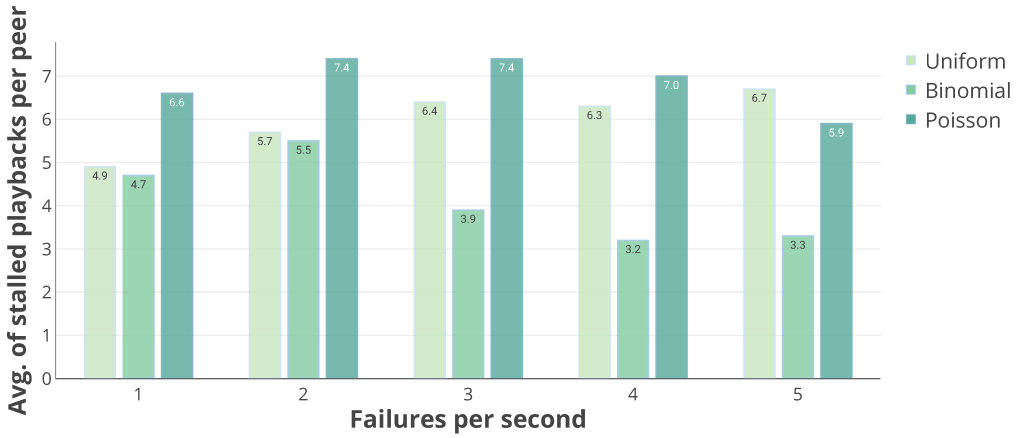


Fig. 5.6.: Linked list topology under induced failures (Uniform, Binomial, and Poisson distribution). Average stalled counter per peer.

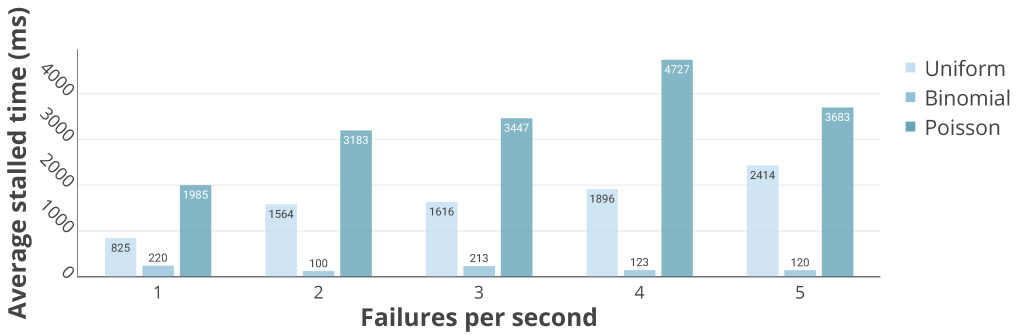


Fig. 5.7.: Tree topology under induced failures (Uniform, Binomial, and Poisson distribution). Average stalled time of playback per peer.

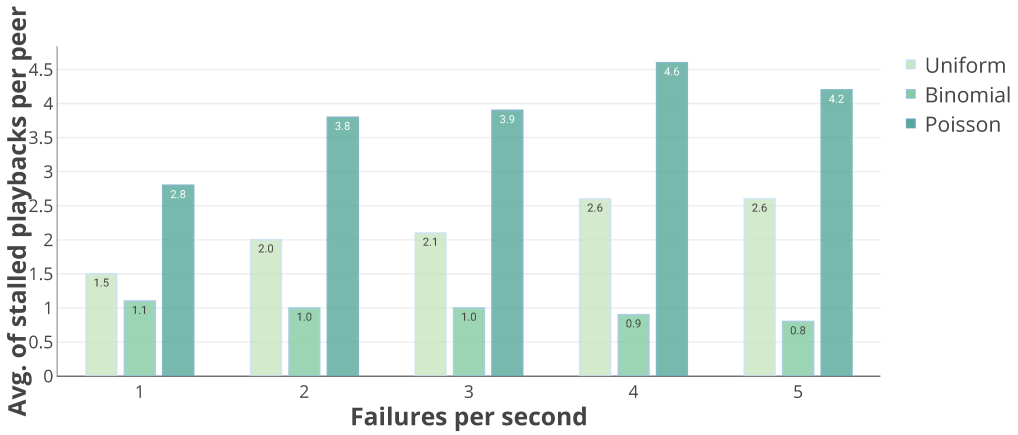


Fig. 5.8.: Tree topology under induced failures (Uniform, Binomial, and Poisson distribution). Average stalled counter per peer.

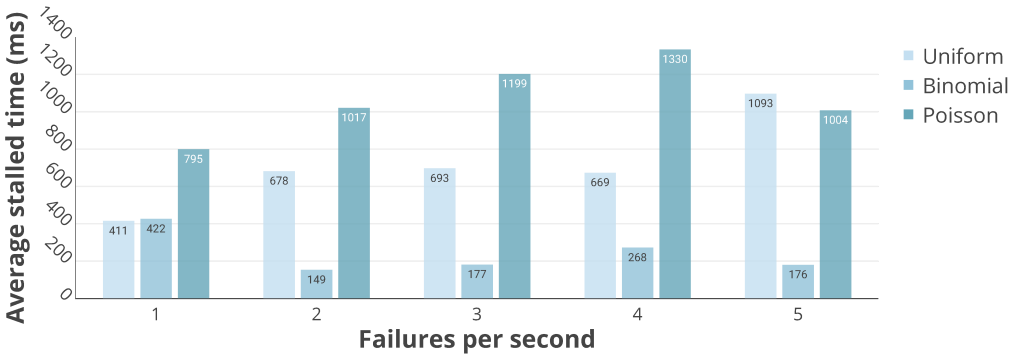


Fig. 5.9.: Mesh topology under induced failures (Uniform, Binomial, and Poisson distribution). Average stalled time of playback per peer.

