UNIVERSIDADE FEDERAL DE CAMPINA GRANDE

CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA

UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

LUCAS MEDEIROS CAVALCANTE

ENHANCING SECURITY IN CLOUD-BASED STORAGE SERVICES USING EXTENSIBLE TRANSPARENT PROXIES

CAMPINA GRANDE

2020

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Enhancing Security in Cloud-Based Storage Services Using Extensible Transparent Proxies

Lucas Medeiros Cavalcante

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Sistemas Distribuídos e Computação em Nuvem

Andrey Elísio Monteiro Brito
(Orientador)

Campina Grande, Paraíba, Brasil

C376e Cavalcante, Lucas Medeiros.

Enhancing security in cloud-based storage services using extensible transparent proxies / Lucas Medeiros Cavalcante. - Campina Grande, 2020.

54 f. : il. Color.

Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2020.

"Orientação: Prof. Dr. Andrey Elísio Monteiro Brito".

Referências.

1. Computação na Nuvem. 2. Computação Confidencial. 3. Privacidade. 4. Intel SGX. 5. PROXY. Provisionamento de Recursos Intel SGX. I. Brito, Andrey Elísio Monteiro. II. Título.

CDU 004.056.53(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA ITAPUANA SOARES DIAS CRB-15/93

ENHANCING SECURITY IN CLOUD-BASED STORAGE SERVICES USING EXTENSIBLE TRANSPARENT PROXIES

LUCAS MEDEIROS CAVALCANTE

DISSERTAÇÃO APROVADA EM 28/10/2020

ANDREY ELÍSIO MONTEIRO BRITO, Dr., UFCG
Orientador(a)

REINALDO CÉZAR DE MORAIS GOMES, Dr., UFCG
Examinador(a)

EDUARDO DE LUCENA FALCÃO, Dr., UNIFACISA
Examinador(a)

CAMPINA GRANDE - PB

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
POS-GRADUACAO CIENCIAS DA COMPUTACAO
Rua Aprigio Veloso, 882, - Bairro Universitario, Campina Grande/PB, CEP 58429-900

FOLHA DE ASSINATURA PARA TESES E DISSERTAÇÕES

LUCAS MEDEIROS CAVALCANTE

ENHANCING SECURITY IN CLOUD-BASED STORAGE SERVICES USING EXTENSIBLE TRANSPARENT PROXIES

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação como pré-requisito para obtenção do título de Mestre em Ciência da Computação

Aprovada em: 28/10/2020

Prof. Dr. ANDREY ELÍSIO MONTEIRO BRITO, UFCG, Orientador

Prof. Dr. REINALDO CÉZAR DE MORAIS GOMES, UFCG, Examinador Interno

Prof. Dr. EDUARDO DE LUCENA FALCÃO, UFRN, Examinador Externo

Documento assinado eletronicamente por **REINALDO CEZAR DE MORAIS GOMES**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 12/01/2022, às 16:29, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **ANDREY ELISIO MONTEIRO BRITO**, **PROFESSOR 3 GRAU**, em 13/01/2022, às 10:51, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **Eduardo de Lucena Falcão**, **Usuário Externo**, em 13/01/2022, às 15:15, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da

Portaria SEI nº 002, de 25 de outubro de 2018.

A autenticidade deste documento pode ser conferida no site https://sei.ufcg.edu.br/autenticidade, informando o código verificador **2065170** e o código CRC **960666B1**.

Resumo

A computação moderna está mudando para a nuvem, visto que este é o melhor ambiente para atender às necessidades do estilo de vida com compartilhamento massivo de dados, que surgiu após a criação dos serviço de redes sociais e, mais recentemente, o aumento esmagador do número de computadores pessoais na forma de smartphones. Como consequência, existe uma preocupação cada vez maior com a segurança dos dados na nuvem, principalmente quando se considera o uso de provedores públicos. No entanto, a maioria dos vazamentos que acontecem no armazenamento em nuvem acontece devido a erros humanos cometidos na configuração de acesso ou na manipulação de chaves. A segurança também está em risco quando as ferramentas de análise de dados são comumente utilizadas para inferir tendências em relação a indivíduos.

A comunidade de pesquisa em segurança da informação é muito ativa, e abordagens simples, usando criptografia, técnicas de anonimização, e até mesmo privacidade diferencial, têm sido usadas em conjunto com Ambientes de Execução Confiável. Estes constituem a nova solução de hardware para trazer confiança aos ambientes em nuvem, prometendo garantias de integridade de dados e código e confidencialidade até para modelos mais estritos de ameaça. Entretanto, tais soluções são desenvolvidas para cenários específicos e para atingir confidencialidade e privacidade, por exemplo, seria necessário encadear várias soluções, resultando em maiores penalizações em performance. Além disso, essas soluções são muito específicas e não podem ser usadas com as ferramentas e aplicativos mais comumente utilizadas por usuários finais. Neste trabalho propomos uma arquitetura para um proxy seguro e transparente, capaz de executar diversos pipelines customizados, beneficiando-se, assim, dos algoritmos desenvolvidos pela comunidade em uma única solução. No contexto desta pesquisa, usamos SCONE em conjunto com Intel SGX, o que simplifica a construção de aplicativos confidenciais.

Avaliamos nossa solução em relação a aplicativos front-end bem conhecidos, como o sistema de publicar/assinar Apache Kafka e a ferramenta de Business Intelligence Metabase, conectada a duas das soluções de armazenamento mais usadas em ambientes de nuvem: o banco de dados NoSQL MongoDB e a solução para armazenamento de objetos MinIO (compatível com Amazon S3).

Abstract

Modern computing is moving to the cloud as it is the best environment to meet the needs of the data-heavy lifestyle that emerged after the creation of social networking services and more recently the overwhelming increase in the number of personal computers in the form of smartphones. As a consequence, there is an ever-growing concern about data security in the cloud, especially when dealing with public providers. However, most leaks that happen in cloud storage are due to human error when configuring access or handling keys. Security is also at risk when data analysis tools are so commonly used to infer trends regarding individuals.

The information security research community is very active and approaches, using cryptography, anonymization techniques and even differential privacy have been used together with Trusted Execution Environments (TEE). This novel hardware solution is responsible for bringing trust to cloud environments, promising guarantees of data and code integrity and confidentiality even in strict threat models. However, such solutions developed to specific scenarios are not always ideal and to achieve confidentiality and privacy, for example, one would need to chain multiple solutions, accumulating more overhead. Moreover, these solutions fail to be usable with the most common tools and applications used by end-users. In this work, we propose an architecture for a secure, transparent proxy, able to run many custom pipelines, thus benefiting from the algorithms developed by the research community in a single solution. In the context of this research, we use SCONE in addition to Intel SGX, which simplifies building confidential applications.

We evaluate our solution against well-known front-end applications such as the publish/subscribe system Apache Kafka and the Business Intelligence tool Metabase connected to two of the most used storage solutions running in Cloud environments, the NoSQL database MongoDB and the Amazon S3 compatible object storage solution MinIO.

Agradecimentos

Durante esses dois anos recebi o suporte de diversos amigos e familiares, dentre eles gostaria de destacar a contribuição:

- Dos meus primeiros mentores, Georges, Vera, Geraldo, Nina, Getúlio, Magna, Leudo e Sirley, e seus alunos Letícia, Victor, Vinícius, Laís e Rodrigo, que foram essenciais na minha formação humana.
- Dos professores que participaram da minha formação acadêmica, em especial o meu orientador Andrey Brito, que me acolheu no LSD e desempenhou os papéis de professor, gerente, orientador, guru e conselheiro.
- Dos vários parceiros que fizeram parte direta e indiretamente dessa jornada, com dicussões de valor como, Kaio, Clenimar, Vinha, Fabio, Benardi, Maysa, Lilia, Pablo, Isla, Hugo, Fellype, Caieu, Rodrigo, Bruno, e varios outros (vocês sabem quem são :).
- Da única pessoa que consegue me aturar *quase* o tempo todo, Alice Pimentel e também o suporte da sua família.
- Da eficiente equipe de suporte do LSD que sempre me ofereceu bastante ajuda.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

From the creation of websites that emphasise user-generated content and ease of use that resulted in the social networking services at the beginning of the century, to the overwhelming amount of personal computers that can reach more than 1 per inhabitant in some countries[1] in the form of smartphones shapes modern computing around the demands of high availability, instant scalability, green computing and quick deployments. With these requirements, the best environment for the back-end to applications is cloud infrastructure. Hence 83% of all corporation workloads will run in a cloud by 2020, 41% of these in public clouds [13].

With this massive amount of end-user data, stored in remote "easy" to access locations, cloud applications are a perfect target for adversaries trying to obtain access to confidential user data. Credit card information, social security numbers, medical data and even geolocation data can be dangerous to users, exposing them to thefts and stalking. Therefore, there is a growing concern about data security in cloud environments as there is an increasing number of data leaks, especially the ones that happened due to user errors or negligence. For instance, the lack of care in the configuration of Amazon Simple Storage Service (S3)[2] leads many companies such as CHS Consulting and Verizon to leak personal information such as passport scans, tax documents, emails, contracts, job applications, and background checks related to thousands of individuals [64]. Furthermore, some confidential data should never be stored in public clouds as these environments are not trustworthy and are susceptible to collusion attacks from its employees.

[1] https://data.worldbank.org/indicator/IT.CEL.SETS.P2

[2] http://aws.amazon.com/s3/

Another factor that affects data security is the usage of Data mining tools to infer trends and patterns, especially regarding individuals. Some information must be restricted to protect individual privacy. The increasing concern with personal data protection leads to the creation of laws imposing data protection and accountability such as the European General Data Protection Regulation (GDPR) [18] and the Brazilian "*Lei Geral de Proteção de Dados*" LGPD [19].

The GDPR came to update regulations created in the '90s when people were not sharing private information for free every minute, and it is called *General* because it must provide protection and rights to all individuals from the member countries of the European Union (EU). It outlines reputational damage and hefty fines for organisations that breach its rules and mishandles user information. The regulation is extensive and contains 99 individual articles that focus on protecting *personal data*, meaning information that allows a living person to be directly, or indirectly, identified from data that is available, which means that not only to names or addresses but also to less apparent individual data like IP addresses and cookie identifiers [18].

Although coming from the EU, GDPR can also apply to businesses based outside the region. If a business in the US, for instance, does business in the EU, then GDPR can be applied as it is a controller of EU citizens. GDPR is considered a progressive approach to how people's personally identifiable data should be handled and lead the creation of regulations across other regions, for instance, the LGPD in Brazil.

1.2 Problem Statement

Security, as defined in the Merriam-Webster, is "the quality or state of being secure—to be free from danger."[3]. A secure system should have multiple layers of security to protect its operations such as Physical Security, Operations Security, Network Security and **Information Security** [61].

This work mainly focuses on increasing **Information Security** and protecting data from dangers such as disclosure, destruction and modification.

Confidentiality is one of the basic elements of information security. Confidential information means protection from disclosure or exposure to unauthorized individuals or systems. Confidentiality guarantees exclusive access to information to a group of parties with special rights and privileges. To ensure confidentiality, one need to use all the techniques designed for security like access control, encryption, authentication and defence against penetration attacks [61].

Securing data from unauthorized modification, corruption, or other disruptive actions against its

[3]Definition for security: https://www.merriam-webster.com/dictionary/security

original state ensures **integrity** and is the cornerstone of data systems as data has no value if users cannot verify its probity. Ensuring confidentiality may help protect data integrity as potential attackers cannot access the data, although authorized parties can disrupt data; additionally, other techniques such as file hashing can detect if data was compromised.

Availability is another fundamental element in information security. It is vital to make sure that data access is always available to authorized personnel. Availability in information security means matching network and computing resources to compute data access and implement a better policy for disaster recovery purposes [61].

While confidentiality is an all-or-nothing concept, **Privacy** can be seen as the capacity to control the amount of information disclosed. Thus, a system considered privacy-friendly, it enables users to selectively control how much information it shares with other users or systems. When privacy does not imply confidentiality, it means that accuracy decreased due to some information contained in a piece of data being reduced, but not wholly obfuscated.

Production-ready information systems such as SQL/NoSQL databases and object storage uses some techniques that can guarantee some level of confidentiality and integrity. However, these techniques sometimes do not consider security in a cloud environment, and further increasing protection or tweaking it to a particular use case is discouraged as it involves changes to source code. Furthermore ensuring privacy can be difficult as different data sets have different needs.

A solution to mitigate these problems is to add a **intermediate** layer of protection, between the front-end application and back-end cloud storage, that enforces custom rules to obtain data access. This way, in order to provide access to someone, an organization must first decide which additional permissions should be granted to a user (e.g., a data analyst) beforehand based on intent. This extends the default application permissions, thus reducing data leakage from these sources due to misconfiguration. Furthermore, to enable multiple processes to be applied that increases data protection of individuals, the additional layer can act as a **Pipeline Engine**, ensuring online examination of records and an extra level of personal data protection by running custom user-defined privacy and anonymization algorithms.

1.3 Contributions

This work contributes to the current knowledge on designing transparent proxies for increasing security in the following ways:

1. Identifying what the research community is using to increase confidentiality and ensure indi-

vidual data protection;

2. Proposing an architecture that enables current and upcoming solutions to be applied, in sequence, thus reducing latency and offering flexibility (when compared to similar researches);
3. Testing the proposed solution against different front-end and back-end applications that are commonly used by companies;
4. Showing the performance overhead of the proposed solution, identifying possible bottlenecks;
5. Outlining improvements to the solution aiming for production usage of the secure proxy;

1.4 Requirements

This work proposes an intermediate system that can enable information security enforcement increasing confidentiality and integrity while keeping both ends (client/target) unaware of such changes. We, therefore, declare the following requirements for the proposed system:

- **Transparency**. Data scientists and application developers use many different tools to meet their need; the most used programs already support connections with many different back-end storage solutions. The proposed solution must be transparent to both front-end clients and back-end servers to be used with well-known applications;
- **Confidentiality**. Protection against data disclosure is one of the most important requirements of any secure application. The proposed solution must define its threat model and describe how it is safe from malicious individuals;
- **Privacy**. New regulations are being created focused on protecting personal data. The solution proposed must offer an engine that helps data owners protect individual privacy;
- **Generalization**. The research community is always developing novel techniques that increase security. The proposed solution must have an extensible pipeline engine to adapt to new developments in Information Security;
- **Usability**. Data owners must protect the data even if the measures taken slows down the analysis process. However, we understand that performance is an issue when evaluating the usability of a particular software; therefore the proposed solution must have a somewhat unnoticeable performance overhead to a client

1.5 Threat Model

We assume that cloud providers and client users are untrustworthy: although they are conscious of their reputation and knowledgeable of the implications of data leaks they may not take the necessary precautions to keep data integrity, therefore misconfigurations, hardware failures, software bugs, malpractice and even negligence can all result in data leaks [4]. Under this threat model, we define client users and cloud providers as "Imperfect and Selfish", meaning that they are susceptible to accidental mistakes and selfish about revealing leaks as it could damage their reputation [47]. As they have full control of the machines running the back-end, data owners can only protect their data by encrypting it before it reaches the server; thus, data-in-transit must also be protected.

The proxy runs on a trustworthy machine as it runs on a secure SGX enclave, so we inherit the SGX threat model: Attacker have superuser privileges on the infrastructure where the applications run, can control the entire software stack, including, but not limited to, operating systems, hypervisors, and cloud computing platform software. In Summary, adversaries have read and write access to any memory region except the enclave memory (EPC). Adversaries performing side-channel attacks are out of the scope of this solution, although there are some mitigations for side-channel using SGX [41]. The proxy secures data storage and also individual privacy using security pipelines defined by the data owner, a security expert that understands the privacy risks and uses optimal strategies developed by the research community.

1.6 Chapter Overview

Chapter 2 contains the background knowledge necessary for a better comprehension of this work, including concepts, technologies and also information on some well-known applications. Chapter 3 describes the related works to this research. Chapter 4 describes the solution, including the architecture and how we enforce security. In Chapter 5, we test the solution against two different common scenarios, one more concerned with confidentiality while the other focuses on privacy. Chapter 6 presents the final remarks, including discussion on possible improvements as well as other achievements.

Chapter 2

Background

2.1 Privacy

The Greek philosophers already had a perception over the division of information as public or private. Ideas of ownership and possession were already present on ancient cultures; the concept of what belongs to an individual, family, or even to a community [5]. Socrates division of students in an initial "inner" circle that engages in a discussion over a theme, therefore generating data, all that while an "outer" circle of students is limited to observe the discussion analyzing the performance of the inner circle is in direct correlation with the modern work performed by data analysts when they interpret statistical information over collected data to obtain business value over it. [46]

The discussion over privacy increased after the many scientific breakthroughs brought by the Second World War, and in 1962, Alan Westin published the book "Privacy and Freedom", a result of his research over the impact of science and technology on privacy [17; 7]. Westin defines privacy as "the claim of individuals, groups, or institutions to determine for themselves when, how, and to what extent information about them is communicated to others. Privacy is the voluntary and temporary withdrawal of a person from the general society through physical or psychological means, either in a state of solitude or small-group intimacy or, when among larger groups, in a condition of anonymity or reserve" [60].

