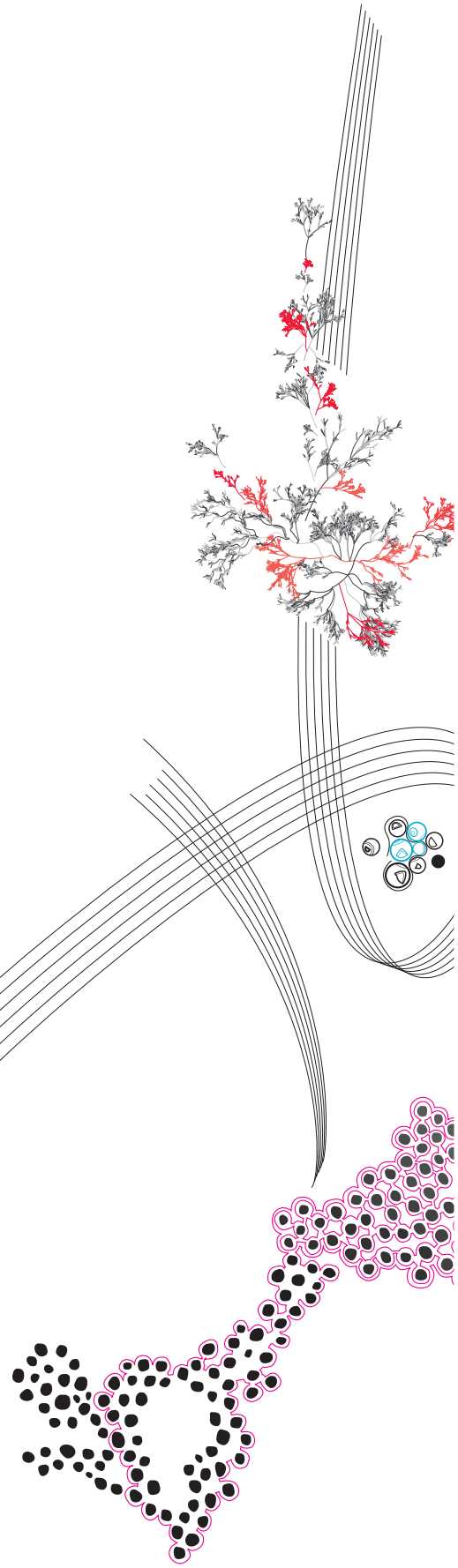

MSc Computer Science
Final Project

Protecting against internal attackers with hardware-aided proxy re-encryption

Martijn Brattinga

Supervisors:
Florian Hahn (UT)
Faiza Bukhsh (UT)
Edmond Varwijk (RDW)
Gert Maneschijn (RDW)

July, 2023

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

Abstract

This research proposes an architecture that eliminates sensitive plaintext data at a trusted service provider. This architecture reduces the impact of data breaches, as they do not involve plaintext data. A typical use-case for the proposed architecture is a service provider which allows authorized third parties to request data from and insert data into a database via an API. The service provider is in control of the data and can use regular SQL functionality on encrypted data, while no plaintext is present on both the API application server and the database server. An Intel SGX trusted execution environment extends the Microsoft Always Encrypted cryptography by re-encrypting sensitive data towards third parties. Results shows that the additional security eliminates plaintext leakage at the price of an acceptable performance impact, demonstrating the feasibility and potential of the proposed architecture in practice.

Keywords: Trusted Execution Environment, Intel SGX, Proxy Re-Encryption, Always Encrypted, Database Encryption, Secure Architecture

Chapter 1

Introduction

While digitization brings many opportunities and advantages, it comes with its risks as well. The HaveIBeenPwned[1] website that catalogues public data breaches contains more than 12 billion breached accounts, some only containing usernames, others including more sensitive data like addresses, phone numbers, and even passwords. Leakage of personal data poses privacy risks, and sales on the dark web [25] indicate that personal information is valuable to criminals or whoever is interested. Businesses are required to protect against data breaches in order to prevent reputation loss or fines up to 10 million Euro or 2% of the firm's worldwide annual revenue from the GDPR[29].