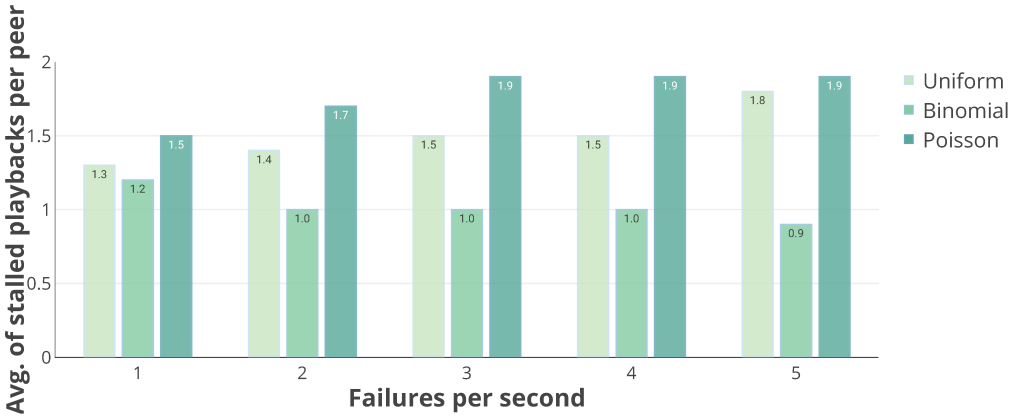


Fig. 5.10.: Mesh topology under induced failures (Uniform, Binomial, and Poisson distribution). Average stalled counter per peer.

5.4.4 Scalability and Goodput

The Figure 5.11 compares the resilience of each topology varying the failure rate. For the analysis, we choose the uniform distribution to have an equal probability of failure for each node of the topology. The viewers in the linked list topology experience the worst quality of service among the three topologies. A failure on this topology has the more significant impact since it affects the higher number of peers (recall that each peer is serially connected with each other). For instance, a failure on a peer close to the source (*broadcaster*) requires reconfiguring most of the network. The stalled count and the stalled time for the linked list topology at least double the values of the other two topologies. The stalled time per peer during the whole experiment goes from 2807 ms to 7325 ms for zero and five induced failures respectively.

The service quality increases drastically with more scalable topology like a tree. This topology is more resilient to failures and offers a shorter reconfiguration time in the presence of disconnection. However, the topology is highly vulnerable to failures near the root or source (*broadcaster*), as we observed earlier. The stalled time in this topology varies from 169 ms (an order of magnitude less than the linked list topology) to 2410 ms. However, the best quality of service is offered by the mesh topology. Even though the tree and mesh topology behave similar with minimal disconnections, the mesh topology is more robust to failures. The mesh offers faster reconfiguration and overcomes the vulnerability to failures near the source by providing more alternative connection than the tree topology. The peers in the mesh topologies experience fewer stalls with respect the other two topologies, and the average stalled time is less than half of the respective value in a tree. The stalled time goes from 169 ms (similar than tree topology) to 1093 ms. Furthermore, as we can observe from Figure 5.12, the mesh network offers the highest goodput per peer, which is 10% than the goodput on the tree topology.

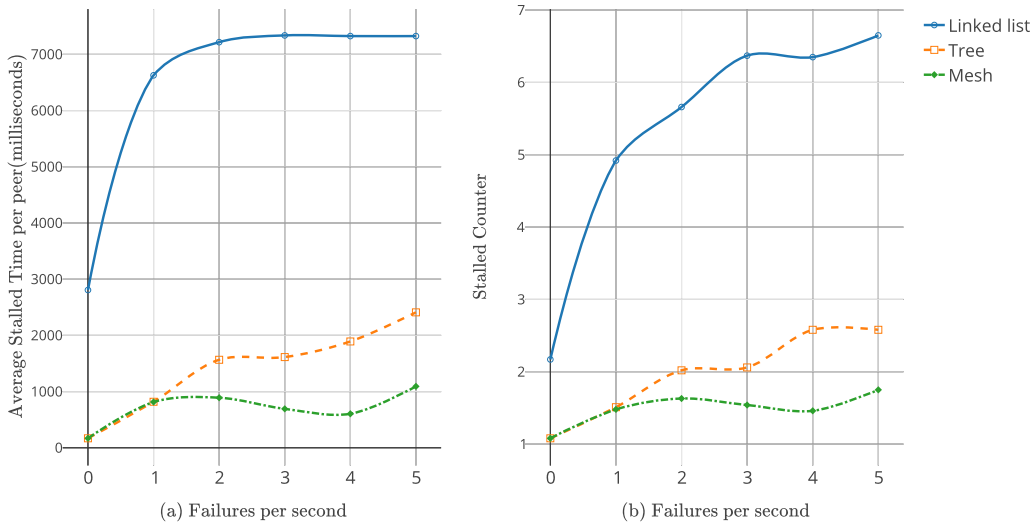


Fig. 5.11.: Scalability under induced failures (Uniform, Binomial, and Poisson distribution) according to network topology. (a) average stalled time of playback per peer; (b) average stalled counter per peer.

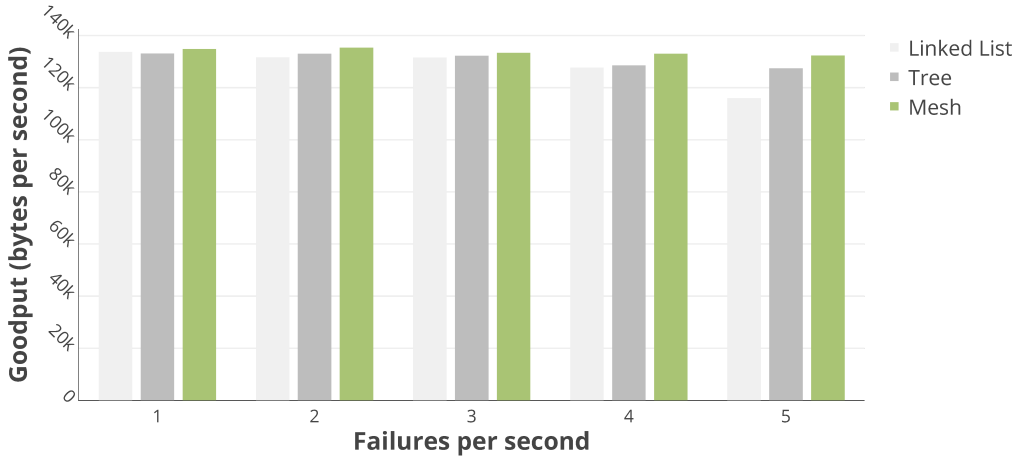


Fig. 5.12.: Goodput (bytes per second) under induced failures using the uniform distribution.