The value of privacy as a need for personal autonomy, a human right, outgrows the profit opportunities of the data industry. Therefore, ensuring personal data protection is a must when dealing with information security.

In this work, we implemented strategies from two lines of work studied by the privacy protection research community: Anonymization and Differential Privacy.

2.1.1 Anonymization

Data shared among different parties must be anonymized as proper disclosure of information keeps individual integrity and also continues to have business value. However, sometimes data looks anonymous when, in fact, it is not. Data owners, including government agencies, usually remove all explicit identifiers, such as name, addresses, Social Security Numbers, so that the information can be indiscriminately shared, thus incorrectly assuming that these identifiers cannot be determined. However, this technique known as de-identifying provides no guarantee of anonymity [58]. The released data often contains some public attributes such as birth date, gender and ZIP codes. These attributes are also known as *quasi-identifiers* and can be linked to other publicly available databases to re-identify individuals. This problem was demonstrated when a combination of data from IMDB (Internet Movie DataBase) and Netflix resulted in revealing private user information [38].

Samarati and Sweeney (1998) proposed a formal foundation for achieving anonymity while ensuring protection against linking attacks and introduced *k-anonymity* as a characterization of the degree of data protection against this kind of attack [52]. k-anonymity can be achieved by means of generalization and suppression of disclosed data, and an algorithm can be used to compute the minimal generalization of a given table. The k-anonymity requirement is that "Each release of data must be such that every combination of values of quasi-identifiers can be indistinctly matched to at least k individuals". Therefore k-anonymity demands that we generalize individual rows of a dataset into a group of at least k rows and replace the quasi-identifiers of these rows with aggregate (or even suppressed) quantities, such that it is no longer possible to read the individual values. This characterization protects people by ensuring that an adversary who knows all quasi-identifiers of a person can only find out which group a person might belong to but not know if the person is indeed in the dataset.

In this work, our proposed proxy solution implements a pipeline using the *Mondrian* algorithm for achieving k-anonymity in a dataset [33], which uses a greedy search algorithm to partition the original data into smaller and smaller groups. After obtaining the partitions, the values of the quasi-identifiers and the sensitive attributes in each k-anonymous group are aggregated. Examples of aggregations performed are, truncation of decimal values and replacing numerical attributes with a range (e.g. "age: 24-28") [33].

2.1.2 Differential Privacy

The emergence of **differential privacy** as a formal definition of privacy related to providing robust privacy assurance to individuals based on adding noise to answers to queries on the data, has revolutionized the field of privacy-preserving data mining [12]. Differential privacy allows general statistical analysis to be performed in datasets, whilst private information of specific individuals is protected. It differs from anonymization techniques that are more focused on data publishing using generalization, while differential privacy uses noise injection and control to allow data mining [21].

Differential privacy algorithms defend against adversaries by adding random noise to the data; therefore, an adversary is only able to receive inaccurate data that is unable, to an extent to obtain information about an individual. Even when knowing that data from a singular person *is* in the dataset it is very hard or impossible to know which data corresponds to whom [20]. In one hand, incorporating too much noise can lead to unusable data, on the other hand with every new query more sensitive information is leaked, thus determining the maximum privacy loss is crucial so that meaningful privacy is guaranteed, this is called the *privacy budget*.

In this work, to evaluate how a secure proxy can offer a privacy protection layer, we implemented a simple algorithm that ensures differential privacy called Randomized Responses Technique [20] (RTT). Using RTT participants, before answering to a sensitive question like “Have you ever committed a crime?”, must first flip a coin, if it lands on heads then the participant must answer truthfully, if it lands on tails, then the participant throws the coin again and answer “Yes” if it is the result is head or answer “No” if it lands on tails. This technique allows analysts to make statistical deductions on data because they know the probability of the coin flip. Also, notice that participants have refutability on individual responses, even if the data leaks they always can say that their response was randomized, this also enforces confidentiality [9].

2.2 Trusted Execution Environment

Trusted Execution Environment or TEE is a hardware solution designed to bring security to untrusted platforms by creating an isolated processing environment in which applications can be securely executed, safe from interferences from rest of the system [51].

Different hardware companies have their implementation of Trusted Execution Environments solutions. Intel uses secure memory enclaves with Intel SGX, AMD increases the security of virtualized environments with Secure Encrypted Virtualization, and similar to SGX, ARM also uses a certain region to enforce security in the form of ARM TrustZone [45; 30; 14]. In this work, we use the Intel

Software Guard Extensions (SGX) as a TEE requirement for untrusted environments with the help of SCONE (Secure CONtainer Environment) to facilitate the usage of SGX enclaves.

2.2.1 Intel SGX

Intel Software Guard Extensions (SGX) are a set of CPU instructions that enable the creation of protected memory regions, called enclaves [15]. The access to these regions is shielded even from processes with root privileges. Developers build SGX-based secure applications by splitting the architecture into two parts: An untrusted part that serves as the interface to the program; and the trusted enclaves. The communication between the trusted and untrusted parts is managed by the SGX driver, which is itself an untrusted kernel module. The two parts exchange data across well-defined call gates. The code loaded in the enclave is signed so as to verify that the code being executed is indeed a genuine application from the developer. The enclave code can be attested locally (on the same machine) or remotely (on a different machine), making it possible to build secure distributed applications. When dealing with an adversary that is in control of the hypervisor and the operating system, deployed software can be susceptible to side-channel attacks, SGX applications are no exceptions to these attacks, thus affecting both target backends and our proposed proxy solution. However, some recent works help to mitigate these kinds of attacks: Varys fully protects against all L1/L2 cache timing attacks and mitigates page table side-channel attacks [41].

2.2.2 SCONE

SCONE (Secure CONtainer Environment) is a software solution that facilitates the execution of secure applications by leveraging SGX enclaves [3]. SCONE is based on a modified version of the *libc musl* library, thus allowing applications to be developed without the modification of the source code that would be necessary when using the Intel SGX SDK.

In contrast with the SGX SDK that limits the support to C and C++, SCONE can be utilized by other languages that use libc such as Python, GO, Fortran and RUST. The solution proposed in this work was implemented in RUST, as it provides an ownership feature that makes memory safety guarantees without needing a garbage collector[1]. This work could be done in any language supported by SCONE, we choose RUST for comfort reasons. SCONE relies on the container environment and is limited to Linux machines with support for Intel SGX.

[1] https://medium.com/the-innovation/how-microsoft-is-adopting-rust-e0f8816566ba

Scone applications can be attested remotely via CAS. CAS stands for *Configuration and Attestation Service* and is able to verify the identity of applications and also works as a secret storage and provisioning solution, able to inject environment variables, generate keys and provide environment variables at runtime.

SCONE also features a file encryption mechanism, called FSPF, that supports transparent (meaning that no application code changes are needed to support this) encryption and authentication of files of the running container root filesystem [8].

Chapter 3

Related Work

Our solution proposes using a transparent proxy running inside a trusted execution environment as a means of ensuring confidentiality and integrity to general-purpose applications. We compare our work with solutions also using Intel SGX as a means of creating trustworthy software, as shown in Section 3.1. The concern with cloud security did not start only after the creation of TEEs; therefore, we also compare our work with non-TEE solutions in Section 3.2

3.1 TEE Solutions

The concept of TEEs revolves around bringing trust to cloud environments, thus enabling developers to offer a layer of protection to existing solutions. We noticed that this layer of protection could exist in the server-side or in-between client and server in the form of a proxy. Therefore, we compare our work with other proxy-based solutions on Section 3.1.1 and server-side layers on Section 3.1.2.

3.1.1 Transparent Proxy

Adding a transparent layer of protection to existing applications is not a novel solution for security in cloud environments. There have been several approaches to use TEEs for establishing secure proxies.

Solutions like Troxy and Bloxy integrates Byzantine Fault Tolerance (BFT) solutions to heterogeneous systems such as HTTP clients and blockchains infrastructure [34; 50]. Similar to our approach, they use SGX to leverage a new functionality, a consensus protocol, thus increasing integrity and availability. Both solutions are built on top of Hyster [6], a hybrid BFT system, and differ mainly on the use case as Bloxy is tailored for the blockchain use case.

Another solution that is also concerned with integrity is the audit library LibSEAL [4]. With the

help of Intel SGX, LibSEAL generates an audit log based on client request and service responses, acting as a termination endpoint for TLS (transport layer security) connections, a service-specific relational audit log is created and is persisted in an embedded database running inside the TEE. LibSEAL checks the audit logs for invariants to discover integrity violations and can be used as proof of SLA (Service Level Agreement) violations for cloud storage solutions such as Dropbox or Google Drive.

Silva proposes a solution to increase Apache Kafka security that utilizes a proxy component called "Message Interceptor" responsible for filtering messages sent by publishers and received by subscribers, and to further increasing Kafka confidentiality, and integrity messages are transparently encrypted before reaching the server, messages are also decrypted before reaching the consumers [56]. Silva also evaluates the difference in performance brought by his proxy in a non-TEE scenario, using the Intel SGX SDK and using Scone.

The solutions mentioned above, as other similar solutions we found in the literature, do not address all security issues we aim to solve. Furthermore, adding multiple layers of proxies leads to an avoidable overhead in performance. In our work, we try to provide a custom generic proxy that relies on stacking user-defined pipelines. This way, one could have, attached to each cloud application, a pipeline setup that increases privacy then it creates an audit log and finally it encrypts all the data before sending it to a not trusted environment, using a single proxy, thus reducing latency and performance overhead.

3.1.2 Securing Server-side

Another strategy when dealing with not trustworthy environments is to use a secure back-end that can guarantee confidentiality and integrity even when used in public clouds.

SGX-FS [10] combines the FUSE (filesystem in userspace) [59] framework with an SGX module, thus benefiting from trusted execution environments security enforcement to create a secure userspace filesystem. SGX-FS uses a sequence of SGX callbacks for each file system operation (i.e., read, write, open, etc.), unencrypted blocks are kept inside the enclave and, given the size limit, if they become too large are encrypted and dumped in the local hard drive.

At a higher level of secure alternatives for data storage PESOS [31] describes an implementation of an object storage solution for untrusted third-party storage providers that runs inside an enclave and uses Kinetic Open Storage for trusted storage. PESOS also implements a logic of per-object security policy specification at a controller level, separate from the storage stack. The PESOS controller mediates IO operations to ensure that the policies are respected.

Another high-level back-end for secure storage is EnclaveDB, a database engine that also utilizes Intel SGX to store sensitive data such as tables, indexes and metadata [48]. EnclaveDB is an in-memory database, and it relies on using an ahead-of-time compiler on a trusted machine to compile queries on sensitive data to native code. The pre-compiled queries are sent to an untrusted machine running the EnclaveDB executor inside an enclave, that authenticates the query request, runs the query, encrypts the result and sends it back to the trusted client.

The above solutions try to leverage security at the server-side; therefore, they can be much more efficient than a proxy-based solution, as there is no latency overhead, and there is more room for performance optimization. However, the majority of client-side consumer applications, for example, Business Intelligence tools such as Microsoft PowerBI[27], are designed to work with well-established databases (PostgreSQL[1], MySQL[2]), therefore if one would want to use EnclaveDB it would be necessary to add a custom plugin to PoweBI. The same can be said about PESOS when we compare it with Amazon S3, as an object storage solution, in terms of overall usability. Having a proxy-based solution is, therefore a much more flexible solution when we want to leverage security for commonly used applications.

3.2 Non-TEE Solutions

Not all solutions rely on using trusted execution environments to build confidential applications. In Section 3.2.1 we discuss works using client-side encryption to ensure confidentiality in cloud storage and Section 3.2.2 discusses secure processing of sensitive data.

3.2.1 Data Storage

For cloud **data storage**, using a trusted computer for encryption before sending data to the cloud can ensure data confidentiality. MetaCloudDataStorage proposes an architecture for efficient encryption to protect big data in cloud environments. MetaCloudDataStorage [36] runs in a trusted environment and partitions the data, encrypts and finally stores it in many different data providers, therefore reducing the amount of data encrypted at once and also increasing data protection against malicious cloud administrators.

Zardari et al. proposes using a K-NN algorithm to classify data between sensitive and non-sensitive (public), and advocates that only sensitive data should be encrypted and therefore stored in

[1] https://www.postgresql.org

[2] https://www.mysql.com

a different virtual machine than the non-encrypted public data that is stored unencrypted [63]. The authors proceed to show the proposed approach in a simulated environment, however they do not evaluate the difference in computational costs between their solution versus encrypting all the data, also the costs of classifying all the data and decrypting some of it versus decrypting all the data.

It is important to notice that simply encrypting data before storing it in an untrusted environment is not an ideal solution as keys can be lost, mistakenly leaked or extracted by a malicious user. A combination of other authentication methods can be used to further extend confidentiality, such as a two-factor data protection mechanism [62]. The process of encryption and decryption can even be performed in multiple computers. Also, **Having to depend on a trusted environment is by itself a security flaw** Furthermore, such solutions are very centralized and seem to be difficult to scale. Therefore, creating a security proxy able to run in an untrusted cloud environment that offers TEE hardware has an inherent scalability potential.

3.2.2 Data Processing

Public Clouds offers easier access to a wide range of powerful processing and analytical tools [2]. However, these environments are not trustworthy, and many industries such as the healthcare system cannot rely on them to safely compute sensitive data [43].

A solution for this problem is the usage of Homomorphic Encryption (HE), allowing specific computation to be performed on encrypted data, without the need for decryption [25; 11]. One of the first works on privacy-preserving machine learning by Lindell and Pinkas, discuss the applicability of a *secure multiparty computation* (SMC) protocol as a means of computing ID3 (a basic algorithm for constructing decision trees) using homomorphic encryption [35].

Chapter 4

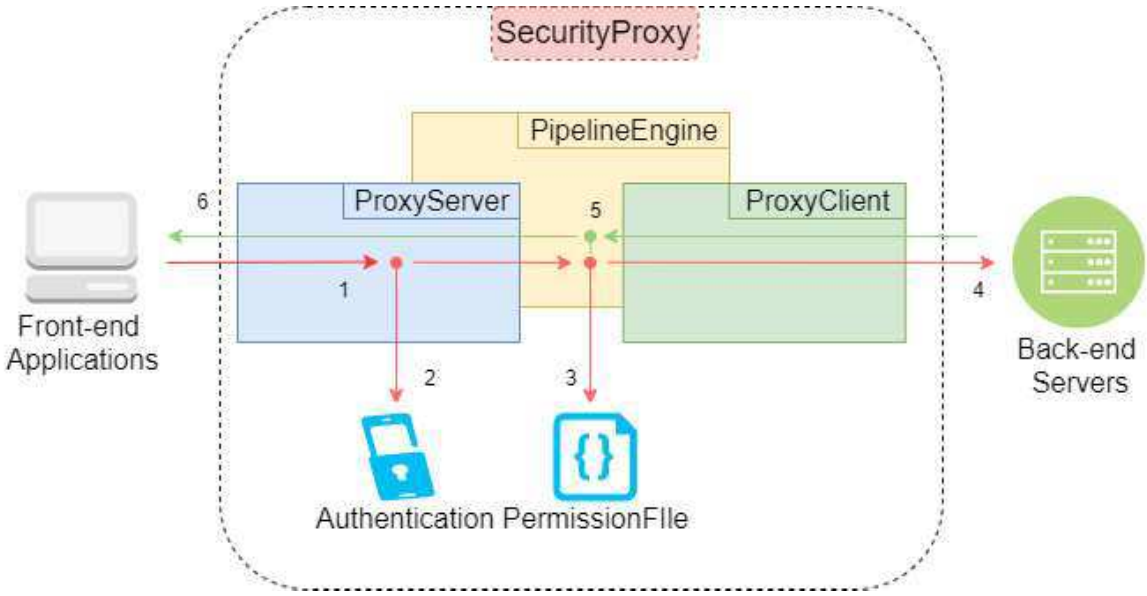
Solution

This chapter describes the components of the proposed solution: the Architecture Overview (Sec. 4.1), consisting in the proxy server components, a model structure for defining wire protocols and how to define permissions for clients; the Pipelines (Sec. 4.4) showing how to integrate custom pipelines enabling different security and privacy preserving algorithms to be applied.