Encryption is a widely used technique to prevent the exposure of sensitive data. A mathematical operation transforms plaintext data into a ciphertext, which only parties with a valid key can decrypt back into the original plaintext. The plaintext value cannot be deduced from properly encrypted data. When the ciphertext leaks without the corresponding decryption key, the sensitive data remains protected.

End-to-end encryption is a type of encryption where data always is encrypted between two endpoints. No intermediary can ever obtain the plaintext data. A common use case is encrypted chat applications[7], where only the sender and receiver possess the decryption keys of their messages. Encrypted data storage is another use case of end-to-end encryption, where the data owner encrypts it's data before storing it in untrusted places. In these applications, the service provider can provide the service without requiring access to the plaintext data.

While end-to-end encryption allows for great privacy, the lack of functionality on encrypted data makes such schemes not applicable to all scenarios. Furthermore, the end-users fully control who can access their data, while sometimes the service provider should have this control instead. For example, we can consider a governmental organization which is entitled by law to have access to its citizens' data. End-users must never be able to circumvent such access. The end-to-end encryption guarantee in which not even service providers can access the data is too strong in this case, as the government as a service provider is and has to be trusted.

This research focuses on the common scenario where a service provider stores data in a database and allows authorized third parties to request that data. The service provider is trusted and in control of the decryption keys, but nevertheless not trusted to be completely secure. Therefore, we require the service provider to only have encrypted data on its servers at all times. Furthermore, we assume that the service provider is capable of storing the decryption keys securely. Various mechanisms exists to support such secure key storage, from cloud based key stores to physical hardware security modules.

[1] https://haveibeenpwned.com

Besides encrypting data on a hard disk (*at rest*) and while being sent over a network (*in transit*), recent technology such as the Always Encrypted functionality of the Microsoft SQL Server (MsSQL) allows data to be encrypted while it is being processed (*in use*) at the database server. While this allows the database server to only have encrypted data, the application server still decrypts all data. The plaintext data and corresponding decryption keys are present on the application servers *in use*, which leaves the service provider an attractive target for attackers.

This research aims to reach a trust boundary where besides the authorized third parties only a secure key management system at the service provider is included. This extends the security provided by technology such as Always Encrypted over the application server, such that the complete service provider has no plaintext on its systems at any time. Figure 1.1 shows our desired architecture with the service provider consisting of an application and a database server. This research proposes an architecture where plaintext data is protected under encryption at all times until an authorized third party obtains the data and decrypts it locally. Such an architecture meets our defined trust boundary.

Our architecture is based on various state-of-the-art components from the literature. Microsoft Always Encrypted with Enclaves is used to allow encryption of sensitive columns in a SQL database. Instead of decrypting the query results at the application server, our architecture re-encrypts the data using a Trusted Execution Environment such that the security guarantees from the database server are extended to the application server. To the best of our knowledge, this paper is the first to combine all these techniques to obtain our desired minimal trust boundary.

The results show that implementing such an architecture is feasible. While the additional security comes at a performance cost, the performance penalty might be justified in scenarios demanding a high level of privacy and security.

The rest of this paper only considers the service provider and the transit to authorized third parties, thus the security of the data once arrived at the third parties is out of scope. If the service provider requires access to plaintext data itself, we separate that part from the service provider and consider it to be an external authorized third party. For instance, a customer service centre is not considered part of the service provider in our terms, but rather a separate third party that can request data from the service provider to do its job, like any other third party. While it may seem like the functionality of the service provider is limited, any functionality that requires access to the data can be considered a third party and therefore is still possible.