5.4.5 Integrity Validation of Video Chunks Overhead

We now discuss the total number of stalled video chunks due to the integrity validation. In Figure 5.13, 5.14 we can observe different scenarios using thresholds[4] starting from 10% up to 50%. The average delayed video chunks increase according to the threshold when the probability limit was below 10 percent the quality of service improved considerably.

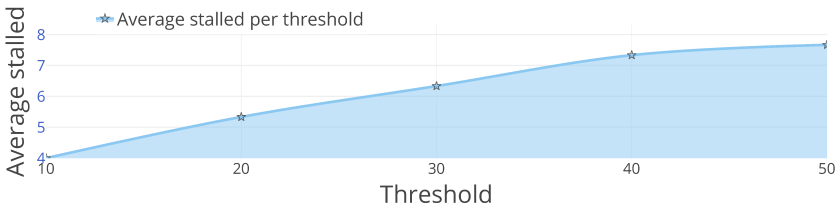


Fig. 5.13.: Average Number of Stalled Video Chunks per Threshold.

[4]The threshold represents the probability of the integrity analysis of a video chunk.

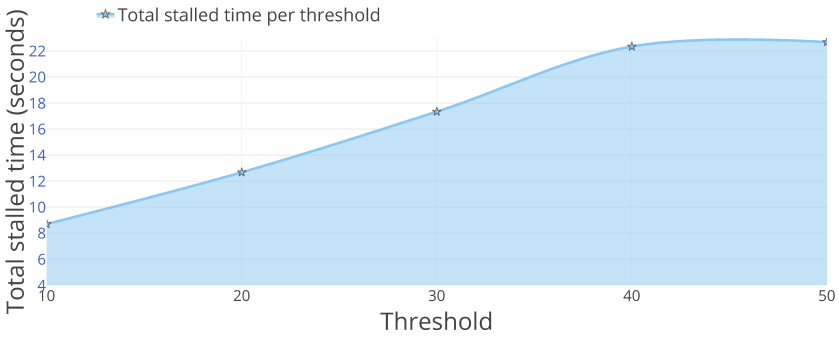


Fig. 5.14.: Total Stalled Time per Threshold.

5.4.6 Security Layer Overhead

Experimental Setup

Single-machine experiments were run using a machine with Intel Core i7 (4 cores @2.8GHz, 8MiB cache) with 16GiB of RAM. We tested the NodeJS Crypto Library [127].

HMAC

The experiment consists of 10 runs of HMAC instantiations and 10 HMAC digests generation. The figure 5.15 shows the overhead caused by this library per HMAC digest generation.

5.4.7 Analysis

The main contribution of our work is to expose a proof of concept of a decentralized (in the sense of content distribution) of a media content streaming using peer-to-peer communication. Also, we demonstrated that it is feasible to maintain considerably good quality in exchange for resource cost and network overhead. All experiments with different distributions and failure rates aimed to simulate real-life failure scenarios, and how the system will behave in such extreme conditions. So

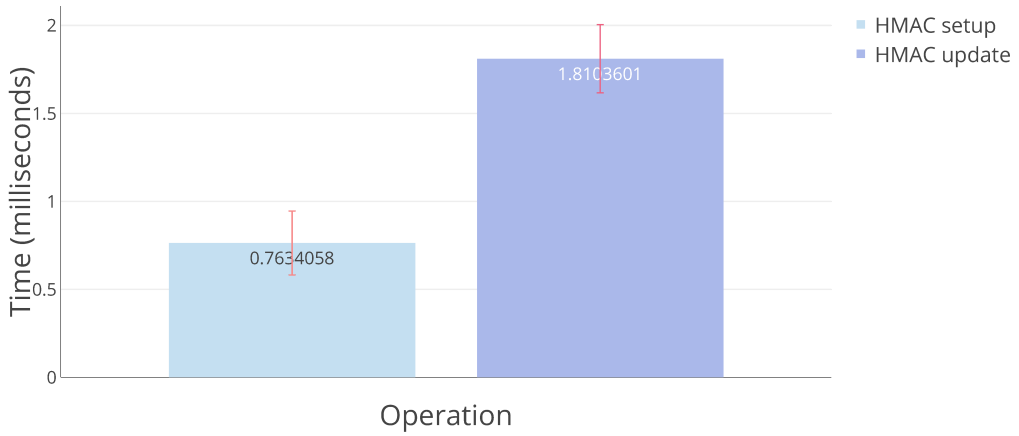


Fig. 5.15.: HMAC setup and digest generation.

far, we have shown in figures 5.5-5.10, how the streaming quality is affected and how stalled playbacks scale as failure rate increases. Furthermore, the average stalled time in the overall topology is below one second in most scenarios, independent of the distribution. Hence, the total stream interruptions in the topology are not longer than one second, and in most cases are milliseconds which can be tolerated and perhaps be so brief that are not even perceived by the viewer.

Besides, the study offers a comparison between different network topologies, highlighting the benefits and drawbacks of each one. As shown in Figures 5.11- 5.12, the mesh topology is more robust to failures than the other topologies analyzed, offering a higher goodput and fewer stalled playbacks. Moreover, the stalled playback on a node equals 1.8% of the total video play. The amount is insignificant if we take into consideration that the experiments aimed to simulate extreme conditions, with as many as five failures per seconds. Finally, the integrity check includes a considerable overhead; hence an acceptable threshold will be lower than ten percent of the total of chunks.