4.1 Architecture

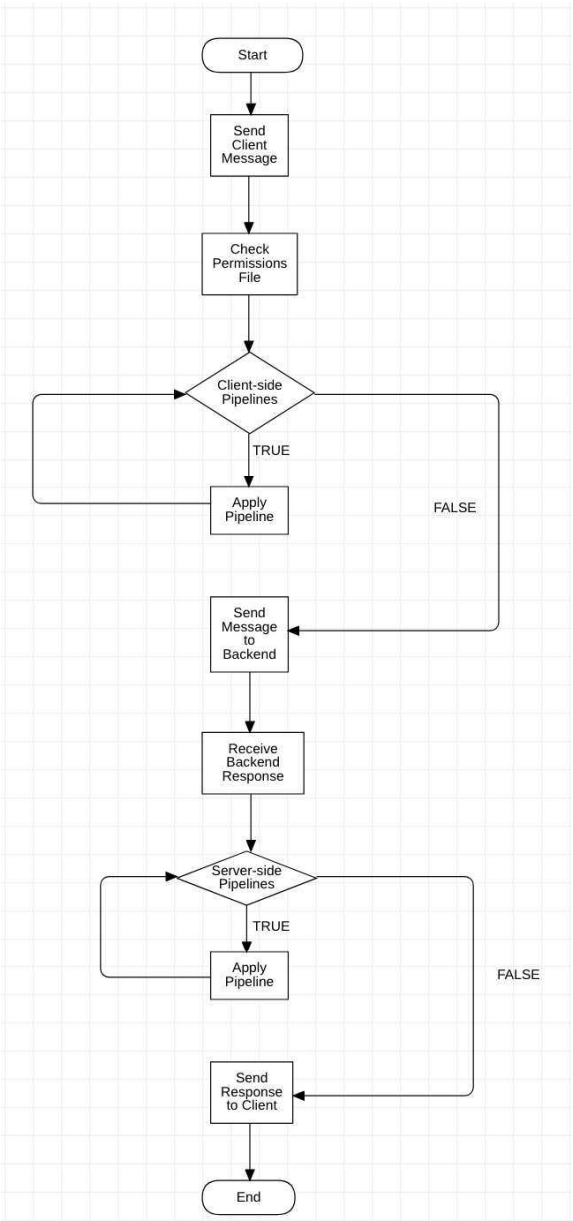
Our proposed solution is an implementation of a transparent proxy server that applies custom pipelines that can be applied at the client or server side. Figure 4.1 describes the architecture proposed in this work, and Figure 4.2 furthers describes the workflow.

Figure 4.1: Secure Proxy Architecture



We consider that Front-End Application users (application developers, data scientists, etc.) are not trustworthy, as they can perform mistakes. The arrow labeled with "1" describes the first step, common to traditional proxies, when the front-end application connects to a proxy that is acting as a server. Step 2 of a secure proxy must be a proxy authentication, different from back-end authentication; mitigating the effect of leaked keys. Step 3 can be considered one of the cores of our solution and is where the security Pipelines are run, following what is described in a Permission file, written by the data owner. In Step 4, the proxy acting as a client communicates with the back-end. Additional pipelines can be run on the back-end response, at Step 5. Finally, at Step 6 the front-end receives the secure message.

Figure 4.2: Flowchart Diagram Proxy



As any transparent proxy should do, our solutions start by receiving client messages, acting as if it is the target server. Then, it checks the stored client permissions file, so it knows which security enforcement will be applied in the form of pipelines. Some security algorithms can be applied directly in this client message, as to protect the data before it reaches the untrusted back-end. An example of a client-side pipeline that increases data confidentiality is an encryption algorithm. Therefore, before sending the message to the real server, client-side pipelines are run.

Sometimes one may want to apply extra layers of security at once, so the proxy should be able to allow pipelines to applied in sequence, for example, to reduce user access to a particular database a

pipeline could check the complexity of a query applied to a database. Another pipeline could change the **SELECT** statement of a query to reduce the spectrum of the projections, and finally, the last pipeline can keep track of the number of queries performed by the client in order to prevent query replay attacks. After the pipeline loop ends the client message is sent to the untrustworthy back-end. If a pipeline fails, the client gets no response. The proxy now is acting as if it is the real client. Notice that being both transparent to the Client and Server is essential for our solution as one of the main objectives is to be a flexible solution that can be used together with the most common software solutions used in cloud environments.

Now the proxy has the opportunity to apply pipelines before sending the response back to the client. These are called server-side pipelines. In opposition to what happens with the client-side pipelines, that enforces security because they do not trust the back-end cloud environment, the server-side pipelines, in general, exist because the user managing the client is untrustworthy. This step is where most of the privacy-preserving techniques should be applied, and algorithms should be able even to erase all the response, if some unavoidable leak potential is detected. Server-side pipelines can also be applied in sequence. The final step is to send the filtered response back to the client, ending the workflow of the proxy.

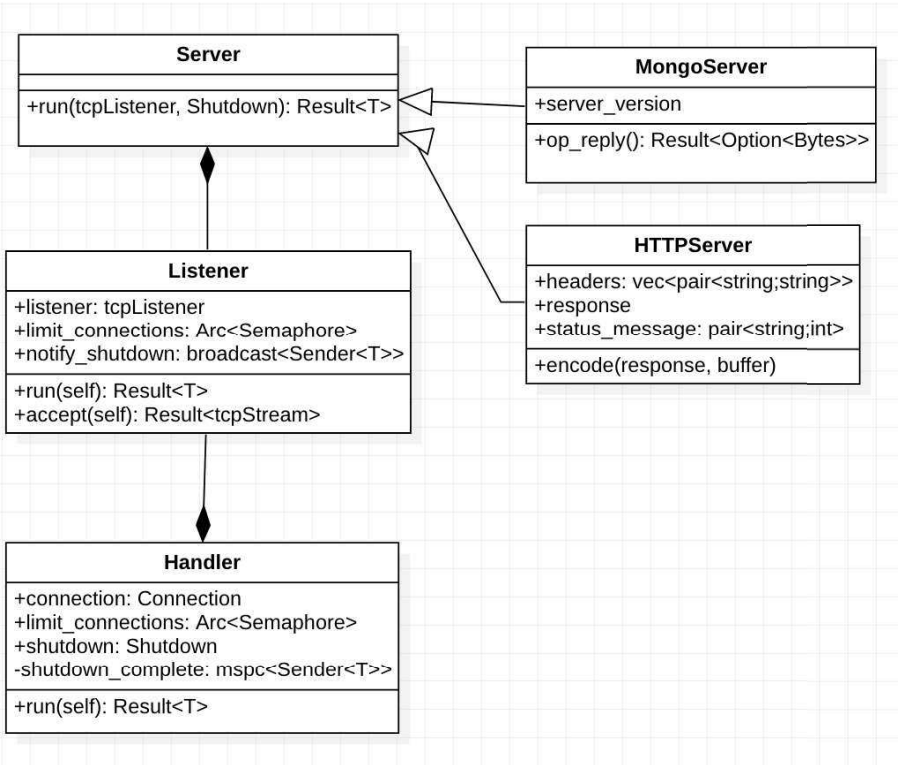
The following subsections will detail each of the core proxy components, showing how they relate to each other.

4.1.1 Proxy-Server

A **Listener** state is created by the **Server** and has a Rust-Tokio [1] TCPListener. The Listener has a *run* method responsible for listening for inbound connections and spawn a task to process each connection. Tokio tasks are like asynchronous green threads and are executed concurrently. The *accept* method is responsible for accepting inbound connections. Errors are handled by backing off and retrying. An exponential backoff strategy is used. After the first failure, the task waits for 1 second. After the second failure, the task waits for 2 seconds. Each subsequent failure doubles the wait time. If accepting fails on the 6th try after waiting for 64 seconds, then this function returns with an error, we decided that this was a fair amount of tries before dropping the connection.

[1] https://tokio.rs

Figure 4.3: Class Diagram Proxy-Server



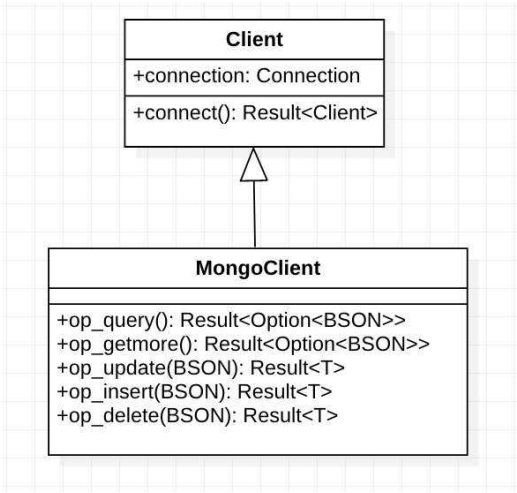
A *Semaphore* is used to limit the max number of connections. Before attempting to accept a new connection, a permit is acquired from the semaphore. If none are available, the Listener waits for one. The Listener is also able to broadcast a shutdown signal to all active connections, in case of proxy termination. The Server *run* caller provides the initial "shutdown" trigger.

For each connection, the Listener has a **Handler** reading requests from its **connection** and applying commands to its buffer. A connection represents a TCP connection decorated with the desired protocol encoder/decoder. This protocol can be an HTTP 1.1 implementation or even represent the MongoDB Wire Protocol. Section 4.1.3 have further details on Connections. The Handler is responsible for returning the permit to the Listener when it completes the processing of a connection so that it can receive more connections.

4.1.2 Proxy-Client

Backed by a single tokio *TcpStream*, **Client** provides basic network client functionality. Connections are established using the *connect* function.

Figure 4.4: Class Diagram Proxy-Client



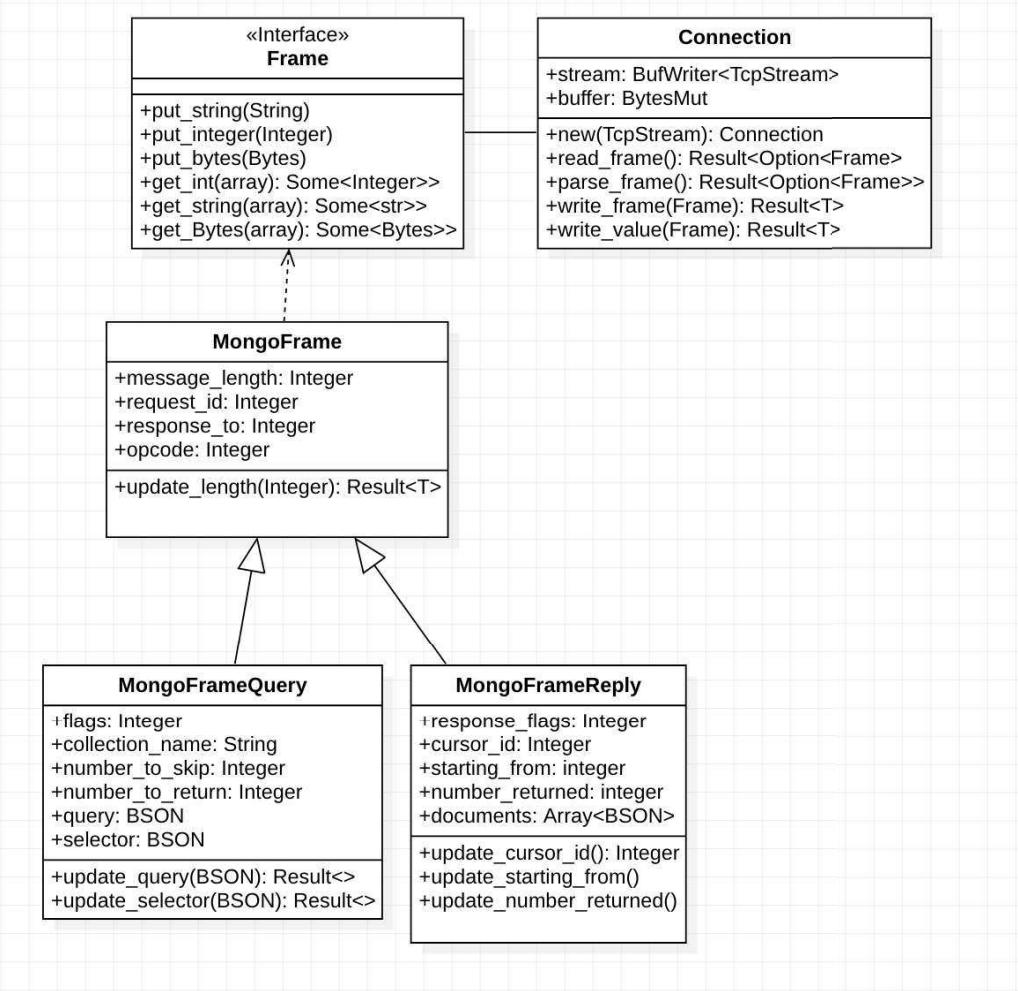
When a **Listener** receives an inbound connection, the *TcpStream* is passed to a new *Connection*, which initializes the associated buffers. The connection allows the handler to operate at the **Frame** level and keep the byte level protocol parsing details encapsulated in **Connection**. The relationship between **Frame** and **Connection** is further explained in subsection 4.1.3.

The TCP connection is decorated with the appropriate protocol encoder/decoder and uses a buffered *TcpStream*. Figure 4.4 describes an example of a MongoDB Client. Observe that *op_query*, *op_getmore op_update*, *op_insert* and *op_delete* are all operations described by the MongoDB Wire Protocol and all these requests are issued using the various methods of the decorated MongoDB **Client**.

4.1.3 Frame and Connection

We decided, in this implementation that protocols must be modeled using an intermediate representation. In this proposed architecture we created the **Frame** abstraction to represent these structures. Frames also provide utilities for parsing frames from a byte array.

Figure 4.5: Class Diagram Frame and Connection



Connection is responsible for sending and receiving **Frame** values from a remote peer. When implementing networking protocols, a message on that protocol is often composed of several smaller messages known as frames. The purpose of **Connection** is to read and write frames on the underlying *TcpStream*.

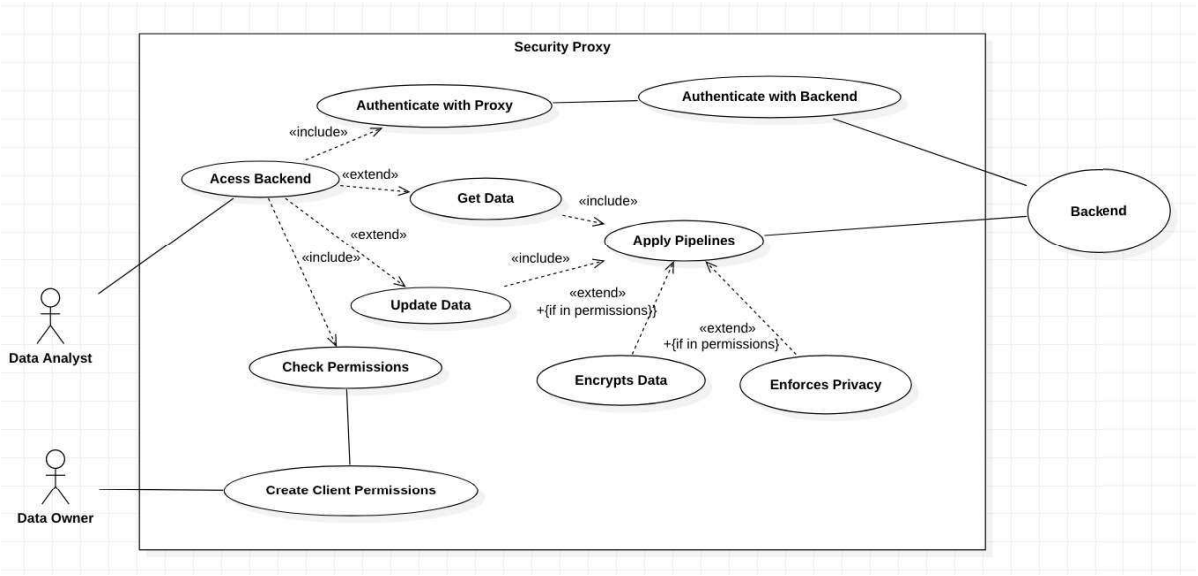
To read frames, the **Connection** uses an internal buffer, which is filled up until there are enough bytes to create a full frame. Once this happens, the **Connection** creates the frame and returns it to the caller. When sending frames, the frame is first encoded into the write buffer. The contents of the write buffer are then written to the socket.

Figure 4.5 describes the relationship between **Frame** and **Connection** as well as depicts an implementation of a MongoDB frame in the form of **MongoFrame**. In MongoDB query and reply frames are quite different so we divided the implementation between **MongoFrameQuery** and **MongoFrameReply**.

4.2 Permissions

As shown in the flowchart diagram (Figure 4.2) before running the pipelines, our proxy solution checks for the permission file of the user behind the client. Figure 4.6 shows an example of how data analysts can use the proxy to access the untrusted target back-end, and data owners can configure said access for each analyst.