1.1 Motivation

When data is encrypted at all times any data breach at the service provider only involves encrypted data. This reduces the impact of data breaches, as leakage of encrypted data is assumed to be less of a problem compared to leakage of plaintext data. End-to-end encryption can bring such guarantees but is not applicable to all scenarios. In scenarios where the service provider should have full control over the data and where functionality such as searching through a database is required, traditional end-to-end encryption schemes do not suffice.

In contrast to end-to-end encryption, which distrusts every party except the end user, our architecture trusts the service provider. The service provider even is responsible for key management, which makes sure that lost user keys or unwillingness to share data do not reduce data availability to authorized parties. While the service provider does not need access to the plaintext data itself, in theory, it has access to the decryption keys and can

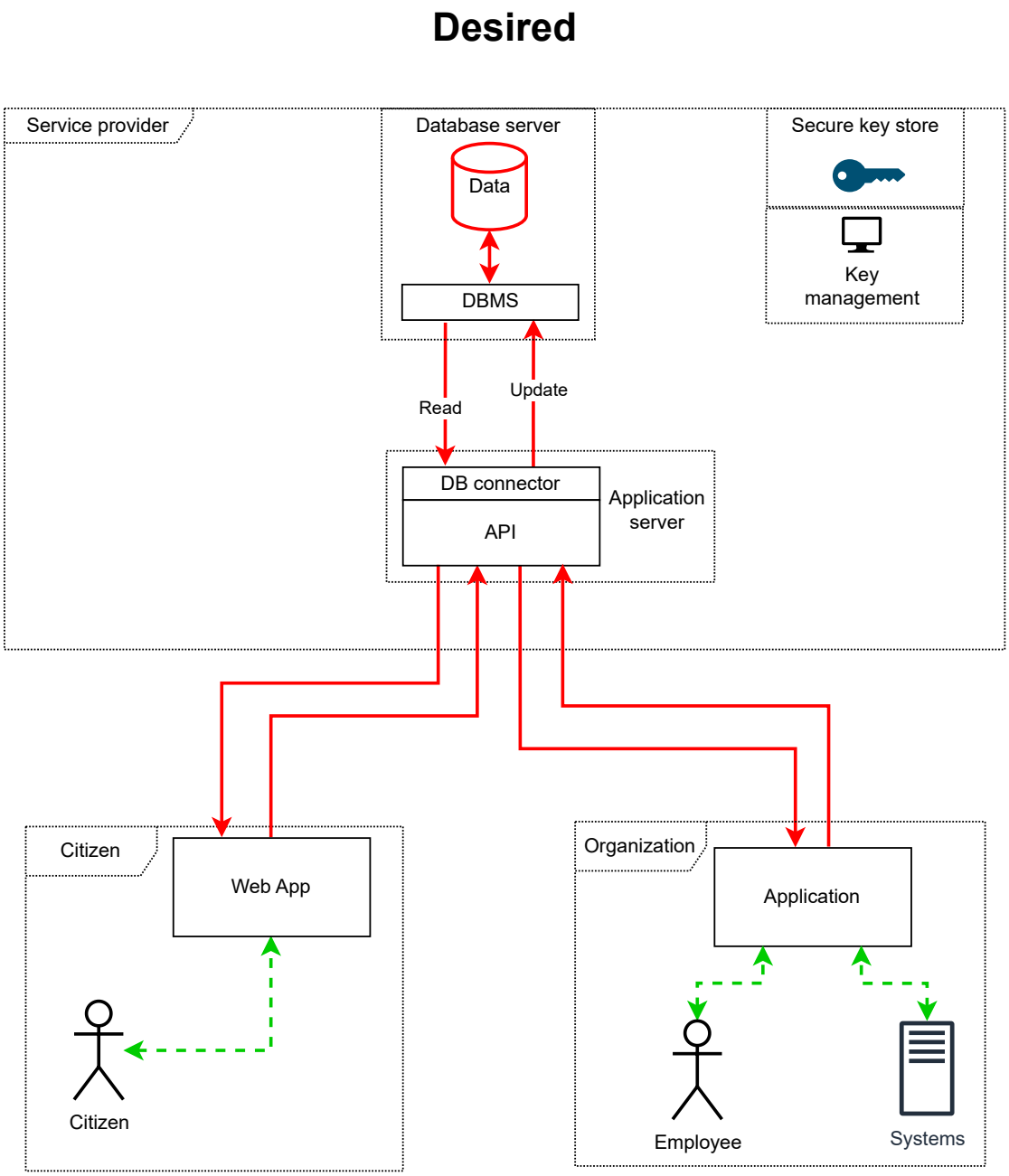

FIGURE 1.1: Architectural overview of the desired system. Red (solid) indicates encrypted data, green (dashed) indicates plaintext data.

provide the data to any authorized third party. Note that trusting the service provider does not mean trusting their systems to be secure, but rather allowing the service provider to securely manage and thus have access to decryption keys. Both the application server and database server are not trusted to be secure. Only a secure key management system is trusted to be secure, to store keys and to transfer encrypted keys towards the servers that require them.

Besides putting less trust in the servers' security, a common use case for database encryption is outsourcing the database to a cloud service provider. Since the database server only handles encrypted data, we do not necessarily have to trust the cloud service

provider. With our architecture, the application server could be outsourced to a cloud service provider as well, as it does not handle any sensitive plaintext data either. The benefits that cloud computing brings can be used to a greater extend, while maintaining privacy and security.

1.2 System requirements

This section sets the requirements our architecture should meet in order to be considered practical in real-world applications. These requirements are divided into security and functional requirements and will be used as a guideline to develop and analyze the prototype. A list of non-requirements that are out of scope is given as well.