5.4.8 Use cases

We have shown how to maintain considerable good quality in a live stream topology using mainly three components: *supervisor*, *broadcaster*, *receivers*. We are presenting our system in the context of live media streaming. On the other hand, there is no reason why this same concept cannot apply to other scenarios which also relies on CDN to distribute content. Some of the potential scenarios to apply the same architecture are:

- *Mobile applications content distribution*: Recently, many mobile applications (i.e., social media), distribute media content in a graph fashion. I.e., friends to friends. This case will be even simpler than our scenario since the stream of content is not real-time. Implementing a similar architecture would significantly reduce the cost of scale-out large mobile applications which need to distribute media content.

- *Software Updates*: Modern software, such as those built with Electron [120], supports hot updates, meaning that the application can be downloaded in the background and installed inside of a sandbox without the need of natively uninstall and install a different binary. The same concept applies to distributed hot updates without requiring large content distribution networks.

5.5 Future Work

In the experiments section, we showed the potential of our system. However, the system should have many other qualities to be considered robust. We aim to improve the quality of the streaming further and prevent stalled playbacks. First, MioStream will include high-availability in case of failures through *supervisor* replication in our topology. Second, MioStream's communication layer will detect *relay points* through machine learning techniques. The relay points aim to avoid stalled playbacks and

congested paths. Finally, MioStream will provide new enhancements in the security layer (e.g., Elliptic Curve Cryptography).

5.6 Related Work

MioStream shares some characteristics as Peer2View [128]. For example, Peer2-View is built on top of UDP-based transport library which provides reliability and security guarantees. Also, they authenticate their users against a central authority (e.g., in our case the *supervisor*). They provide encryption using SSL and integrity. However, there are many differences between MioStream and Peer2View; the latter solves the NAT problem with NATCracker [128], the former uses WebRTC, ICE, STUN and TURN. Moreover, Peer2View is a commercial peer-to-peer live video streaming (P2PLS) using Content Distribution Network (CDN). In contrast, MioStream is an Open Source system.

We now compare against an Energy-Efficient Mobile P2P Video Streaming [129]. Their goal is to minimize and balance the energy consumption of participating devices in the video streaming session. Similarly, our goal is to provide the best quality of video streaming across all participants of our video streaming session balancing the traffic in the network using data structures to provide congestion control (maximizing network utilization). Wichtlhuber et al. have energy consumption as its primary goal; we can extend our vision and minimize the usage of mobile devices to avoid degradation of our network and maximize battery life in mobile devices.

We based much of our work on PROMISE and CollectCast [97,98,100]. However, there are many important differences. For instance, we assume constant bandwidth for each peer (e.g., we do not have the notion of partial bandwidth contribution, MioStream uses all bandwidth available on the WebRTC data channel), and there is only one *broadcaster* for every stream. Available streams are provided by the *supervisor* to the *receivers*. In [97, 98] they exploit the properties of the underlying network in which one receiver can collect data from multiple senders. PROMISE is

independent of the underlying P2P network. Therefore, PROMISE can be deployed using Pastry [125], Chord [130], and CAN [131]. MioStream depends on WebSockets and WebRTC architecture; the *supervisor* exposes the object lookup. Furthermore, MioStream implements its personalized object lookup, called *Stream Manager*.

In their paper, Rhinow et al. provided an analysis of the feasibility of implementing live video streaming into web applications [93]. Although they offered performance measures, they did not include an open source repository, nor they studied the security implications and quality of service under failures. MioStream exposes an analysis of QoS in the presence of failures. Lopez et al. introduced Kurento Media Server an open source WebRTC media server that offers features such as group communication, recording, routing, transcoding, and mixing [94]. They provide APIs that facilitates the development of web-based video applications. Lopez et al. did not present an analysis of the quality of service under failures or security attacks—i.e., an adversarial environment. Garcia et al. introduced NUBOMEDIA an open source cloud platform as a service (PaaS) designed for WebRTC services [95]. NUBOMEDIA exposes a set of APIs that facilitate the development of WebRTC applications.

5.7 Conclusion

An increasing number of companies require serverless privacy-preserving frameworks for live-video streaming. We presented the design, implementation, and evaluation of a novel P2P service based on WebRTC (web browsers with Real-Time Communications) called MioStream. MioStream directly targets this essential and practical application. This paper contributes an open-source alternative for distributed media streaming that runs on the edge of the network without incurring in costly and extensive CDN infrastructure. We contributed a unique mix of algorithms using WebRTC data channels. Also, under network degradation and high-churn environments, MioStream restructures the topology dynamically. MioStream provides authentication, privacy, and integrity of video chunks.

This paper exposed a set of micro-benchmarks to measure the quality of service under network degradation and high churn environment (inducing failures). Failures are induced in the topology using three distributions (e.g., Uniform, Binomial, and Poisson) and three network topologies such as mesh, tree, and linked list. The Mesh topology offers the highest goodput per peer; the stalled time goes from 169 ms to 1093 ms, the stalled playback on a node equals 1.8% of the total video play. Our results show the feasibility of this proof of concept under high-churn environments. We conclude that the total stream interruptions in the topology are not longer than one second under the binomial distribution, and in most cases are milliseconds. The integrity check includes a considerable overhead.

6. CONCLUSION

In this dissertation, we have introduced a lineage of research that has proposed moving the computation kernels out of the traditional cloud computation model to provide practical frameworks that compute on (un)encrypted graph-structured data. Further, we also examined the cloud-model drawbacks related to latency, throughput, and privacy. Similarly, this dissertation provided an analysis of the current solutions for the supply chain that rely on blockchain technologies as a building block; we provided a practical solution uniquely utilizing current techniques to accomplish auditable, oblivious, and automated certification process in the supply chain. Additionally, we analyzed state-of-the-art graph data stores and their shortcomings related to usability and privacy. We investigated the access locality exhibited by traversal algorithms on top of the graph model to present novel layout algorithms that can provide authentication and auditability features. These techniques and derived open-source frameworks have introduced new auditable, automated, oblivious, and privacy-preserving methods for use cases in which the preservation of graph-data ownership is imperative.