Figure 4.6: Use Case Diagram



As shown in figure 4.6 permissions are uploaded by the data owner that wants to protect the data that is stored in an untrusted environment and analyzed by an untrusted user.

Each dataset has its characteristics. Therefore, the one with the best understanding of the data and its characteristics is the data owner. We provide a solution architecture for enabling further security measures to be applied transparently and although some general-purpose solutions for confidentiality, for example, data encryption, can be performed with no knowledge of the data content, some solutions, especially the ones that deal with privacy-preserving techniques, are dependent on knowledge over the data. Therefore, its the responsibility of the data owner to choose the appropriate methods (pipelines) that will meet the security level desired. The proxy is then only considered transparent on the perspective of the client (front-end) users.

Figure 4.7: Permissions File Example

```
{
   "permissions":{
      "collections":[
         {
            "name":"SampleCollections.video_reviews",
            "trusted_tables":[
               "rating",
               "date"
            ],
            "allowed_pipelines":[
               {
                  "kanon_floor":{
                     "k":"3",
                     "tables":"rating"
                  }
               }
            ]
         },
         {
            "name":"SampleCollections.video_movies",
            "trusted_tables":[
               "title",
               "year"
            ]
         },
         {
            "name":"SampleCollections.Sakilla_actors",
            "trusted_tables":[
               "FirstName",
               "LastName"
            ],
            "allowed_pipelines":[
               {
                  "record_supression":[
                     "phone",
                     "address"
                  ]
               }
            ]
         }
      ]
   },
   "user_id":"lucas"
}
```

As shown in figure 4.7 Permissions are described in a JSON file format, and should be defined for each user. The privacy proxy has a **Permission** interface that needs to be implemented for each target backend. In this example, the permission file is for a MongoDB backend and describes the permissions for the *user_id* “lucas”. This user have access to the following MongoDB collections: “SampleCollections.video_reviews”, “SampleCollections.video_movies” and “SampleCollections.Sakilla_actors”. Whenever a query targeting “SampleCollections.video_reviews” (a collection of movie reviews done by users of a movie rental service) is performed, the projection is reduced to only show the columns *rating* and *date* as shown in the field “trusted_tables”. Furthermore, the MongoDB result goes through a anonymization pipeline called *kanon_floor*, with parameters k = 3 and column = “rating”.

4.3 Authentication

Our solution revolves around dealing with the untrustworthy cloud environment as well as the untrustworthy user behind the client.Clients should not be trusted as the many of the Amazon AWS data leaks are due to misuse and misconfiguration [64], thus indication a flaw.

At a daily basis, users are sending by mistake API access secrets to public sources such as GitHub [2]. Therefore, in our proxy solution we decided to implement an authentication layer to the proxy that encapsulates the authentication to the target back-end.

Figure 4.8: Key Derivation Function

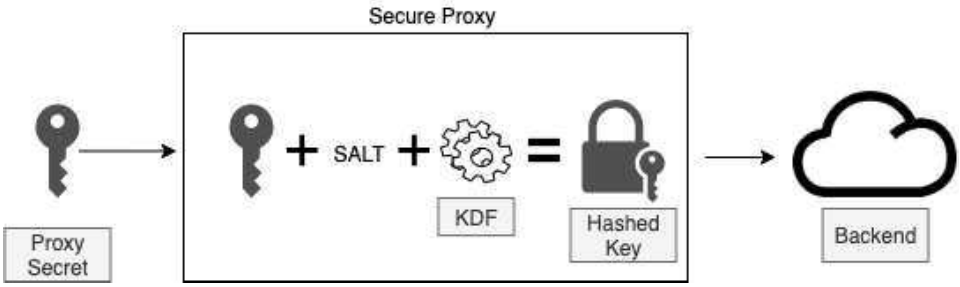


Figure 4.8 shows the process in which a hashed key is created by the proxy to be used as the real authentication secret. A **Key Derivation Function** (KDF), also known as password stretching is a common technique used in cryptographic applications to create hardened passwords based on the number of random bytes used as additional input, commonly called “Salt”.

Using KDF as the authentication method for the proxy further protects the data from mistakes that could be performed by a user. Leaking the *Proxy Secret* only allows a malicious individual to have access to the proxy, protecting the backend. Additionally, to establish a connection to the proxy, a client also needs to perform a certificate-based authentication, as the communication with the proxy is protected by Transport Layer Security (TLS).

4.4 Pipelines

The proposed proxy leverages pipelines as a solution for the generalization of security layers for back-end connection. Different applications have distinct needs for security, therefore it is a must that our solution provides an easy way to extend the pipeline set, and also provides seamless integration with the **Permission** component.

[2]https://github.com

Figure 4.9: Class Diagram Pipelines and Permissions

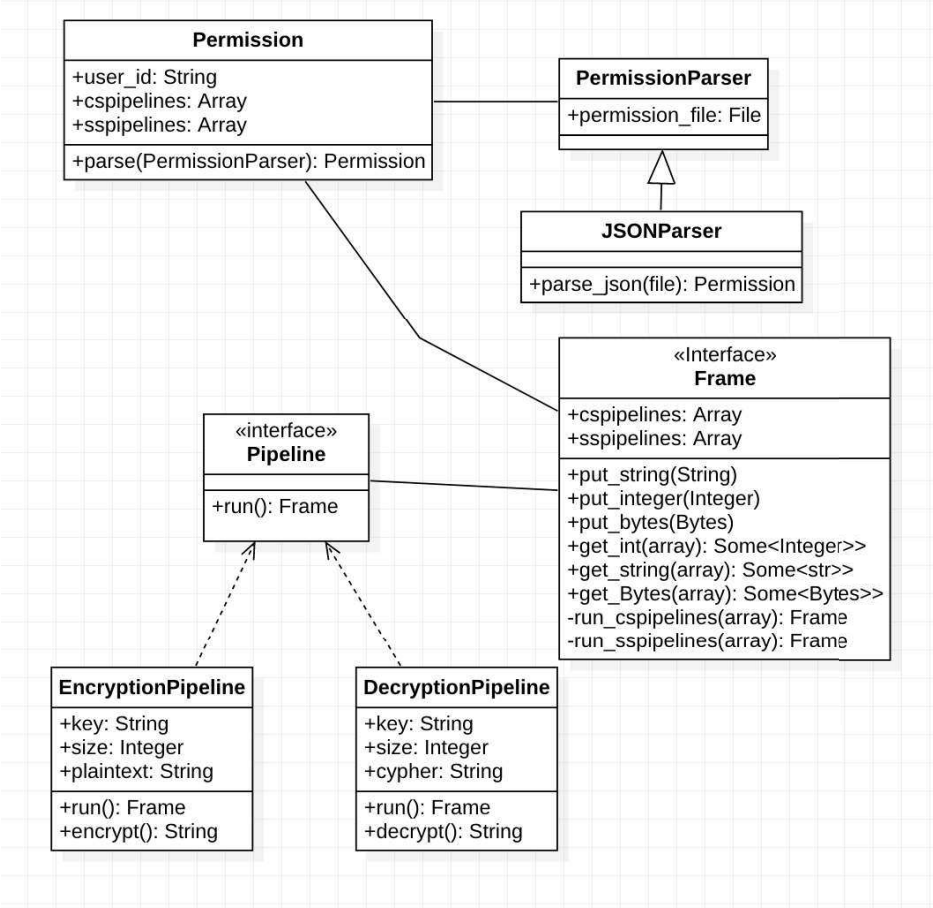


Figure 4.9 depicts a cut of the relationship between **Pipeline**, **Permission** and **Frames**. After parsing the Permission file, the proxy automatically creates an array of Pipelines, both for client-side and server-side Pipelines. Pipelines are seamlessly run by an internal call of *run_cspipelines* and *run_sspipelines*.

Pipelines can be extended by implementing the **Pipeline** interface (trait for rustaceans). In this example, it is possible to see an implementation of a client-side pipeline, in the form of **EncryptionPipeline**, and a server-side pipeline, in the form of **DecryptionPipeline**.

Uploading a Permission file describing a Pipeline that is not yet implemented raises an error. Similarly, parameters described in a Permission file that are not described in the implemented Pipelines also raises an error. Runtime erros in the Proxy are reported to the client and logged, but proxy failure handling is not one of the objectives of this work

4.5 Leveraging Security with Scone

The described architecture can be used to protect the data from being misused by an ordinary user (*i.e.* a data analyst) and also enforces the security by increasing confidentiality and integrity when considering backends deployed in a cloud environment with untrustworthy servers. However, the secure proxy itself is susceptible to many forms of attacks. In section 1.5, we defined a threat model for our solution, identifying and prioritizing potential threats. We then mitigate these issues by using SGX and Scone to enhance security, even when considering scenarios such as deploying the proxy in an untrustworthy cloud environment. In our case, the secure proxy can be used as an encryption mechanism. Furthermore, the communication between client and proxy and between proxy and server is protected via TLS.

The proxy could be deployed in a trusted environment, thus protecting from adversaries that have access to the system memory. However, this is an unrealistic scenario and a nuisance for ease of access and scalability. Therefore, we consider that our solution should run in a TEE and due to being able to simplify application usage of Intel SGX we built a version of the secure proxy with SCONE.

Our solution utilises the SCONE Configuration and Attestation Service (CAS) as a means of attesting the secure proxy, storing application secrets and managing FSPF volumes. Therefore we define a policy session for the secure proxy and upload it to CAS. Figure 4.10 shows an example of policy that can be used.

Figure 4.10: Proxy Policy Session

```
name: proxy_policy
version: "0.2"

services:
  - name: proxy
    image_name: proxy-service
    command: ./secure_proxy -h 127.0.0.1:8081 -p 8080
    mrenclaves: [$MRENCLAVE]
    pwd: /
    fspf_path: /fspf.pb
    fspf_key: "$FSPF_KEY"
    fspf_tag: "$FSPF_TAG"
images:
  - name: proxy_service
    volumes:
       - name: encrypted_permissions_volume
         path: /permissions
    injection_files:
       - path: /etc/ca.crt
         content: $$SCONE::PROXY_CA_CERT.chain$$
       - path: /etc/server.crt
         content: $$SCONE::proxy.crt$$
       - path: /etc/server.key
         content: $$SCONE::proxy.key$$
       - path: /etc/client.crt
         content: $$SCONE::PROXY_CLIENT_CERT.crt$$
       - path: /etc/client.key
         content: $$SCONE::PROXY_CLIENT_CERT.key$$

secrets:
  - name: proxy-key
    kind: private-key
  - name: proxy
    private_key: proxy-key
    issuer: PROXY_CA_CERT
    kind: x509
  - name: PROXY_CLIENT_KEY
    kind: private-key
    export:
    - session: $PROXY_SIMPLECLIENT
  - name: PROXY_CLIENT_CERT
    private_key: PROXY_CLIENT_KEY
    issuer: PROXY_CA_CERT
    common_name: PROXY_CLIENT_CERT
    kind: x509
    export:
    - session: $PROXY_SIMPLECLIENT
  - name: PROXY_CA_KEY
    kind: private-key
  - name: PROXY_CA_CERT
    kind: x509-ca
    common_name: PROXY_CA
    private_key: PROXY_CA_KEY
    export:
    - session: $PROXY_SIMPLECLIENT

volumes:
  - name: encrypted_permissions_volume
```

Chapter 5

Evaluation

To better illustrate the interaction between the users of the proposed proxy solution, namely the **data owner** and the **data analyst**, with the system and also how the system itself manages front-end requests and back-end responses to meet the security requirements stipulated by the data owner, we then define two distinct *Scenarios* for data security on cloud environments. Section 5.2 describes how to protect data from a Kafka Server stored at an Object Storage solution in the form of S3 Buckets using encryption pipelines. Section 5.3 further utilises pipelines in a Business Intelligence scenario to protect privacy in a remote MongoDB database.

5.1 Experiment Environment

The evaluation used four different virtual machines (VMs) launched in a private cloud environment at UFCG. The virtual machines have been tested with the popular I/O benchmark FIO[1] and reported 2910 IOPS and 11.9MB/s writing speed. There were two different proxies deployed, one in a 30MB EPC machine and on in a 8MB EPC machine. Table 5.1 describes each flavour configuration. Sampaio (2018) described how to configure a OpenStack cloud environment to use Intel SGX for virtualization [53].

[1] https://fio.readthedocs.io/en/latest/fio$_doc.html$

Table 5.1: VM Configurations

vCPU	RAM	EPC
4	8 GB	30MB
4	8 GB	8MB
4	8 GB	0MB

5.2 Object Storage - Kafka and S3 Buckets

A simple use for our solution is as an encryption/decryption proxy as a means of enabling confidentiality in cloud-based data storage solutions such as an object storage back-end. This kind of storage is used as an easy way to store data coming from many different applications such as websites, mobile apps, backups, IoT devices, and big data analysis.

Amazon S3 is one of the most used object storage solutions; huge companies such as Netflix use S3 as a means of cloud storage [1]. However, despite access to S3 storage abstraction, named *S3 Bucket*, being private by default and only accessible through a secret key leaked data is very common and **millions** of records have been compromised already [16; 54; 55]. These incidents happen mainly due to "human error" in security configuration on cloud servers; accidental misconfigurations leads to private settings being changed to public. Furthermore, as we mentioned before, even the cloud provider should not be trusted, being imperfect and susceptible to mistakes and even collusion.

To avoid these leaks from happening and also protect data from inside attacks, one can choose to *only* store encrypted data on the cloud. The problem is that every front-end application storing data would need to implement an encryption algorithm before sending the data, and front-end application would need to decrypt this data before putting it to use. Moreover, encryption keys should not be used in untrusted environments. This is a perfect scenario to show how can a transparent proxy to solve this issue.

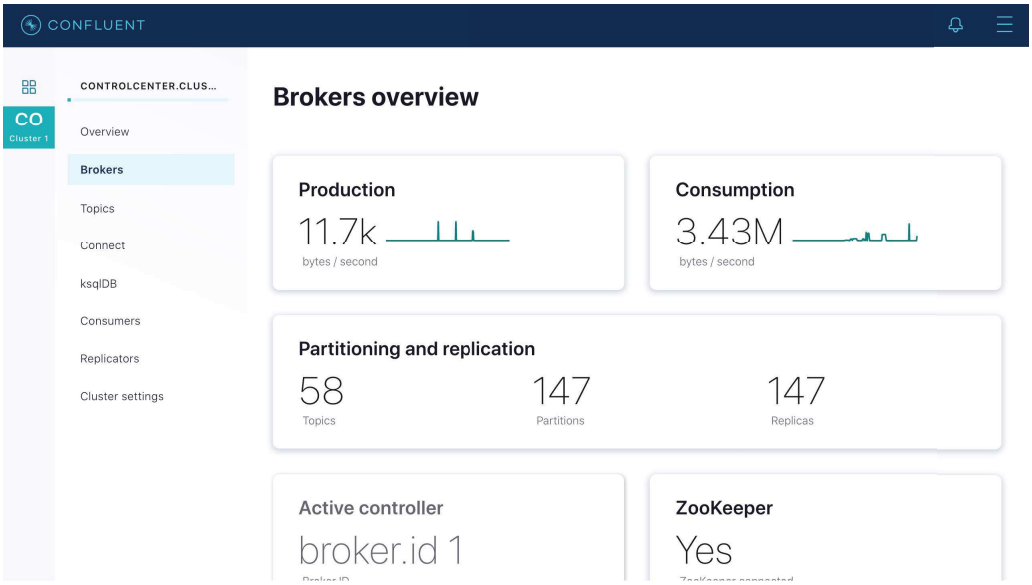
The Amazon S3 responds to REST API calls, therefore to connect it as a back-end to the proxy we had to implement support for the HTTP protocol, meaning the creation of a **HttpFrame**, **HttpClient** and a **HttpServer**. We also implemented an client-side pipeline **EncryptionPipeline** that encrypts all the messages sent by the Proxy-Client and a server-side pipeline **DecryptionPipeline** able to decrypt responses from the Proxy-Server. Encryption is done using AES-GCM with 256 bits key with the

help of Rust library "aes_gcm" [2].

We had at our disposal a private cloud infrastructure, so instead of using an AWS S3 server, we used MinIO as our object storage. MinIO is an Open Source high-performance object storage that is designed to be fully compatible with the S3 API.

It is in our best interest to test our proxy against well-known software components in a scenario as realistic as possible, this way we can have a better view of the performance overhead and also see that these applications are unaware of the proxy existence, showing the flexibility of our solution. We will be connecting our proxy to Kafka, one of the most used Open Source streaming platforms.