As secure system design always is a trade-off between security, functionality, and performance, the requirements have been broadly ordered by importance. Functional requirements up to and including **FR4** and security requirements up to and including **SR2** are the bare minima required to create a system that can show the feasibility of our design. The other requirements should be met in order to consider the system to be practical for a broader range of scenarios. The availability of the data (**FR3**) and the required throughput are of utmost importance for many scenarios, and they should be balanced against the provided security.

1.2.1 Functional requirements

FR0 The system should allow for storage of at least millions of data records, including sensitive and non-sensitive columns

FR1 The system should allow authorized third parties to request and get a subset of the data, at least millions of records per day

FR2 The system should allow authorized third parties to insert new data

FR3 The system should guarantee availability of all data

FR4 The system should support backups and recovery

FR5 The system should allow authorized third parties to update individual data entries under encryption

FR6 The system should require third parties to have to make minimal changes to their IT services, at most to include key management and decryption

FR7 The system should require at most limited changes to applications behind the database connector

FR8 The system should integrate with existing services

FR9 The system should allow underlying components to be changed in the future

FR10 The system should be maintainable by the service provider's ICT employees.

FR11 The system should be measurable in terms of performance

FR12 The system should provide clear error messages without revealing sensitive data

FR13 The system should log data access without revealing sensitive data

1.2.2 Security requirements

SR0 The system should support encryption of sensitive attributes

SR1 The system should not have plaintext sensitive data present on the system

SR2 The system should store decryption keys securely. I.e. no unauthorized party should be able to obtain the decryption keys

SR3 The system should prevent unauthorized parties to learn anything about sensitive data from leaked data

SR4 The system should support key-rollover for all keys used. I.e. all keys must be replaceable on demand.

SR5 The system should be based on underlying cryptography that is at least CPA secure. I.e. attackers that can encrypt known plaintext data should not learn anything from the resulting ciphertexts.