To address the real-world challenge of certification frameworks that preserves data ownership in the supply chain, we presented AGAPECert. AGAPECert is an auditable, generalized, privacy-enabling certification framework that protects the confidentiality of data, participants, and code. AGAPECert utilizes a unique mix of blockchain technologies, trusted execution environments, and a real-time graph-based API to define for the first time Oblivious Smart Contracts (OSCs) that generate auditable Private Automated Certifications (PACs). AGAPECert offers pragmatic performance and is generalizable to many use cases and data types. AGAPECert has a significant impact providing an open source [49] framework that can be adopted as a standard in any regulated environment to keep sensitive data private while enabling an automated workflow.

AuditGraph.io introduced an auditable, authenticated, and blocked graph processing model. AuditGraph.io exploits the structure of the network to generate blocked individual authenticated sets with high-locality. AuditGraph.io utilizes the structure of the access hierarchy (authentication graph) to define the ordering of blocks and entities in the disk or memory layout. A *similarity graph* with edges representing the relationship between authenticated sets (edge betweenness centrality as weight) determines the final ordering of blocks in shards or memory blocks. AuditGraph.io contributes auditable computation on top of the graph-model utilizing trusted execution environments and blockchain technologies.

We address the usability challenges presented by graph data stores introducing TruenoDB. TruenoDB is a user-friendly (i.e., easy to use) distributed and scalable graph datastore and computational engine. We demonstrated TruenoDB flexibility and usability through a variety of applications. TruenoDB provided facile graph analysis and visualization via the Web UI. The integrated distributed graph computation (i.e., Spark GraphX) is extremely useful when handling large datasets. We validated TruenoDB excellent performance providing evidence through comparisons against state-of-the-art commercial graph databases and a variety of datasets (e.g., biology datasets, social networks, citations networks, and the LDBC benchmark).

Finally, MioStream presented the design, implementation, and evaluation of a new P2P service based on WebRTC (web browsers with Real-Time Communications) called MioStream. MioStream contributes an open-source alternative for distributed media streaming that runs on the edge of the network without incurring costly and extensive CDN infrastructure. We contributed a unique mix of algorithms using WebRTC data channels. Also, under network degradation and high-churn environments, MioStream restructures the topology dynamically using a graph-based representation in the supervisor. MioStream provides authentication, privacy, and integrity of video chunks.

In conclusion, the techniques for auditability, authentication, privacy-preserving computation, and trust model presented in this dissertation highlight the impact-

ful advantages of all these solutions. AGAPECert proved our work to be practical; moreover, we expect the adoption of our solution in parts of the supply chain. Similarly, despite an in-memory only implementation (e.g., in a docker container), AuditGraph.io has a significant potential to provide authentication and auditability enhancements for graph datastores.

REFERENCES

REFERENCES

[1] "Neo4j." [Online]. Available: http://neo4j.com

[2] "JanusGraph (release 0.5.2)," https://janusgraph.org/, Apr. 2020, online.

[3] "TruenoDB." [Online]. Available: https://github.com/TruenoDB

[4] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One Trillion Edges : Graph Processing at Facebook-Scale," *Vldb*, vol. 8, no. 12, pp. 1804–1815, 2015.

[5] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012*. USENIX Association, 2012, pp. 31–46.

[6] Y. Xia, L. Nai, and J. H. Lai, "Towards Balance-Affinity Tradeoff in Concurrent Subgraph Traversals," *Proceedings - 2015 IEEE 29th International Parallel and Distributed Processing Symposium, IPDPS 2015*, pp. 936–945, 2015.

[7] "ArangoDB," https://www.arangodb.com/, Apr. 2020, online.

[8] "Neo4j security checklit," https://neo4j.com/docs/operations-manual/current/security/checklist/, Apr. 2020, online.

[9] "ArangoDB Security," https://www.arangodb.com/why-arangodb/arangodb-enterprise/arangodb-enterprise-security/, Apr. 2020, online.

[10] "IBM Food Trust," https://www.ibm.com/blockchain/solutions/food-trust, 2019, [Online; accessed April 16, 2019].

[11] B. Pirus, "BeefChain receives first USDA certification for a blockchain company," https://www.forbes.com/sites/benjaminpirus/2019/04/25/beefchain-receives-first-usda-certification-for-a-blockchain-company/#3f678c6c7607, 2019, [Online; accessed April 27, 2019].

[12] Lowry, "Lowry Solutions Sonaria Blockchain Platform: Making IIoT into Blockchain," https://lowrysolutions.com/, 2019.

[13] Ripe, "Ripe.io: Blockchain of Food," http://ripe.io, 2019.

[14] OriginTrail, "OriginTrail: Blockchain-powered Data Exchange Protocol for Interconnected Supply chains," http://origintrail.io, 2019.

[15] SAP, "SAP: Blockchain Service," https://www.sap.com/products/leonardo/blockchain.html, 2019.

[16] M. Brandenburger, C. Cachin, R. Kapitza, and A. Sorniotti, "Blockchain and trusted computing: Problems, pitfalls, and a solution for hyperledger fabric," 2018.

[17] J. Ekberg, K. Kostiainen, and N. Asokan, "The untapped potential of trusted execution environments on mobile devices," *IEEE Security Privacy*, vol. 12, no. 4, pp. 29–37, July 2014.

[18] V. Costan and S. Devadas, "Intel SGX Explained," *Cryptology ePrint Archive, Report 2016/086*, 2016.

[19] W. Dai, C. Dai, K. R. Choo, C. Cui, D. Zou, and H. Jin, "Sdte: A secure blockchain-based data trading ecosystem," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 725–737, 2020.

[20] M. Russinovich, "Announcing the Confidential Consortium Blockchain Framework for enterprise blockchain networks," https://azure.microsoft.com/en-us/blog/announcing-microsoft-s-coco-framework-for-enterprise-blockchain-networks/, 2017, [Online; accessed June 10, 2019].