Figure 5.1: Kafka Dashboard



Kafka has a component called Kafka Connect, able to facilitate the connection between Kafka and other systems, such as databases, key-value stores, search indexes, and file systems [3]. When used to pull data from an external system, the connector is called a **Source** and is more commonly used to store metrics and do stream processing. A connector that pushes data to an external system is called **Sink** and is often used for offline analysis with map-reduce solutions or as a secondary index and backup, i.e when connected to an *S3 Bucket*.

In summary, our scenario describes the use of Apache Kafka, a distributed streaming platform, and this Kafka will have a **S3-Sink** connector that is used to backup data from topics in an S3 Bucket. The Sink will never directly connect to the cloud storage; instead, it will be sending requests to a secure proxy that will encrypt every message sent to the untrusted cloud storage (MinIO). The Proxy

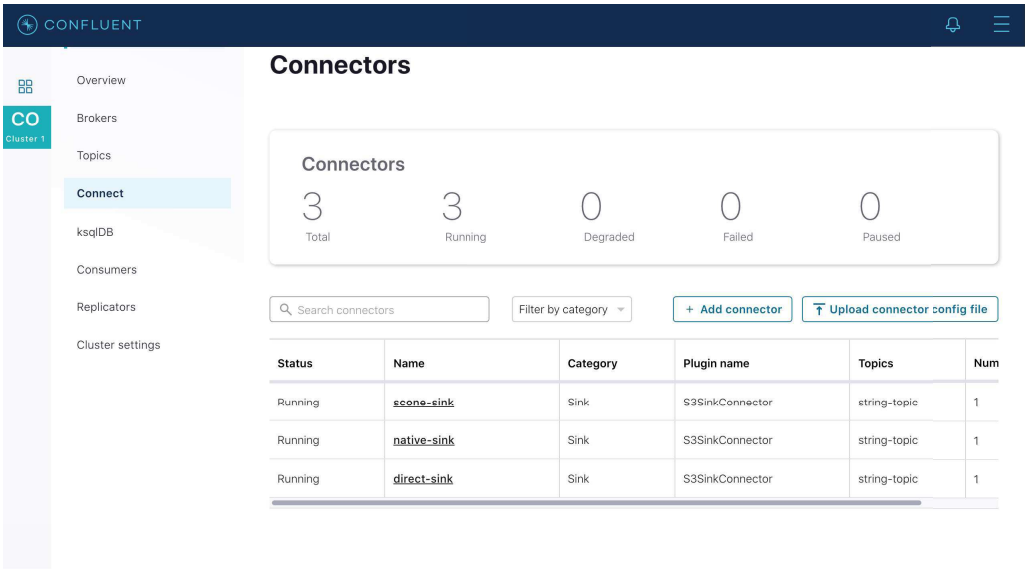
[2]https://docs.rs/aes-gcm/0.7.0/aes_gcm/

[3]https://docs.confluent.io/current/connect/index.html

runs in a VM with Intel SGX hardware. Therefore, Kafka is the untrusted front-end client connected to the Proxy, running in a VM, and the cloud storage is the back-end, residing in an untrusted server.

We created three buckets, "Direct" will receive data from Kafka directly. "Proxy-Native" will only receive data coming through a secure proxy running in native mode (outside an enclave), and finally, "Proxy-Scone" that receives data from the secure Proxy running inside an enclave with the help of SCONE.

Figure 5.2: Connectors



5.2.1 Methodology

It is possible to configure the size of the batch sent by the sink to the S3 bucket, and this is named the flush size. We decided to run an experiment with an increasing flush size so we can see the difference in performance between the three sinks. In this experiment, each message sent to Kafka has a size of 1.5KB, and we used flush sizes of 1, 10, 100 and 1000, this means that objects of 1.5KB, 15KB, 150KB and 1500KB are sent to each S3 bucket, respectively.

To send messages to Kafka, we used the benchmarking tool *kafka-producer-perf-test*, that simulates a Kafka producer and is a commonly used tool for evaluating throughput and latency. However, the throughput we are interested in is between the S3 Sink connector and the MinIO S3 bucket; for that, we collect the metrics provided by Kafka using Java Management Extensions (JMX).

The design of experiment evaluates different sink **types** as a factor {direct; proxy-native; proxy-scone} representing respectively a sink directly connected to MinIO, a sink connected to a secure proxy running in native mode, and a sink connected to a secure proxy running in SCONE. The other

factor is the **flush** size $\{1; 10; 100; 1000\}$ and the response variable is the **throughput**. The experiments were run 15 times at different times of the day to mitigate the effect of cloud network usage.

5.2.2 Results and Discussion

In this first experiment (Figure 5.4, Table 5.2), we can see that the direct connection to the MinIO scales with the flush size. This is a good indicator the MinIO cluster is not bottlenecked by disk performance, in these scenarios. In Figure 5.3, we zoomed in when the flush sized is equal to 1 to better observe the difference in performance.We also can see that the native proxy adds a performance overhead, most of it due to the encryption pipeline, and maintains a good scalability level of the direct connection. However we can see that the Scone proxy does not scale very well. SGX applications are limited by the size of the EPC memory, so we decided to reduce the EPC size in order to see if this was the bottleneck.

Figure 5.3: Encrypted S3 sink - Flush Size = 1

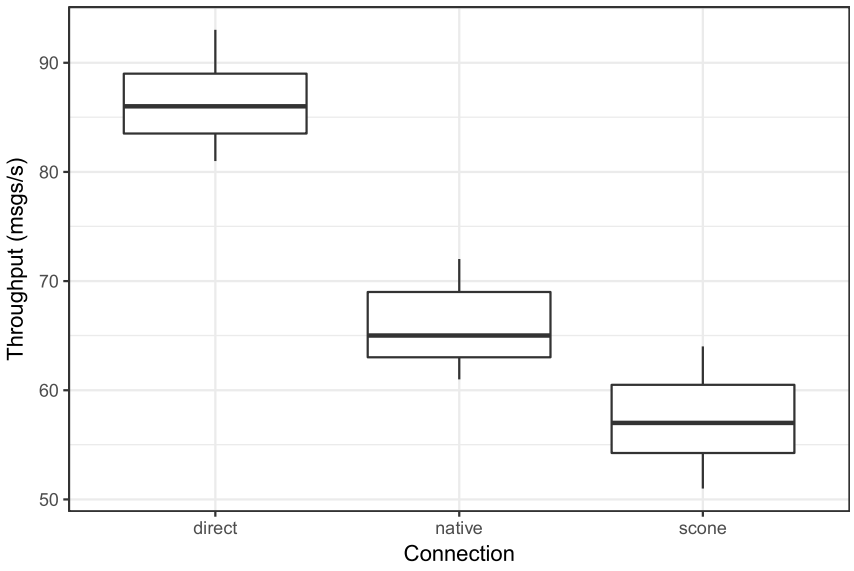


Figure 5.4: Encrypted S3 sink - 30MB EPC Machine

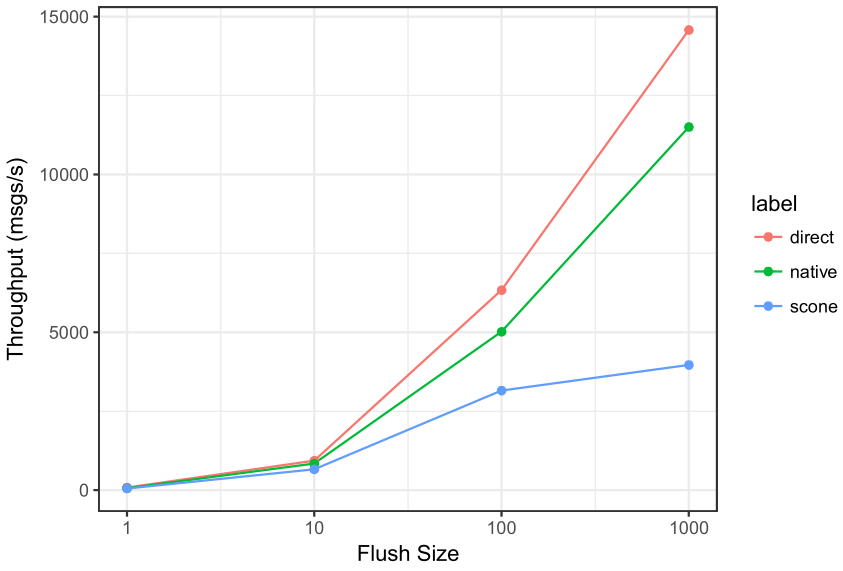


Table 5.2: Encrypted S3 sink - 30MB EPC Machine- Median

Sink	Flush size	Messages/sec
Direct	1	81
Direct	10	902
Direct	100	6267
Direct	1000	14381
Proxy-Native	1	68
Proxy-Native	10	838
Proxy-Native	100	5927
Proxy-Native	1000	12653
Proxy-Scone	1	51
Proxy-Scone	10	662
Proxy-Scone	100	3035
Proxy-Scone	1000	3944

We reduced the EPC size to 8MB, and run the same experiments (Figure 5.5). Now, the application is already bottlenecked at flush size 10, further indicating that EPC memory is the bottleneck

for this encryption pipeline. Although in this scenario this limitation seems to hinder the scalability potential of the proxy solution, enclaves dedicated physical memory is limited today to 256 MB, we do not have these machines at our disposal in UFCG private cloud, furthermore EPC size is expected to grow to be better suited for cloud computing [29; 42].

Figure 5.5: Encrypted S3 sink - 8MB EPC Machine

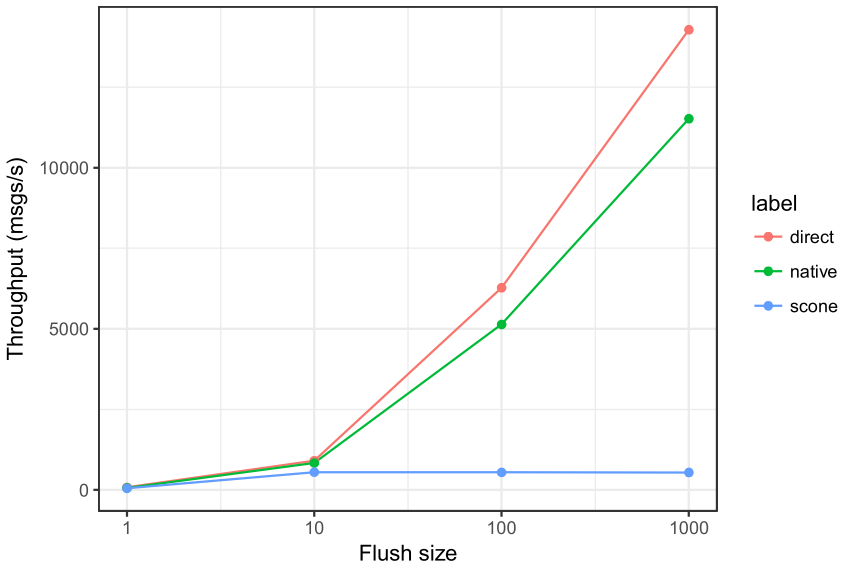


Figure 5.6: Encrypted S3 sink - Flush = 1 - 8MB EPC Machine

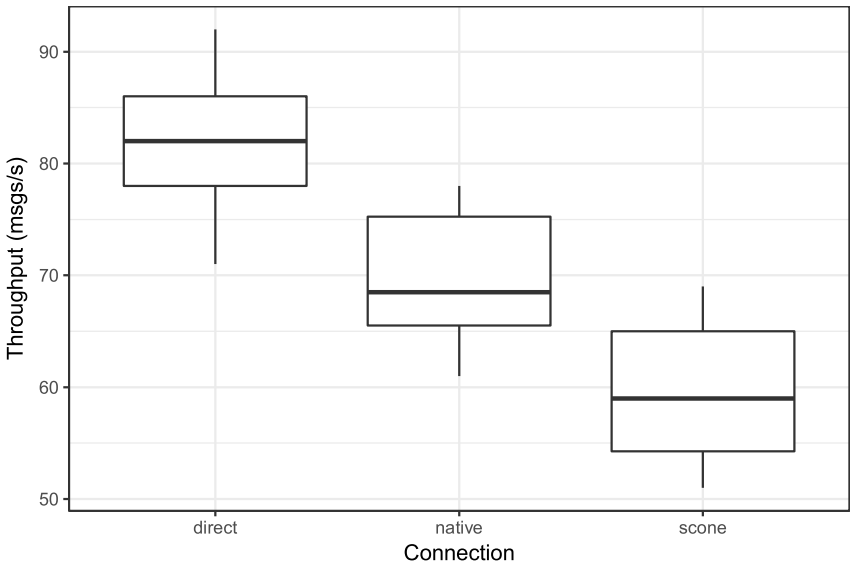


Table 5.3: Encrypted S3 sink - 8MB EPC- Median

Sink	Flush size	Messages/sec
Direct	1	98
Direct	10	897
Direct	100	6259
Direct	1000	13956
Proxy-Native	1	78
Proxy-Native	10	741
Proxy-Native	100	5019
Proxy-Native	1000	11160
Proxy-Scone	1	51
Proxy-Scone	10	423
Proxy-Scone	100	419
Proxy-Scone	1000	421

5.3 Database - Business Intelligence and MongoDB

To demonstrate how a proxy-based solution can add a privacy-preserving layer of security, we outline a Business Intelligence scenario, and for that, we choose Metabase as our front-end application. Users can summarize and visualize data, even not knowing SQL/NoSQL as they do not need to write queries. Metabase can be connected to SQL, H2, MongoDB, Amazon Redshift, BigQuery and many other databases. In this scenario, the back-end database connected to Metabase is a MongoDB instance running in the cloud.

MongoDB is a general-purpose, document-based, distributed database [28], supporting end-to-end encryption and role-based access control. Natively, MongoDB has no privacy guarantees, neither it should as preserving privacy is impossible without knowledge of the nature of the data. With a secure proxy, one can add anonymization and privacy-preserving algorithms as pipelines, data will be changed in a transparent fashion, and the client application can never know the information in the original data.

We connected a Metabase instance to a MongoDB server directly, calling the database “mongo-orignal”, and also connected to a secure proxy, calling it “mongo-secure-proxy”, running anonymiza-

tion and privacy preserving pipelines. Figures 5.7 and 5.8 picture Metabase UI and the connected databases.

Figure 5.7: Metabase Dashboard

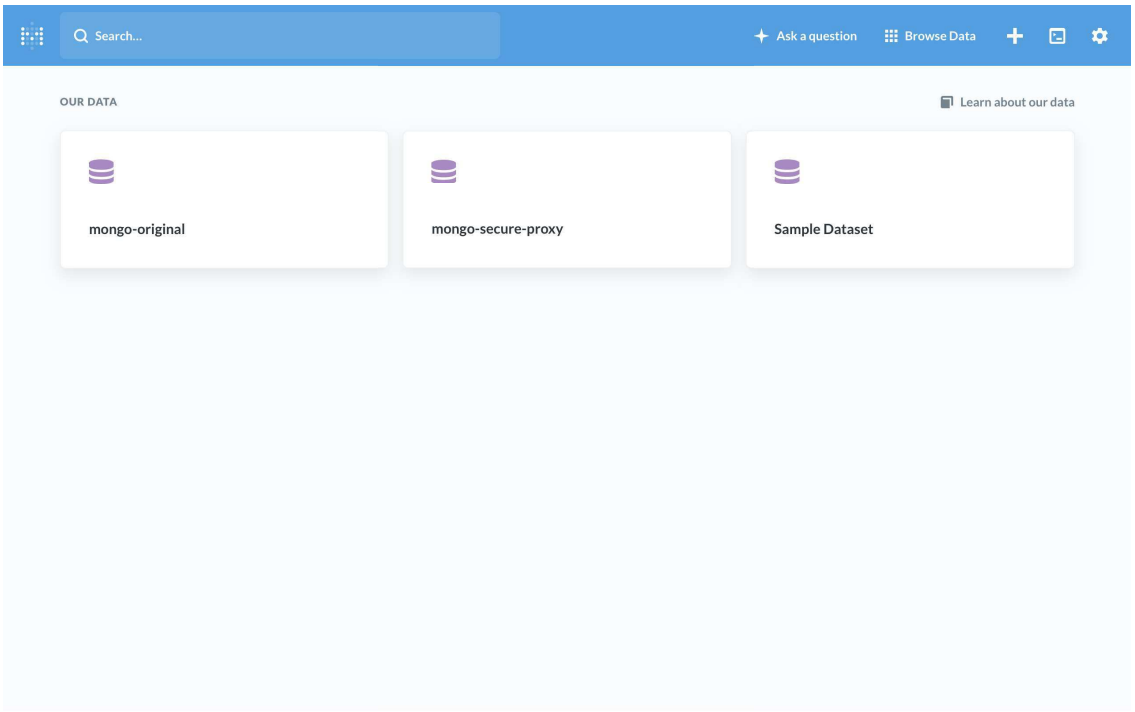
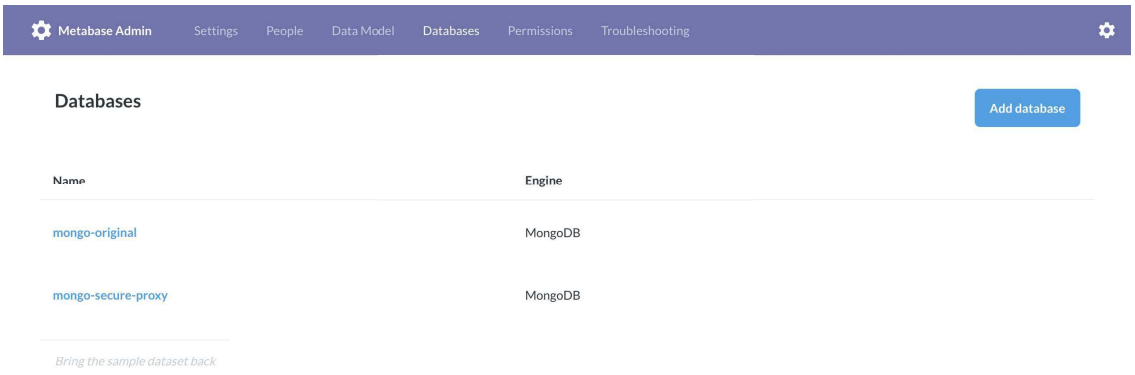


Figure 5.8: Metabase Dashboard



The sample database used has many collections as pictured in Figure 5.9. In this example we configured the proxy so that the client only has a access to a subset of these collections as seen in figure 5.10, this is done by selecting the desired collection in the permissions and enabling the access control pipeline for MongoDB collections.