SR6 The system should be developed according to current secure software development best practices, e.g. as defined by NIST[27]

SR7 The system should support cryptographic agility. I.e. the cryptographic primitives must be replaceable

SR8 The system should prevent a persistent attacker at the application or database provider from learning anything about sensitive data

SR9 The system should detect successful attacks that compromise the security

1.2.3 Out of scope

NR0 The system is a prototype that shows feasibility of such a system, and does not have to be production ready

NR1 The system does not have to hide access patterns to sensitive data, as they are deliberately logged

NR2 The system's cryptographic access control does not have to replace the traditional access control, and assumes third parties to be authenticated and authorized

NR3 The system does not prevent data leakage at third parties

1.3 Research questions

During the design and evaluation of the proposed architecture, the following research questions have been answered:

RQ0 How can leakage of sensitive data be mitigated at a trusted service provider?

RQ1 What functional and security requirements should a secure system for sensitive data meet?

RQ2 What cryptographic tools exist and how can they be combined to reach these requirements?

RQ3 How does the proposed architecture compare to a system without encryption?

- **RQ3.1** How does the proposed architecture compare to a system without encryption in terms of functionality?
- **RQ3.2** How does the proposed architecture compare to a system without encryption in terms of performance?
- **RQ3.3** How does the proposed architecture compare to a system without encryption in terms of security?

1.4 Contributions

This research aims to contribute in three ways. First, we present an architecture that combines various state-of-the-art components in order to obtain our desired trust boundary. This architecture can be used as a starting point for further improvements or alternative designs. Second, we implemented a prototype that shows the feasibility of our architecture. The source code can serve as a starting point for refined or alternative implementations. Lastly, the prototype is analysed in terms of functionality, performance, and security. Such an analysis can help decision-makers with the trade-off between functionality, security, and performance that has to be made in all secure software development projects.

1.5 Outline

This section will give a brief overview of the rest of the paper. Chapter 2 presents the various building blocks used in this research, as well as related work. After this, details of our proposed architecture are explained in chapter 3. In addition, this section shows the organizational impact for an organization adopting the architecture. Chapter 4 shows the result of our functionality, performance, and security analysis. This paper ends with limitations and suggestions for future work in chapter 5 and a conclusion in chapter 6.

Chapter 2

Background

This chapter starts with an overview of the trust boundaries briefly introduced in the introduction. After this, section 2.2 introduces Trusted Execution Environments. Section 2.3 describes the technique of Proxy Re-Encryption, and finally encrypted database search is explained in section 2.4.

2.1 Trust boundaries

In order for a system to be considered secure we require that no sensitive data can leak, which can either be achieved by disallowing components to access sensitive data, or by allowing components to access sensitive data but assuming that it prevents any sensitive data leakage. Components that do have access to sensitive data should be trusted to be secure, as insecurity can reveal sensitive data. We define the *trust boundary* of a system as the set of components that need to be considered secure, e.g. which are trusted not to be compromised. If a component is untrusted, thus outside of the trust boundary, it can be compromised without revealing any sensitive data. Therefore, the smaller the trust boundary, the more components can be compromised without affecting the system security guarantees. The following sections shows three variants of trust boundaries, with our desired trust boundary in 2.1.3.

2.1.1 End-to-end encryption

The idea of end-to-end encryption is that only the end-user(s) possess the decryption keys, thus they are the sole owner of their data. Only the users and parties delegated by the users can access the data, which prevents any attacker, service provider, or other man-in-the-middle from accessing the data. As only end users have access to plaintext sensitive data, they are the only ones included in the trust boundary.

Various applications promise to offer end-to-end encryption (E2EE), including the messaging apps WhatsApp and Signal[7] as well as email, calendar, and cloud storage provider Proton[15]. These application are similar to our use case, as they provide an application and store user data. However, the service provider for these applications must not necessarily have access to plaintext data, in contrast to our target scenario. E2EE protects data in transit, at rest, and in use. One should note that computations on encrypted data are often impossible with traditional schemes.