[21] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An Oblivious and Encrypted Distributed Analytics Platform," *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[22] Trellis, "The Trellis Framework," https://github.com/trellisfw, 2019.

[23] OADA, "The Open Ag Data Alliance: a API framework for automated, cross-industry data exchange," https://github.com/oada, 2019.

[24] Intel, "Intel Software Guard Extensions Documentation," https://software.intel.com/en-us/articles/intel-sgx-web-based-training, 2015.

[25] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 1996.

[26] Intel, "Intel Software Guard Extensions (Intel SGX) SDK for Linux* OS," https://01.org/intel-softwareguard-extensions, 2019.

[27] ——, "Intel Enhanced Privacy ID Ecosystem," https://intel-epid-sdk.github.io/ecosystem/, 2017.

[28] "Intel SGX DCAP (release 1.6)," https://download.01.org/intel-sgx/sgx-dcap/1.6/linux/docs/, Apr. 2020, intel.

[29] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," http://bitcoin.org/bitcoin.pdf, 2008.

[30] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, ser. OSDI '99. Berkeley, CA, USA: USENIX Association, 1999, pp. 173–186. [Online]. Available: http://dl.acm.org/citation.cfm?id=296806.296824

[31] D. Wood, "Ethereum: A Secure Decentralized Generalized Distributed Ledger," 2014.

[32] Trellis, "Trellis Authorization Protocol," https://github.com/OADA/oada-docs/blob/master/rest-specs/Authentication_and_Authorization.md, 2016.

[33] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "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, p. 991–1008. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[34] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 973–990. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[35] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 142–157.

[36] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, "Obliviate: A data oblivious filesystem for intel sgx," *Network and Distributed System Security Symposium*, 2018.

[37] Y. Xiao, M. Li, S. Chen, and Y. Zhang, "Stacco: Differentially analyzing side-channel traces for detecting ssl/tls vulnerabilities in secure enclaves," in *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*, ser. CCS '17. ACM, 2017, pp. 859–874.

[38] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx," in *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*, ser. CCS '17, vol. 2017. ACM, 2017, pp. 2421–2434.

[39] "L1 Terminal Fault - Security Advisory," https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault, Apr. 2020, intel.

[40] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, Stanford, CA, USA, 2009, aAI3382729.

[41] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques*, ser. EUROCRYPT'99. Berlin, Heidelberg: Springer-Verlag, 1999, pp. 223–238. [Online]. Available: http://dl.acm.org/citation.cfm?id=1756123.1756146

[42] A. Paverd, A. Martin, and I. Brown, "Modelling and Automatically Analyzing Privacy Properties for Honest-but-Curious Adversaries," https://www.cs.ox.ac.uk/people/andrew.paverd/casper/casper-privacy-report.pdf. [Online]. Available: https://www.cs.ox.ac.uk/people/andrew.paverd/casper/casper-privacy-report.pdf

[43] "Microsoft SEAL (release 3.5)," https://github.com/Microsoft/SEAL, Apr. 2020, microsoft Research, Redmond, WA.

[44] "OpenEnclave," https://github.com/openenclave/openenclave, 2020, [Online; accessed April 20, 2020].

[45] M. Zaharia, M. Chowdhury, T. Das, and A. Dave, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," *NSDI'12 Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pp. 2–2, 2012. [Online]. Available: https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

[46] O. Weisse, V. Bertacco, and T. Austin, "Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, vol. 45, no. 2. ACM, 2017, pp. 81–93.

[47] "Chai NPM Library," https://www.chaijs.com/, 2020, [Online; accessed April 20, 2020].

[48] "Mocha NPM Library," https://mochajs.org/, 2020, [Online; accessed April 20, 2020].

[49] AgapeCert, "AGAPECert," https://github.com/agapecert, 2020.

[50] A. Yasar, B. Gedik, and H. Ferhatosmanolu, "Distributed block formation and layout for disk-based management of large-scale graphs," *Distributed and Parallel Databases*, vol. 35, no. 1, pp. 23–53, 2017.

[51] P. Xie and E. P. Xing, "Cryptgraph: Privacy preserving graph analytics on encrypted graph," *CoRR*, vol. abs/1409.5021, 2014. [Online]. Available: http://arxiv.org/abs/1409.5021

[52] I. Hoque and I. Gupta, "Disk layout techniques for online social network data," *IEEE Internet Computing*, vol. 16, no. 3, pp. 24–36, 2012.

[53] R. Soulé and B. Gedik, "RailwayDB: adaptive storage of interaction graphs," *VLDB Journal*, vol. 25, no. 2, pp. 151–169, 2016.

[54] A. Kundu and E. Bertino, "How to authenticate graphs without leaking," *Proceedings of the 13th International Conference on Extending Database Technology*, pp. 609–620, 2010. [Online]. Available: http://doi.acm.org/10.1145/1739041.1739114

[55] M. U. Arshad, A. Kundu, E. Bertino, K. Madhavan, and A. Ghafoor, "Security of graph data," *Proceedings of the 4th ACM conference on Data and application security and privacy - CODASPY '14*, pp. 223–234, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2557547.2557564

[56] A. Kundu and E. Bertino, "Privacy-preserving authentication of trees and graphs," *International Journal of Information Security*, vol. 12, no. 6, pp. 467–494, 2013.

[57] E. Stefanov, M. van Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: An extremely simple oblivious ram protocol," 2012.

[58] S. Palacios, A. Ault, J. V. Krogmeier, B. Bhargava, and C. G. Brinton, "Agapecert: An auditable, generalized, automated, and privacy-enabling certification framework with oblivious smart contracts," Forthcoming, submitted.