Figure 5.9: MongoDB Original Collections Overview

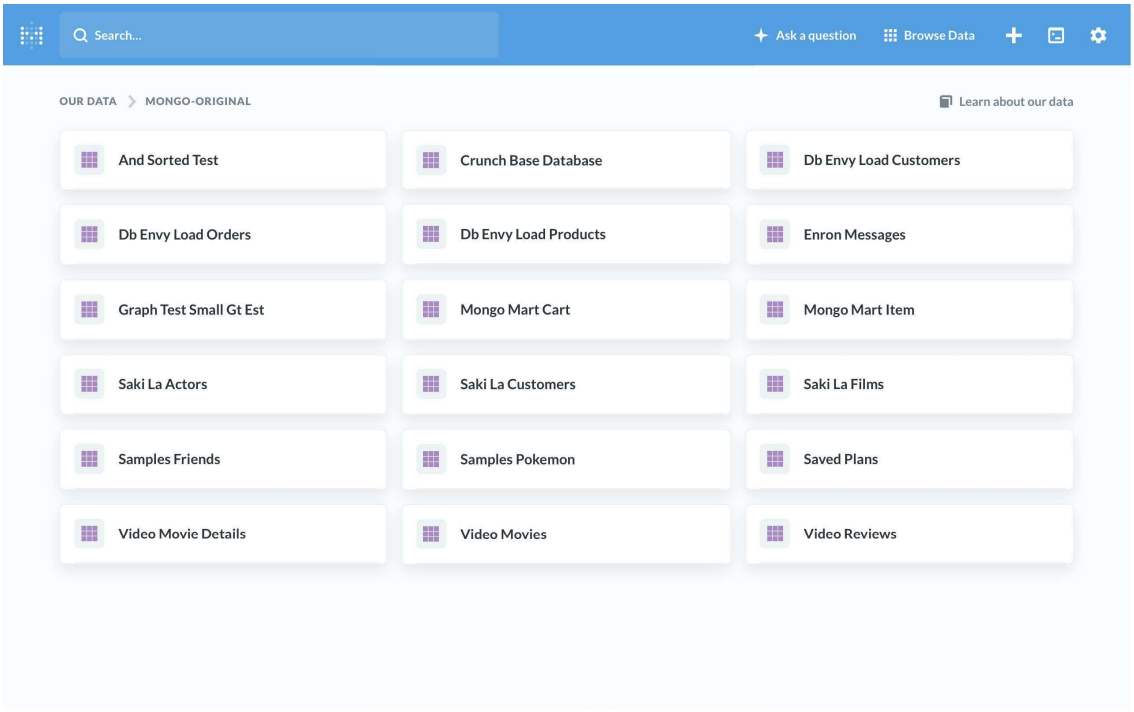


Figure 5.10: MongoDB Original Collections Overview

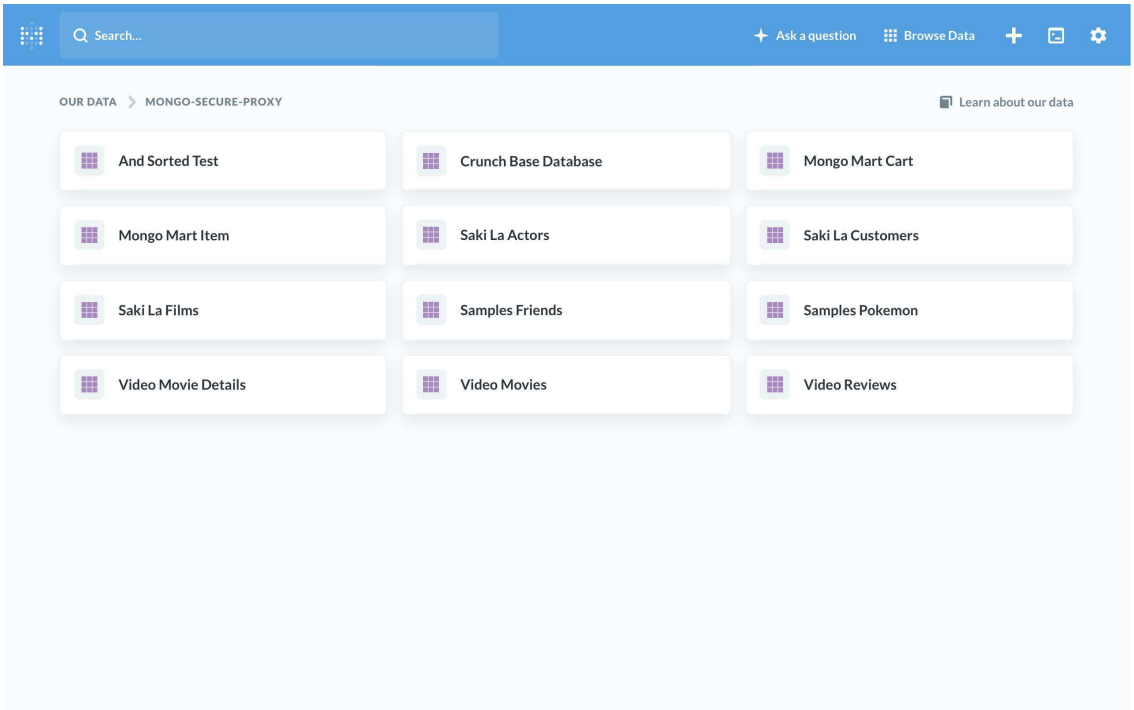


Figure 5.11 shows the result of querying all documents in the "Video Reviews" collection. This collection contains data about movie reviews made by users of a video rental company. Displaying user names can be considered as a privacy violation, also the review date and the rating can be used as Quasi-identifiers to single out users. Enabling an anonymization pipeline in the server can increase the user privacy by hiding some of this information. Figure 5.12 shows how the documents are transparently reported to Metabase with enforced privacy techniques.

Figure 5.11: Document View - Video Reviews Collection (MongoDB server)

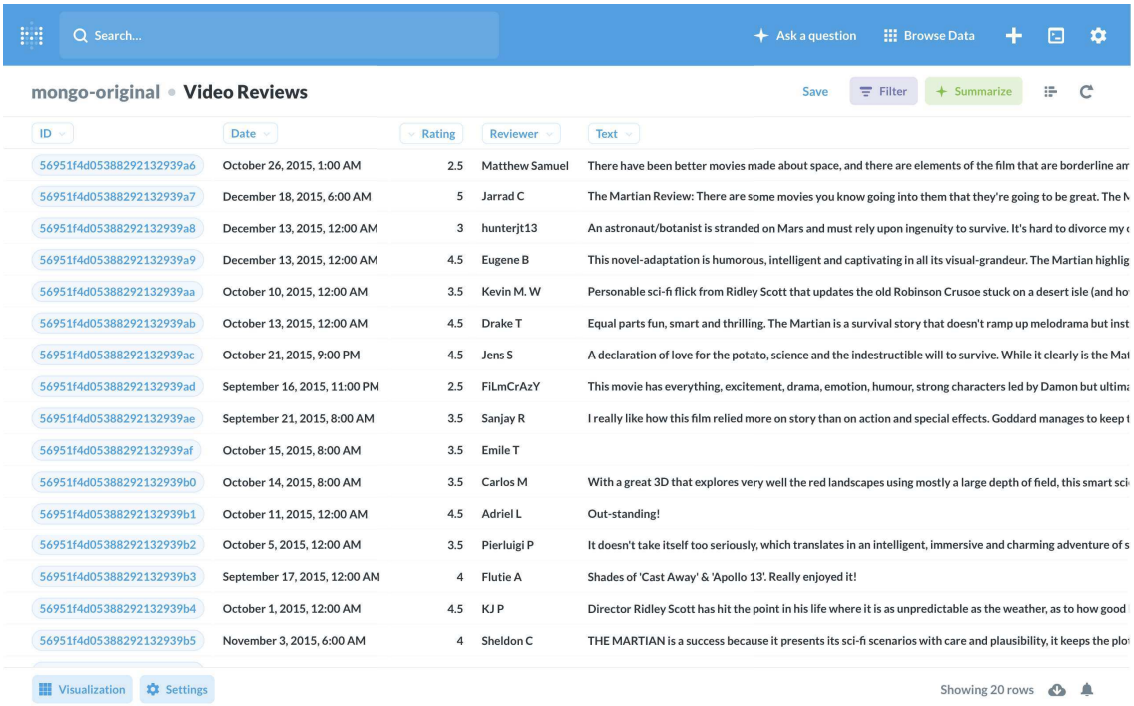
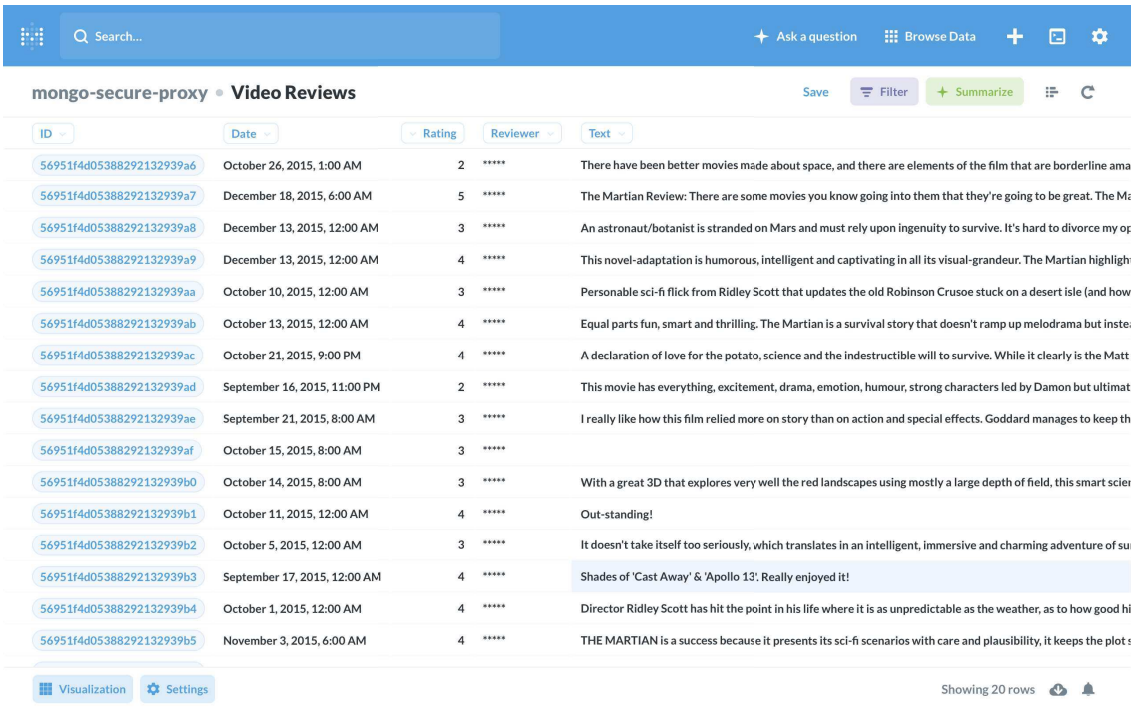


Figure 5.12: Document View - Video Reviews Collection (Proxy server)



Metabase database loading time as reported in logs was 9.73 seconds for the mongo-original and 15.62 seconds for the mongo-secure-proxy. Loading a small collection such as ‘video-reviews‘

Figure 5.13: Rating Bar Chart ID X Rating

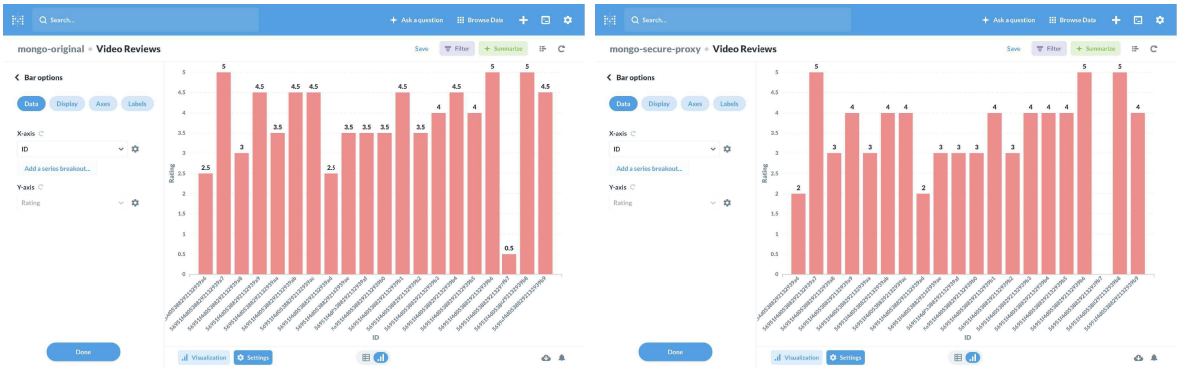
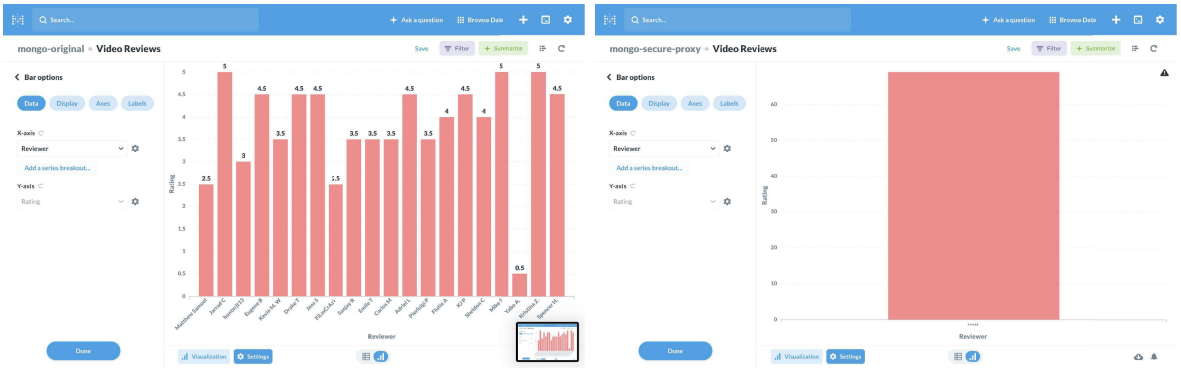


Figure 5.14: Rating Bar Chart reviewer X Rating



took metabase 1.89 seconds connected to mongo-original and 2.23 seconds for mongo-secure-proxy. When considering the usability overhead of using the proxy this performance drawback is barely noticeable for the average user, furthermore it is also overshadowed by the gains in security aspects.

5.3.1 Methodology

Although from a usability perspective, the performance overhead when using a secure proxy is somewhat unnoticeable, it is important to benchmark the proxy in a more stressful scenario. To do that we choose the benchmarking tool **Yahoo! Cloud Serving Benchmark** (YCSB)[4], able to measure the throughput of many different commands, such as insert, query, remove and update with regards to the number of threads, this way it is possible to observe performance overhead and scalability issues.