While E2EE provides a high degree of privacy, the technique cannot be applied in all scenarios. For instance, the responsibility for key management including access control and backups lies utterly at the end-user. As long as there are no techniques in place that allow

all citizens, including elderly and non-technical people, to manage keys easily and safely, it is hard to make them responsible for key management. Proton stores private keys of users encrypted under their account password[15], such that as long as a user has its password, it can always recover its private key and access its data. Proton explicitly states that when an account password is lost, only if a recovery phrase or recovery file has been configured [24] data can be accessed. Signal proposed a technique called Secure Value Recovery, that allows encrypted data protection with an easy-to-remember 4-digit pin-code while still preventing brute-force attacks[17]. These techniques make it easier for users not to get locked out of their data. Nevertheless, the final responsibility always lies with the user, which is not desired in our target scenario as this does not provide guaranteed availability (**FR3**). If a user losses their key, the data is lost.

Furthermore, if a user is not willing to share their data with authorized third parties, the data would be inaccessible to them. In the case of a governmental organization storing data about their citizens, both losing data and the ability to withhold data is problematic. The data is not only important to the users but also to all other authorized third parties.

2.1.2 Database encryption

Database encryption allows data in a database to be encrypted while maintaining the required database functionality. Database encryption exists in various flavours and can protect a database in various ways. Transparent Data Encryption (TDE) protects data at rest against theft of physical storage media and attackers that have access to the database systems[9]. More recently proposed solutions encrypt queries before sending them to the database server, which processes the encrypted query on encrypted data, and data is only decrypted back at the application server. In addition to protection at rest, the data is protected in transit between the application and database, and in use at the database server. The database server never deals with any plaintext data and nor has it access to decryption keys. This setting is particularly helpful when data is outsourced to cloud service providers, as they do not have to be trusted with sensitive plaintext data.

The difficulty for database encryption is to perform database functionality, such as searching for specific data, under encryption. CryptDB was the first to present search functionality by using Property Preserving Encryption such as Deterministic Encryption (DET) and Order-Preserving Encryption (OPE)[22]. Liu et. al. use a combination of homomorphic encryption and order-preserving encryption[16]. Poddar et. al. propose Arx, a strongly encrypted database system using searchable encryption[21], while Kamara et. al. create a system that supports a subset of SQL queries using structured encryption[14]. Bajaj et. al. introduced TrustedDB, which uses a Trusted Execution Environment (TEE)[6], an idea which is also used by StealthDB[28] and the latest version of Microsoft Always Encrypted[3], a production-ready database for the industry. The latter is used in our architecture, and further explored in section 2.4.2.

Since the proposed database encryption in the literature does not require plaintext data at the database server, it does not have to be trusted to be secure in order to guarantee the system's security. The application server on the other hand is trusted, as it handles plaintext data and keys. This leaves the application server an attractive target for attackers. This trust boundary, which includes the application server, does not fulfil our requirements (**SR1**). The next section describes our desired trust boundary.

2.1.3 Desired trust boundary

Our desired architecture allows the service provider to only use encrypted data, not only at rest but in transit and in use as well. As the application and database server only handle encrypted data, they are both excluded from the trust boundary. Since the service provider must be in control of the data and decryption keys, it must have a secure key store. This secure key store is assumed to be safe against all unauthorized parties and thus is included in our trust boundary. Actually, this secure key store is the root of trust for the entire system. A Hardware Security Module or a secure key vault, for example, could be used as a secure key store.

Compared to the database encryption trust boundary (section 2.1.2), the key management system instead of the application server is trusted. The former is easier to secure, which improves the architecture's security. The key management system does not need an internet connection and can be completely offline most of the time. Furthermore, the key management system requires less software decreasing the chance of vulnerabilities in the architecture's critical part.

With this trust boundary, any leaked data snapshot at the service provider only contains encrypted sensitive data, and thus unauthorized parties will not learn anything about the sensitive data. This mitigates many common security vulnerabilities such as SQL injection[12] or insider threats. As no plaintext is present at the service provider's servers, no attack can obtain plaintext data. Even persistent attackers with access to the full system cannot deduce any information besides access patterns (**NR1**).