[59] "Elasticsearch." [Online]. Available: https://github.com/elastic/elasticsearch

[60] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX : Graph Processing in a Distributed Dataflow Framework," *11th USENIX Symposium on Operating Systems Design and Implementation*, pp. 599–613, 2014. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez

[61] "NetworkX." [Online]. Available: https://networkx.github.io/

[62] "Sigma." [Online]. Available: http://sigmajs.org/

[63] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. 12, pp. 7821–7826, 2002. [Online]. Available: http://www.pnas.org/cgi/doi/10.1073/pnas.122653799

[64] G. Chartrand and P. Zhang, *A First Course in Graph Theory*, ser. Dover books on mathematics. Dover Publications, 2012. [Online]. Available: https://books.google.com/books?id=ocIr0RHyI8oC

[65] S. Palacios, K. Solaiman, P. Angin, A. Nesen, B. Bhargava, Z. Collins, A. Sipser, M. Stonebraker, and J. Macdonald, "Wip-skod: A framework for situational knowledge on demand," in *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*. Springer, 2019, pp. 154–166.

[66] OADA, "OADA cache," https://github.com/OADA/oada-cache. [Online]. Available: https://github.com/OADA/oada-cache

[67] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[68] "METIS." [Online]. Available: http://glaros.dtc.umn.edu/gkhome/metis/metis/overview

[69] A.-L. Barabasi and R. Albert, "Emergence of scaling in random networks," *Science (Washington)*, vol. 286, no. 5439, pp. 509–512, 1999. [Online]. Available: http://search.proquest.com/docview/743701710/

[70] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection," 2014. [Online]. Available: https://snap.stanford.edu/snap/

[71] "Titan distributed graph database." [Online]. Available: http://thinkaurelius.github.io/titan/

[72] J. Webber, "A Programmatic Introduction to Neo4j."

[73] M. A. Rodriguez, "The Gremlin Graph Traversal Machine and Language," 2015. [Online]. Available: https://arxiv.org/abs/1508.03843

[74] "Cytoscape." [Online]. Available: http://www.cytoscape.org/

[75] M. Bastian, S. Heymann, and M. Jacomy, "Gephi: An Open Source Software for Exploring and Manipulating Networks," *Third International AAAI Conference on Weblogs and Social Media*, pp. 361–362, 2009. [Online]. Available: http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154

[76] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "GraphLab: A New Framework for Parallel Machine Learning," *Conference on Uncertainty in Artificial Intelligence*, 2010. [Online]. Available: http://arxiv.org/abs/1006.4990

[77] M. Malak and R. East, *Spark GraphX in Action*. Manning, 2016.

[78] Ooyala, "Spark Job Server." [Online]. Available: https://github.com/spark-jobserver/spark-jobserver

[79] Apache, "GraphX Programming Guide." [Online]. Available: https://spark.apache.org/docs/latest/graphx-programming-guide.html

[80] A. Chatr-Aryamontri, R. Oughtred, L. Boucher, J. Rust, C. Chang, N. K. Kolas, L. O'Donnell, S. Oster, C. Theesfeld, A. Sellam, C. Stark, B. J. Breitkreutz, K. Dolinski, and M. Tyers, "The BioGRID interaction database: 2017 update," *Nucleic Acids Research*, vol. 45, no. D1, pp. D369–D379, 2017.

[81] Biogrid, "Biological General Repository for Interaction Datasets." [Online]. Available: https://thebiogrid.org/

[82] M. Malak and R. East, *Spark GraphX in action*, 2016.

[83] J. Maharana and J. H. Schwarz, "Noncompact symmetries in string theory," *Nuclear Physics, Section B*, vol. 390, no. 1, pp. 3–32, 1993.

[84] S. Kar, S. Khastgir, and A. Kumar, "An Algorithm to Generate Classical Solutions for String Effective Action," 1992. [Online]. Available: https://arxiv.org/abs/hep-th/9201015

[85] A. Iosup, T. Hegeman, W. L. Ngai, S. Heldens, A. Prat-Perez, T. Manhardt, H. Chafi, M. Capot, N. Sundaram, M. Anderson, I. G. T, and Y. Xia, "LDBC Graphalytics : A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms , a Technical Report," *Vldb*, vol. 9, no. 13, pp. 1317–1328, 2016.

[86] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "TAO: Facebook's distributed data store for the social graph," *Usenix Atc'13*, pp. 49–60, 2013.

[87] A. Dave, A. Jindal, L. Li, R. Xin, J. Gonzalez, and M. Zaharia, "GraphFrames: An Integrated API for Mixing Graph and Relational Queries," *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems SE - GRADES '16*, 2016. [Online]. Available: http://dx.doi.org/10.1145/2960414.2960416

[88] G. a. Salazar, A. Meintjes, G. K. Mazandu, H. a. Rapanoël, R. O. Akinola, and N. J. Mulder, "A web-based protein interaction network visualizer." *BMC bioinformatics*, vol. 15, p. 129, 2014. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4029974/

[89] B. Shao, H. Wang, and Y. Li, "Trinity," in *Proceedings of the 2013 international conference on Management of data - SIGMOD '13*, 2013, p. 505. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2463676.2467799

[90] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: A peta-scale graph mining system - Implementation and observations," in *Proceedings - IEEE International Conference on Data Mining, ICDM*, 2009, pp. 229–238.

[91] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," *Proceedings of the 2010 international conference on Management of data - SIGMOD '10*, pp. 135–146, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1807167.1807184

[92] A. Kyrola and C. Guestrin, "GraphChi-DB: Simple Design for a Scalable Graph Database System – on Just a PC," *ArXiv e-prints*, 2014.

[93] F. Rhinow, P. P. Veloso, C. Puyelo, S. Barrett, and E. O. Nuallain, "P2P Live Video Streaming in WebRTC," *Computer Applications and Information Systems (WCCAIS), 2014 World Congress on*, pp. 1–6, 2014.

[94] L. Lopez, M. Paris, S. Carot, B. Garcia, M. Gallego, F. Gortazar, R. Benitez, J. A. Santos, D. Fernandez, R. T. Vlad, I. Gracia, and F. J. Lopez, "Kurento: The WebRTC Modular Media Server," *Proceedings of the 2016 ACM on Multimedia Conference*, pp. 1187–1191, 2016. [Online]. Available: http://doi.acm.org/10.1145/2964284.2973798

[95] B. Garcia, L. Lopez, F. Gortzar, M. Gallego, and G. A. Carella, "NUBOMEDIA: The first open source WebRTC PaaS," *MM 2017 - Proceedings of the 2017 ACM Multimedia Conference*, pp. 1205–1208, 2017.