The design of experiment evaluates database **Throughput** as the response variable. The factors are connection type {direct; proxy-native; proxy-scone} that represent respectively YCSB directly connected to a MongoDB instance, connected to a secure proxy running outside a TEE, and a proxy

[4]https://github.com/brianfrankcooper/YCSB/tree/master/mongodb

running using SCONE. We also increase the number of threads to evaluate scalability with YCSB client threads levels {1; 2; 4; 6; 8}. This scenario is ideal for evaluating the performance overhead of different pipelines. Therefore, we run the experiments using the following strategies, {encryption; decryption; truncate; k-anon; random-response}. The record size in each message is approximately 1KB (10 fields, 100 bytes each, plus key) and the experiments were run 15 times in different times of the day to mitigate the effect of cloud network usage.

5.3.2 Results and Discussion

It is important to evaluate the proxy overhead when not running pipelines, as a baseline scenario. We decide to run an asynchronous YCSB MongoDB benchmark with 8 client threads against a MongoDB instance, a secure proxy in native mode and a secure proxy with SCONE. As there are pipelines that are triggered when queries are made and pipelines that are triggered when updating or inserting data we execute a Read-Only (RO) benchmark, Write-Only (WO) benchmark and a Read/Write (RW) benchmark with 1:1 read/write ratio.

Table 5.4: YCSB MongoDB - Write-Only Benchmark

Connection	Throughput(ops/sec)	AVG Latency	Min Latency	Max Latency	95th Latency	99th Latency
Direct	7012.6227	1015.7332	416	207417	1282	2839
Proxy-Native	4721.9703	1669.9426	638	252671	2919	5815
Proxy-SCONE	4119.7020	1922.29235	627	209023	3541	8607

Table 5.4 outlines the performance difference in a Write-Only scenario between the three connections. *Throughout* is the response variable we are evaluating in this work and we can see that there is a 33% performance overhead when using the proxy in native mode. There is a 13% performance overhead comparing native mode and SCONE. In the Read-Only benchmark, Table 5.5, the same 33% overhead is observed between direct connection and proxy-native. However, the overhead between native and scone proxy increased a bit to 15%.

Table 5.5: YCSB MongoDB - Read-Only Benchmark

Connection	Throughput(ops/sec)	AVG Latency	Min Latency	Max Latency	95th Latency	99th Latency
Direct	10645.3192	747.1403	322	55903	1016	1257
Proxy-Native	7154.5087	1109.4890	443	203775	1473	2839
Proxy-SCONE	6167.6779	1374.6632	573	211711	1892	3263

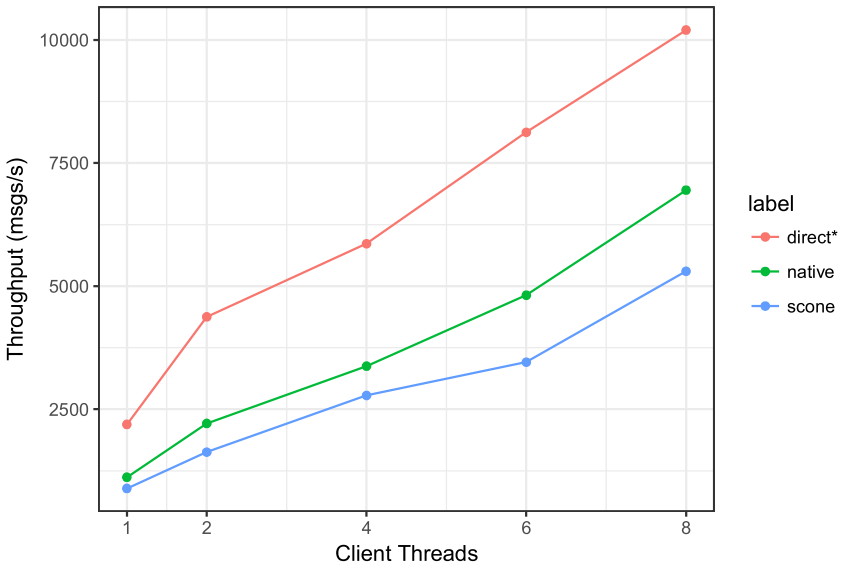
The last baseline benchmark is the Read/Write, with equal number of reads and writes, Table 5.6. In this benchmark the slow write throughput affects the secure proxy a lot more than the direct connection, resulting in 48% and 58% performance overhead compared to proxy-native and proxy-scone respectively. The main cause for this overhead is the lack of cache in the proxy, furthermore although the implemented proxy is designed to deal well with asynchronous I/O, it is still a proof-of-concept solution and lacks some of the performance improvements that production-level applications such as MongoDB have. When we consider that the solution was initially developed to be used by a singular client user, we consider being able to perform 5186 operations per second in a synthetic benchmark a success.

Table 5.6: YCSB MongoDB - Read/Write Benchmark (1:1)

Connection	Throughput(ops/sec)	AVG Latency	Min Latency	Max Latency	95th Latency	99th Latency
Direct (R)	10760.5561	737.9023	343	60031	1027	1411
Direct (W)	10760.5561	739.6209	337	59807	1026	1430
Proxy-Native (R)	5680.4626	1276.8072	470	206335	2255	4035
Proxy-Native (W)	5680.4626	1495.0512	486	206463	2915	6247
Proxy-SCONE (R)	5186.2915	1414.9998	537	58303	2135	4663
Proxy-SCONE (W)	5186.2915	1641.8104	540	58591	2883	8263

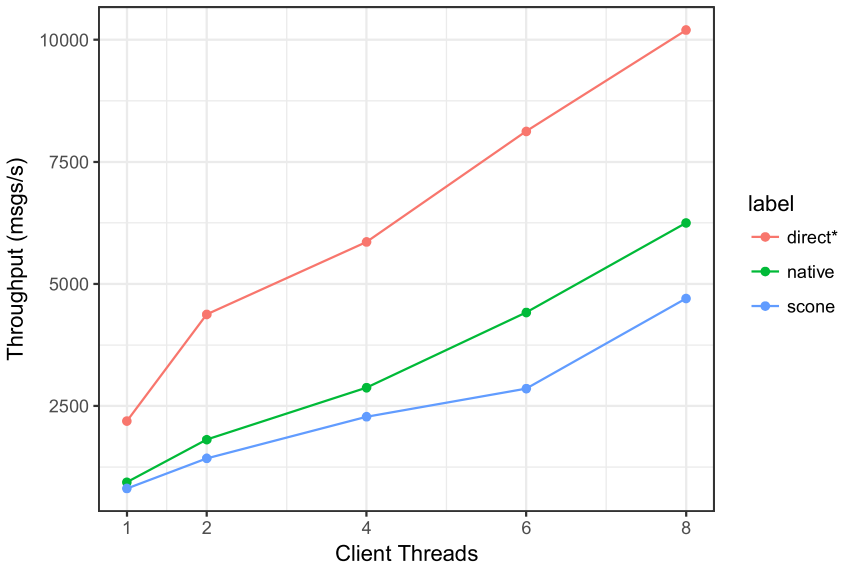
We now want to evaluate the proxy overhead when using pipelines. In this benchmark we used the truncate pipeline, meaning that every float number is truncated before returning the query results to the client. The setup step of the benchmark adds an additional float field (8 bytes), the benchmark then queries for a single record with the random value queried. In Figure 5.15 we can see that despite the known overhead, inherited from the baseline, we have an additional a 20% performance overhead, with 8 threads, when using the proxy-scone compared to the direct connection, furthermore the proxy scales well with multiple client threads. This is a very simple algorithm ($O(1)$) and the cost is mostly due to the buffer being transformed in the **Frame** abstraction.

Figure 5.15: Float query - Integer result (Truncate Pipeline)



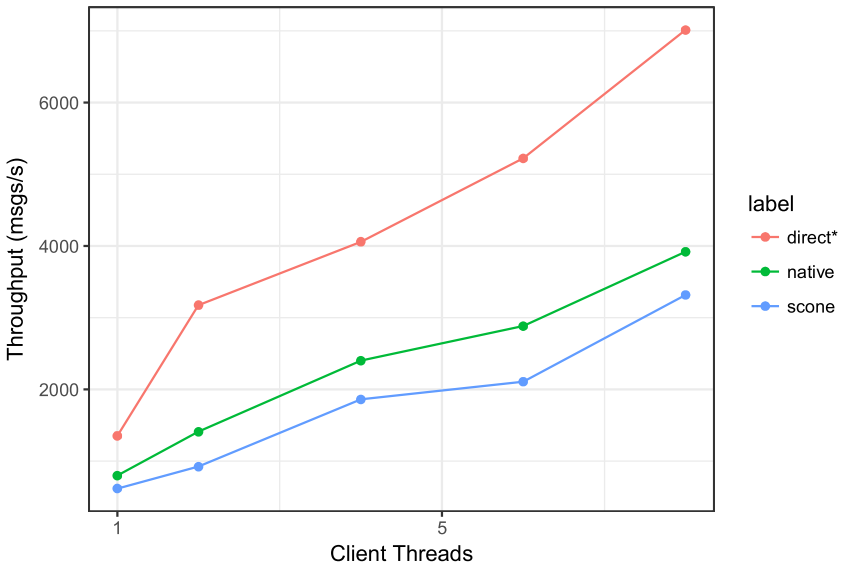
It is important to evaluate the overhead when using a more complex pipeline so we used the k-Anonymity pipeline. The setup step creates an additional integer (8 bytes) field "year". The benchmark then queries for set of documents inside a range. The proxy will use a k-Anonymity pipeline with k=3 meaning that each record is indistinguishable from at least 2 other records. The last digits of the year value are hidden until a k=3 level of anonymity is reached. The overhead, shown in Figure 5.16, with 8 threads is 57%, this k-anon algorithm as described in 2.1.1 is greedy algorithm with $O(log(n))$. The volume of data handled by the proxy at once is far less than in the Kafka-Minio scenario, even with 8 threads, therefore we have no problems with limited EPC.

Figure 5.16: Year range query (kAnon Pipeline)



An example of privacy preserving algorithm that guarantees differential privacy is **Random Response**. In the setup step, the benchmark creates an additional text field "diabetes" to simulate a medical report. When querying for this field the pipeline will flip a coin to decide whether it will give the correct response or if it will give a random response, the odds are 0.5. The overhead results are similar to the truncate pipeline, as this algorithm is also quite simple and fast,

Figure 5.17: Diabetes (Random Response Pipeline)



We also evaluate a confidentiality enhancing encryption pipeline. The results are shown on figure

5.18. Then, we query for documents but now we need to used the decryption pipeline, results are shown on figure 5.19.

Figure 5.18: Encrypted int insert (encryption Pipeline)

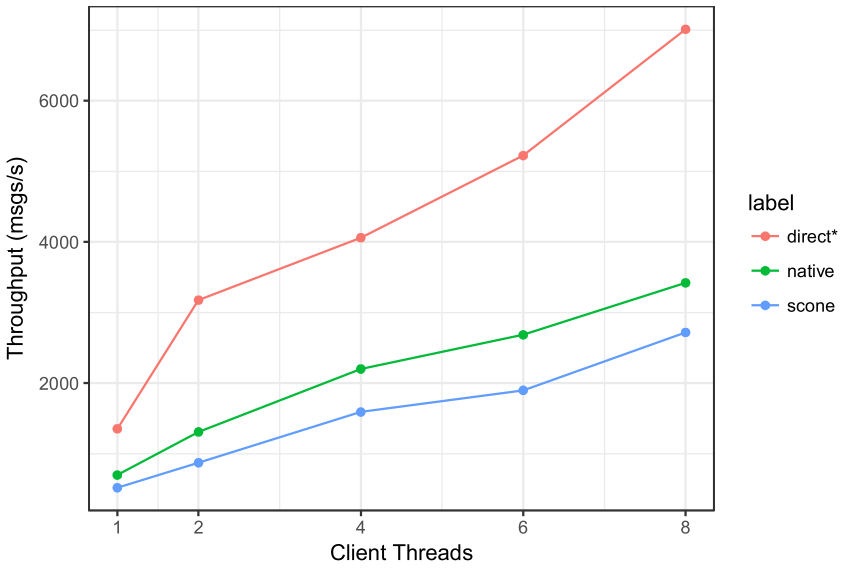
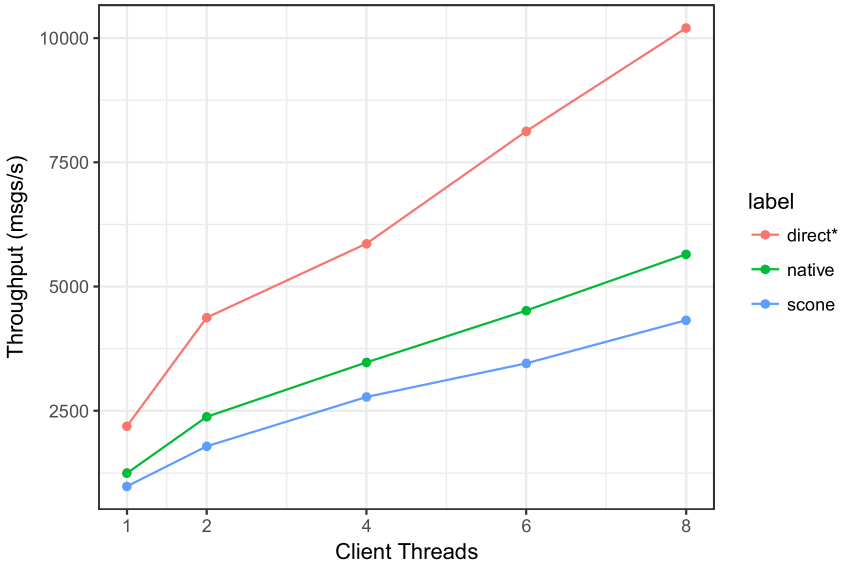


Figure 5.19: Encrypted query range (encryption Pipeline)



Chapter 6

Conclusion

In this work, we proposed a general-purpose solution for adding a security layer of protection to well-known cloud-based applications. We listed similar solutions that tried to enforce security in cloud environments. We defined an architecture that enabled a transparent proxy to connect different front-end client applications to different back-end cloud-based servers, and this architecture was also extensible in the form of pipeline plugins that are divided between client-side and server-side algorithms.

6.1 Summary of Results

Our results show that in a cloud backup scenario, the secure proxy running against an HTTP back-end with an encryption pipeline has a 23% performance overhead when compared to a direct connection. The main issue is that due to the reduced resources of current TEE solutions, the same benchmark running against the proxy inside an Intel SGX enclave has an underwhelming 71% performance overhead. However, new developments in hardware solutions may help mitigate this issue. In a data analysis scenario, we found out that running complex privacy pipelines can be very costly with the proxy adding up to 83% performance overhead, however from the usability perspective, in most cases the client only takes a bit of extra time to load all the data, from 9.73 seconds connected directly to 15.62 seconds using a secure proxy.

We developed support to two different protocols and were able to use them to manage the connection between four different well-known applications. Furthermore, we implemented seven different pipelines able to enforce confidentiality and privacy; all of them are established algorithms developed by the research community. We also describe the proxy architecture in the form of models, outlining

the extension potential to the protocol abstraction and the pipeline plugins, thus enabling the development of the secure proxy in other programming languages. This work is an engineering research with scenarios demonstrated with proof-of-concept solutions; however, we evaluated the secure proxy close to a production environment.

6.2 Future Work

Many improvements can be made to this solution, especially considering large scale usage in a production environment. There are many examples of proxies being used to reduce bandwidth and improves response times by caching and reusing frequent requests [49; 39]. Our proxy solution does not perform caching, thus wasting a great opportunity to reduce overhead. When considering cache for a secure proxy running inside a TEE, it is important to consider hardware limitations, current Intel SGX available hardware severely lacks resources (EPC memory, limited to 256MB) that would be necessary to benefit from caching requests. However, this maybe could be mitigated by integrating distributed memory object caching system to our solution, also with an evaluation whether it would be best to use a generic caching engine or use one engine specific to each protocol.

The initial motivation of the solution was deploying a proxy in the client machines to allow secure connections to back-ends, therefore benefiting from the EPC memory available at modern Intel personal computers. Consequently, our solution only considers one (at most a few) client users accessing the proxy at once. One may want to have a centralised proxy acting as a firewall to all connections from a company to a public cloud. Using a layer-4 load balancer solution to manage multiple connections seems to be an excellent starting point to mitigate this issue. Another interesting research opportunity is evaluating the usage of multiple proxy instances to run distributed pipelines, for example, using map-reduce strategies, this would achieve better scalability and utilisation of TEE hardware.

Disaster recovery is also a must if you consider using a centralised proxy solution. The current solution does not address this at all as it was originally designed to be stateless. In fact, in the original design the proxy would consist in multiple microservices, as each of them must run inside enclaves this was simplified to reduce memory footprint; however, future improvements in TEE hardware may change this and encourage a microservices division [44].