While the service provider does not handle any plaintext data, it has access to the secure key storage and thus in theory could decrypt all data. The application server and database server are excluded from the trust boundary, but the service provider as an organization should be trusted. From a user perspective, the privacy guarantees are weaker compared to end-to-end encryption (section 2.1.1) which distrusts the service provider. However, with current technologies end-to-end encryption is not feasible for every scenario, and the proposed architecture is a great security improvement compared to no encryption or database-only encryption. Table 2.1 shows an overview of the properties of the three discussed trust boundaries.

Component	Desired trust boundary	DB encryption	E2EE
Database server	Untrusted	Untrusted	Untrusted
Application server	Untrusted	**Trusted**	Untrusted
Secure key store	**Trusted**	**Trusted**	-
Authorized 3rd party	**Trusted**	**Trusted**	**Trusted**
Database functionality	Yes	Yes	No

TABLE 2.1: Comparison of our desired trust boundary, database encryption, and end-to-end encryption. The bold components are included in the trust boundary, and are trusted not to be compromised to guarantee the system's security.

2.2 Trusted Execution Environment (TEE)

2.2.1 Overview

A trusted execution environment (TEE) is a tamper-resistant processing environment that guarantees the authenticity of the executed code, the integrity of the runtime state, and the

confidentiality of its code, data and runtime states stored on persistent memory [26]. On a normal CPU, users with root access can inspect both the operations it performs and the data that is in memory. A TEE on the other hand allows operations to be performed while even users with the highest privileges cannot see the data and operations processed within the TEE. While no one can observe computations directly, cryptographic techniques ensure that the enclave is benign. In other words, a TEE can securely perform computations without allowing anyone to see what code and data is used, and while being certain that the TEE does exactly what is desired. TEEs are promising for use in cryptography, as they have performance benefits over pure cryptographic approaches as shown by Sabt et. al.[26] for functional encryption, Zhang et. al. for proxy re-encryption[30], and Fuhry et. al. for searching over encrypted data[10].

Among popular TEEs are Intel's SGX[13][8] and ARM's TrustZone[4]. Intel SGX is further explained in section 2.2.2, as this is the technology used in the proposed prototype. TEEs are comparable to Hardware Security Modules (HSM) but more flexible. TEEs can be programmed by a developer in contrast to an HSM which only allows predefined functionality. Solutions based on TEEs rely on the security of the underlying TEEs. Various attacks exist in the literature against TEEs[20], but research and development of mitigations are active as well.

Instead of performing costly operations on a ciphertext, a TEE can decrypt, then perform operations on the plaintext, and finally encrypt data within its trusted environment. This is an alternative to cryptography that protects data in use. No one can peak in the TEE, not even privileged users or adversaries with physical access to the processor, and thus no one can see the plaintext. This can be considered to have the same security as performing operations directly on a ciphertext, as long as security definitions of the various TEEs available are taken into account. Since decryption, performing an operation, and encryption is usually significantly faster than performing complex operations on ciphertext, TEEs can be promising in our scenario as well.

2.2.2 Intel Software Guard Extensions (SGX)

Intel Software Guard Extensions (SGX) is the Trusted Execution Environment implementation of the Intel CPU manufacturer. Costan et. al. have provided an extensive overview of the SGX architecture[8], of which this section gives a brief summary. The main features include secure memory protection, attestation to ensure enclave authenticity, and sealing to store sensitive data. We chose to use the SGX technology for our prototype, since most servers run on an Intel CPU[1]. Furthermore, the authors develop on Intel based machines which have native support for SGX technology.