[96] S. Palacios, V. Santos, E. Barsallo, and B. Bhargava, "Miostream: a peer-to-peer distributed live media streaming on the edge," *Multimedia Tools and Applications*, Jan 2019. [Online]. Available: https://doi.org/10.1007/s11042-018-6940-2

[97] M. Hefeeda, A. Habib, B. Botev, D. Xu, and B. Bhargava, "PROMISE: Peer-to-Peer Media Streaming Using CollectCast," *Proceedings of the eleventh ACM international conference on Multimedia - MULTIMEDIA '03*, p. 45, 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=957013.957022

[98] M. Hefeeda, A. Habib, D. Xu, B. Bhargava, and B. Botev, "CollectCast: A peer-to-peer service for media streaming," *Multimedia Systems*, vol. 11, no. 1, pp. 68–81, 2005.

[99] "MioStream." [Online]. Available: https://github.com/maverick-zhn/miostream

[100] A. Habib, D. Xu, M. J. Atallah, B. Bhargava, and J. C.-I. Chuang, "Verifying data integrity in peer-to-peer media streaming," in *Multimedia Computing and Networking 2005*, S. Chandra and N. Venkatasubramanian, Eds., vol. 5680, International Society for Optics and Photonics. SPIE, 2005, pp. 1 – 12. [Online]. Available: https://doi.org/10.1117/12.587201

[101] "WebRTC." [Online]. Available: https://webrtc.org/

[102] "WebRTC use cases." [Online]. Available: https://tools.ietf.org/html/rfc7478

[103] "WebSockets." [Online]. Available: https://www.websocket.org/

[104] I. Grigorik, *High Performance Browser Networking*, 2013.

[105] P. R. S. Loreto Salvatore, *Real-Time Communication with WebRTC: Peer-to-Peer in the Browser*, 2014.

[106] "SocketCluster.io." [Online]. Available: http://socketcluster.io

[107] "The WebSocket Protocol RFC6455," 2011. [Online]. Available: https://tools.ietf.org/html/rfc6455

[108] "How does HTTP/2 solve the Head of Line blocking (HOL) issue," 2018. [Online]. Available: https://community.akamai.com/customers/s/article/How-does-HTTP-2-solve-the-Head-of-Line-blocking-HOL-issue

[109] C. Vogt, M. J. Werner, and T. C. Schmidt, "Leveraging WebRTC for P2P content distribution in web browsers," *Proceedings - International Conference on Network Protocols, ICNP*, 2013.

[110] "ICE." [Online]. Available: https://tools.ietf.org/id/draft-ietf-ice-rfc5245bis-13.html

[111] "Session Traversal Utilities for NAT (STUN)," 2008. [Online]. Available: https://tools.ietf.org/html/rfc5389

[112] "Traversal Using Relays around NAT (TURN)," 2010. [Online]. Available: https://tools.ietf.org/html/rfc5766

[113] "SDP: Session Description Protocol," 2006. [Online]. Available: https://tools.ietf.org/html/rfc4566

[114] "Datagram Transport Layer Security," 2006. [Online]. Available: https://tools.ietf.org/html/rfc4347

[115] "Stream Control Transmission Protocol," 2007. [Online]. Available: https://tools.ietf.org/html/rfc4960

[116] "The Secure Real-time Transport Protocol (SRTP)," 2004. [Online]. Available: https://www.ietf.org/rfc/rfc3711.txt

[117] PeerJS, "PeerJS." [Online]. Available: http://peerjs.com/

[118] "JSEP." [Online]. Available: https://tools.ietf.org/html/draft-ietf-rtcweb-jsep-24

[119] NPM, "Node Package Manager." [Online]. Available: https://www.npmjs.com/

[120] "Electron." [Online]. Available: http://electron.atom.io

[121] "Media Source Extensions." [Online]. Available: https://w3c.github.io/media-source

[122] "Chromium." [Online]. Available: https://www.chromium.org

[123] "WebM," 2014. [Online]. Available: http://www.webmproject.org/

[124] N. Modadugu and E. Rescorla, "The Design and Implementation of Datagram TLS," *Proceedings of ISOC NDSS*, p. 14, 2004. [Online]. Available: https://crypto.stanford.edu/ nagendra/papers/dtls.pdf

[125] A. Rowstron and P. Druschel, "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems," *Middleware 2001*, vol. 2218, no. November 2001, pp. 329–350, 2001. [Online]. Available: http://www.springerlink.com/index/10.1007/3-540-45518-3

[126] R. Ferreira, S. Jagannathan, and A. Grama, "Locality in structured peer-to-peer networks," *Journal of Parallel and Distributed Computing*, vol. 66, no. 2, pp. 257–273, 2006.

[127] "NodeJS Crypto Library." [Online]. Available: https://nodejs.org/api/crypto.html

[128] R. Roverso, S. El-Ansary, and S. Haridi, "Peer2View: A peer-to-peer HTTP-live streaming platform," *2012 IEEE 12th International Conference on Peer-to-Peer Computing, P2P 2012*, pp. 65–66, 2012.

[129] M. Wichtlhuber, J. Rückert, D. Stingl, M. Schulz, and D. Hausheer, "Energy-efficient mobile P2P video streaming," *2012 IEEE 12th International Conference on Peer-to-Peer Computing, P2P 2012*, pp. 63–64, 2012.

[130] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," *Conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '01)*, pp. 149–160, 2001. [Online]. Available: http://portal.acm.org/citation.cfm?doid=383059.383071

[131] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-Addressable Network," *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, vol. 31, no. 4, pp. 161–172, 2001. [Online]. Available: http://portal.acm.org/citation.cfm?doid=964723.383072