6.3 Other Achivements

While developing this work we also had the opportunity to work in other projects and published the following works: “Exploiting SLAs through Application of Economic Analysis on Datacenters’ Autonomic Management”, a full paper waiting publication on SBRC’20 (*Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*) [24], also “*Processamento confidencial de dados de sensores na nuvem*”, a tutorial to be published on Symposium on Information and Computational Systems Security (SBSeg) [8], and also a poster published on EuroSys’19 [44].

Bibliography

[1] Barak Alon. Mezzfs — mounting object storage in netflix's media processing platform. https://netflixtechblog.com/mezzfs-mounting-object-storage-in-netflixs-media-processing-platform-cda01c446ba. [accessed: 28.09.2020].

[2] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[3] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. {SCONE}: Secure linux containers with intel {SGX}. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 689–703, 2016.

[4] Pierre-Louis Aublin, Florian Kelbert, Dan O'Keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. Libseal: Revealing service integrity violations using trusted execution. New York, NY, USA, 2018. Association for Computing Machinery.

[5] Alan P Bates. Privacy—a useful concept? *Social forces*, 42(4):429–434, 1964.

[6] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Hybrids on steroids: Sgx-based high performance bft. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 222–237, New York, NY, USA, 2017. Association for Computing Machinery.

[7] Jacques J Berleur and Klaus Brunnstein. *Ethics of computing: codes, spaces for discussion and law*. Springer Science & Business Media, 1996.

[8] Andrey Brito, Clenimar Souza, Fabio Silva, Lucas Cavalcante, and Matteus Silva. Processamento confidencial de dados de sensores na nuvem. In *2020 Symposium on Information and Computational Systems Security SBSeg*, 2020.

[9] Brooke Bullek, Stephanie Garboski, Darakhshan J. Mir, and Evan M. Peck. Towards understanding differential privacy: When do people trust randomized response technique? In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, page 3833–3837, New York, NY, USA, 2017. Association for Computing Machinery.

[10] Dorian Burihabwa, Pascal Felber, Hugues Mercier, and Valerio Schiavoni. Sgx-fs: Hardening a file system in user-space with intel sgx. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 67–72. IEEE, 2018.

[11] Richard Chow, Philippe Golle, Markus Jakobsson, Elaine Shi, Jessica Staddon, Ryusuke Masuoka, and Jesus Molina. Controlling data in the cloud: outsourcing computation without outsourcing control. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 85–90, 2009.

[12] Chris Clifton and Tamir Tassa. On syntactic anonymity and differential privacy. In *2013 IEEE 29th International Conference on Data Engineering Workshops (ICDEW)*, pages 88–93. IEEE, 2013.

[13] L. Columbus. 83% of enterprise workloads will be in the cloud by 2020. https://www.forbes.com/sites/louiscolumbus/2018/01/07/83-of-enterprise-workloads-will-be-in-the-cloud-by-2020/35b611726261, note=[accessed: 28.09.2020],.

[14] Intel Corporation. Intel software guard extensions. Cryptology ePrint Archive, Report 2016/086, 2015. https://software.intel.com/sites/default/.

[15] Victor Costan and Srinivas Devadas. Intel sgx explained. Cryptology ePrint Archive, Report 2016/086, 2016. http://eprint.iacr.org/2016/086.

[16] D. Deahl. Verizon partner data breach exposes millions of customer records. https://www.theverge.com/2017/7/12/15962520/verizon-nice-systems-data-breach-exposes-millions-customer-records. [accessed: 28.09.2020].

[17] Michael L Dertouzos and Joel Moses. Computer age: a twenty-year view. 1979.

[18] Camara dos Deputados do Brasil and Senado Federal do Brasil. General data protection regulation (gdpr) compliance guidelines. https://www.lgpdbrasil.com.br, note=[accessed: 28.09.2020],.

[19] Camara dos Deputados do Brasil and Senado Federal do Brasil. Home - lgpd. https://www.lgpdbrasil.com.br, note=[accessed: 28.09.2020],.

[20] Cynthia Dwork and Jing Lei. Differential privacy and robust statistics. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 371–380, 2009.

[21] Cynthia Dwork, Aaron Roth, et al. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[22] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[23] Factor, Meth, Naor, Rodeh, and Satran. Object storage: the future building block for storage systems. In *2005 IEEE International Symposium on Mass Storage Systems and Technology*, pages 119–123, 2005.

[24] Edurado Falcao, Lucas Cavalcante, Rafael Falcao, Jose Nunes, Kaio Oliveira, and Andrey Brito. Poster: Vallum: Database privacy, confidentiality and access rights for sensitive data in cloud environments. In *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019.

[25] Craig Gentry and Dan Boneh. *A fully homomorphic encryption scheme*, volume 20. Stanford university Stanford, 2009.

[26] Metabase Inc. Metabase: An open source business intelligence server. https://www.metabase.com. [accessed: 28.09.2020].

[27] Microsoft Inc. Data visualization | microsoft powerbi. https://powerbi.microsoft.com/pt-br/, note=[accessed: 28.09.2020],.

[28] MongoDB Inc. Mongodb: The most popular database for modern apps. https://www.mongodb.com. [accessed: 28.09.2020].

[29] Simon Johnson. Scaling towards confidential computing. https://systex.ibr.cs.tu-bs.de/systex19/slides/systex19-keynotesimon.pdf. Keynote presentation at SysTEX 2019.

[30] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. *White paper*, 2016.

[31] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. Pesos: Policy enhanced secure object store. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–17, 2018.

[32] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7, 2011.

[33] Kristen LeFevre, David J DeWitt, and Raghu Ramakrishnan. Mondrian multidimensional k-anonymity. In *22nd International conference on data engineering (ICDE'06)*, pages 25–25. IEEE, 2006.

[34] B. Li, N. Weichbrodt, J. Behl, P. Aublin, T. Distler, and R. Kapitza. Troxy: Transparent access to byzantine fault-tolerant systems. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 59–70, 2018.

[35] Yehida Lindell. Secure multiparty computation for privacy preserving data mining. In *Encyclopedia of Data Warehousing and Mining*, pages 1005–1009. IGI Global, 2005.

[36] Gunasekaran Manogaran, Chandu Thota, and M Vijay Kumar. Metaclouddatastorage architecture for big data security in cloud computing. *Procedia Computer Science*, 87:128–133, 2016.

[37] MinIO. Minio: High performance, kubernetes native object storage. https://min.io. [accessed: 28.09.2020].

[38] Narayanan and Shmatikov. Robust de-anonymization of large sparse datasets. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 111–125, May 2008.

[39] Clément Nedelcu. *Nginx HTTP Server: Adopt Nginx for Your Web Applications to Make the Most of Your Infrastructure and Serve Pages Faster Than Ever*. Packt Publishing Ltd, 2010.

[40] Solomon Negash and Paul Gray. Business intelligence. In *Handbook on decision support systems 2*, pages 175–193. Springer, 2008.

[41] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting {SGX} enclaves from practical side-channel attacks. In *2018 {Usenix} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 227–240, 2018.

[42] Meni Orenbach, Andrew Baumann, and Mark Silberstein. Autarky: closing controlled channels with self-paging enclaves. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[43] Siani Pearson and Azzedine Benameur. Privacy, security and trust issues arising from cloud computing. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 693–702. IEEE, 2010.

[44] Ronny Peterson, Andre Carvalho, Gabriel Fernandez, Lucas Cavalcante, Altrigran Silva, and Christof Fetzer. Poster: Vallum: Database privacy, confidentiality and access rights for sensitive data in cloud environments. In *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019.

[45] Sandro Pinto and Nuno Santos. Demystifying arm trustzone: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 51(6):1–36, 2019.

[46] Vernon C. Polite and Arlin H. Adams. Critical thinking and values clarification through socratic seminars. *Urban Education*, 32(2):256–278, 1997.

[47] Christian Priebe, Divya Muthukumaran, Dan O' Keeffe, David Eyers, Brian Shand, Ruediger Kapitza, and Peter Pietzuch. Cloudsafetynet: Detecting data leakage between cloud tenants. In *Proceedings of the 6th Edition of the ACM Workshop on Cloud Computing Security*, CCSW '14, page 117–128, New York, NY, USA, 2014. Association for Computing Machinery.

[48] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using sgx. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 264–278. IEEE, 2018.

[49] Alex Rousskov and Valery Soloviev. A performance study of the squid proxy on http/1.0. *World Wide Web*, 2(1-2):47–67, 1999.

[50] S. Rüsch, K. Bleeke, and R. Kapitza. Bloxy: Providing transparent and generic bft-based ordering services for blockchains. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 305–30509, 2019.

[51] M. Sabt, M. Achemlal, and A. Bouabdallah. Trusted execution environment: What it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64, 2015.

[52] Pierangela Samarati and Latanya Sweeney. Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression. 1998.

[53] Lilia Sampaio. Estratégias para o suporte a ambientes de execução confiável em sistemas de computação na nuvem. In *UFCG Master Thesis*, 2018.

[54] K. Sheridan. Dow jones data leak results from an aws configuration error. https://www.darkreading.com/cloud/dow-jones-data-leak-results-from-amazon-aws-configuration-error/d/d-id/1329382? [accessed: 28.09.2020].

[55] K. Sheridan. Twilio security incident shows danger of misconfigured s3 buckets. https://www.darkreading.com/cloud/twilio-security-incident-shows-danger-of-misconfigured-s3-buckets/d/d-id/1338447. [accessed: 28.09.2020].

[56] Fabio Silva, Matteus Silva, and Andrey Brito. Kafkaproxy: data-at-rest encryption and confidentiality support for kafka clusters. In *2020 Symposium on Information and Computational Systems Security (SBSeg)*, 2020.

[57] BSON Spec. Bson (binary json): Specification [electronic resource]. http://bsonspec. org/spec. html. [accessed: 28.09.2020].

[58] Latanya Sweeney. Weaving technology and policy together to maintain confidentiality. *The Journal of Law, Medicine & Ethics*, 25(2-3):98–110, 1997.

[59] Miklos Szeredi. Fuse: Filesystem in userspace. *http://fuse. sourceforge. net*, 2010.

[60] Alan F Westin. Privacy and freedom. *Washington and Lee Law Review*, 25(1):166, 1968.

[61] Michael E Whitman and Herbert J Mattord. *Principles of information security*. Cengage Learning, 2011.

[62] Li Yan, Xiaowei Hao, Zelei Cheng, and Rui Zhou. Cloud computing security and privacy. In *Proceedings of the 2018 International Conference on Big Data and Computing*, pages 119–123, 2018.

[63] Zardari, Jung, and Zakaria. K-nn classifier for data confidentiality in cloud computing. In *2014 International Conference on Computer and Information Sciences (ICCOINS)*, pages 1–6, 2014.

[64] Zuo, Lin, and Zhang. Why does your data leak? uncovering the data leakage in cloud from mobile apps. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1296–1310, 2019.

Appendix A

Appendix

A.1 Metabase

Metabase is an open source business intelligence tool [26]. Business intelligence systems combine operational data with analytical tools to present complex and competitive information to planners and decision makers. The objective is to improve the timeliness and quality of inputs to the decision process. Business Intelligence is used to understand the capabilities available in the firm; the state of the art, trends, and future directions in the markets, the technologies, and the regulatory environment in which the firm competes; and the actions of competitors and the implications of these actions [40]. Databases connected to Metabase are listed in a homepage. Navigating through Metabase enables the user to see databases and their contents. It is possible to click on tables to shows rows, also it can perform automatic exploration in the form of what is called by the company as "x-rays", it is possible to see the data reference view of tables to learn more about them. In Metabase, one way to start exploration of tables it is asking a more detailed custom question using the notebook editor, or write a new SQL query. Regarding security, Metabase utilises HTTPS and is able to encrypt data at rest and for authentication it uses username and password, users are divided in groups and each group has different access permissions.

A.2 MongoDB

MongoDB is an open source NoSQL "document store" database [28]. Although MongoDB is non-relational, it implements many features of relational databases, such as sorting, secondary indexing and range queries. MongoDB does not organize data in tables with columns and rows. Instead, data

is stored in “documents”, each of which is an associative array of scalar values, lists, or nested associative arrays. MongoDB documents are serialized naturally as Javascript Object Notation (JSON) objects, and are in fact stored internally using a binary encoding of JSON called BSON [57].

A.2.1 Security

MongoDB offers support to mTLS as a means of authentication, it also has support for username and password, access control is done through role based permissions for users and data in transit is protected with TLS. In MongoDB Enterprise version, there is support for data at rest encryption and also auditability, however we only evaluate the usage of the Community version.

A.2.2 Mongo WireProtocol

The MongoDB Wire Protocol is a simple socket-based, request-response style protocol. Clients communicate with the database server through a regular TCP/IP socket. There is no connection handshake. All integers in the MongoDB wire protocol use little-endian byte order (least-significant byte first).

Standard Message Header. There are two types of messages, client requests and database responses. Each message has a standard message header, that is followed by request-specific data. Table A.1 describes the message header structure

Table A.1: Structure - Message header

type	value	description
int32	messageLength	total message size
int32	requestID	identifier for this message
int32	responseTo	requestID from the original request
int32	opCode	request type

Client Request Messages. Clients can send request messages that specify all but the OP_REPLY opCode. OP_REPLY is reserved for use by the database. Only the OP_QUERY and OP_GET_MORE messages result in a response from the database. There will be no response sent for any other message. You can determine if a message was successful with a getLastError command. The OP_UPDATE message is used to update a document in a collection. The OP_INSERT message is used to insert

one or more documents into a collection. The OP_QUERY message is used to query the database for documents in a collection. The OP_GET_MORE message is used to query the database for documents in a collection. The OP_DELETE message is used to remove one or more documents from a collection. The OP_KILL_CURSORS message is used to close an active cursor in the database. This is necessary to ensure that database resources are reclaimed at the end of the query.

Database Response Messages. The OP_REPLY message is sent by the database in response to an OP_QUERY or OP_GET_MORE message. Table A.2 describes the message structure.

Table A.2: Structure - Reply message

type	value	description
MsgHeader	header	standard message header
int32	resposneFlags	bit vector
int32	cursorID	cursor id if client needs to do get more's
int32	startingFrom	where in the cursor this reply is starting
int32	numberReturned	number of documents in the reply
document*	documents	documents

A.3 Apache Kafka

Apache Kafka is a open source publish/subscribe system [32]. Publish/subscribe systems are known for its loosely coupled nature of interactions between its participating entities, thus being ideal for large scale applications [22]. One of the defining entities is the Publisher and is able to send events to a server, in Kafka these event servers are called brokers. Subscribers are another entity and are interested in being notified about an specific or a pattern of events. Kafka utilises an event scheme based on topics, meaning that all events are associated to a specific topic, publishers produce messages to a topic and subscribers therefore are notified by changes in its subscribed topics. To facilitate scalability, Kafka divides topics in a set of structures called partitions, that way multiple producers and consumers can read and write data simultaneously. Kafka can also communicate with external systems by using components called Connectors. There are two types of connectors, the ones outputing data from Kafka are called Sink connectors and function as consumers, and the ones acting as producers, sending data to Kafka, are called Source connectors.

Kafka supports mutual TLS authentication, and also uses SSL/TLS for data-in-transit encryption,

however does not encrypt locally stored data. Access control is performed in the form of Access Control Lists (ACLs) and there is also support for Role-Base Access Control (RBAC).

A.4 MinIO

MinIO is a High Performance Object Storage released under Apache License v2.0 [37]. Object storage back-end solutions are used to build high performance infrastructures for machine learning, analytics, and other data intensive workloads. The concept of object storage was introduced in the early 1990's by CMU as an academic research project, since then it has greatly matured and is now largely adopted by the industry [23]. It works by moving lower-level functionalities such as space management into the storage device itself, accessing the device through a standard object interface.

We choose to use MinIO because it is open source, cloud native and designed to be compatible with Amazon S3 API, one of the most used solutions for public cloud object storage, even Microsoft Azure uses MinIO as its S3 Gateway. MinIO supports mutual TLS (mTLS) authentication and also uses HTTPS, it also supports two different types of server-side encryption (SSE), using secret key provided by the S3 client or a secret key managed by a Key Management Service (KMS).

A.5 Summary of Limitations

Considering all applications mentioned above, table A.3 summarizes some of the security limitations and therefore outlines where a secure proxy can help enforcing security measures.

Table A.3: Summary of Security Limitations

Application	Authentication	Access Control	Data-in-Transit	Data-at-Rest	Auditability
MongoDB	mTLS	RBAC	TLS	No*	No*
Metabase	User/Pass	Group-based	TLS	Encrypted	No
Kafka	mTLS	ACL and RBAC	TLS	No	No
MinIO	mTLS	No	TLS	Encryted	No