With SGX the trusted part of the code is placed and executed in an enclave. This enclave is isolated and thus provides integrity and confidentiality of computations within the enclave, both for the data and code. This isolation is supported by the hardware of the CPU. Whereas normally root users can access individual memory pages, access to the enclave's memory pages is blocked by the hardware. The application's code base is split into a trusted and an untrusted part. The main reason for this division is the limited memory availability of the current SGX generation. Furthermore, the less code in the trusted code base, the smaller the chance of vulnerabilities that can compromise this trusted part. The untrusted code-base can call the trusted enclave functions for sensitive parts of the code. For example cryptography and calculations on the plaintext can be done inside the TEE. The idea is that sensitive data never leaves the trusted part unless protected by encryption.

Using *attestation*, an application can confirm that it communicates with specific software within an SGX enclave. This can be used to cryptographically ensure that the enclave

an application is communicating with is benign. Attestation works by validating a signed hash of the enclave using the manufacturer's attestation service. Intel has to be trusted for SGX's security, as it is key in the attestation process. Attestation works both locally, attesting an enclave on the same machine, as well as remotely towards another machine. When an application sets up a secure channel using attestation to a remote machine, it can send sensitive data to the enclave while it is guaranteed that the untrusted remote hardware cannot observe the sensitive data. The application can use this secure communication channel to exchange secrets with an enclave.

The hardware design of the Intel CPU ensures that no untrusted parts can access the trusted enclave memory pages. Since hardware access checks happen after software access checks, SGX lets the untrusted operating system handle initialization and management of enclave memory page. If the untrusted operating system tries to access trusted memory in a way that is not allowed, the access is blocked by the hardware.

Since the amount of protected enclave memory is limited, a mechanism is designed that allows storage of sensitive memory contents on disk. This feature is called *sealing* and allows enclave memory pages to be evicted to untrusted memory. Before evicting pages to untrusted memory, the content is encrypted. Once encrypted, the memory can safely be stored outside of the secure enclave pages, since any adversaries inspecting the memory can only obtain encrypted information. When the data is needed again, the operating system loads back the encrypted page into the secure memory and decrypts it. During this process the confidentiality, integrity, and freshness of the data are preserved. The SGX design ensures that only the enclave that sealed the memory, or any future versions of that enclave, can decrypt the sensitive memory contents.

Intel SGX enclaves are developed using a low-level programming language, mostly using C++. These enclaves can be called from higher-level languages such as C#. Several Software Development Kits are available, such as Intel's SGX SDK [1], Microsoft's OpenEnclave [2] that support development for multiple underlying enclave techniques, and Fortanix' rust-sgx [3] and Incubator's Teaclave [4] that support development in the Rust programming language.

In order to use Intel SGX for a production system, a commercial license has to be bought from Intel. Without such a license, the hardware memory isolation is disabled, which destroys all the security guarantees provided by SGX. The costs and exact requirements for a license is currently unknown to the authors. Costan et. al. provide critique for this licensing system, as it "allows Intel to force itself as an intermediary in the distribution of all enclave software", while this is not required for the functionality[8].

Nillson et. al.[20] have published a survey on SGX attacks. The impact of the attacks shown in the survey varies, some including the extraction of secret seal keys and attestation keys from the signed quoting enclave. For all attacks included in the survey mitigations have been proposed. While some mitigations are implemented by Intel, others require consciousness during application design. Costan et. al. provided an analysis of several types of vulnerabilities, and conclude that it is not trivial to mitigate against advanced side-channel attacks[8].

[1] https://www.intel.com/content/www/us/en/developer/articles/guide/getting-started-with-sgx-sdk-for-windows.html

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

[3] https://github.com/fortanix/rust-sgx

[4] https://github.com/apache/incubator-teaclave-sgx-sdk

2.3 Proxy Re-Encryption (PRE)

Proxy re-encryption (PRE) allows a proxy to convert a ciphertext encrypted under one key into a ciphertext encrypted under another key, without seeing the underlying plaintext [5]. Such schemes are based on public-key encryption, where each party has a public key and a private key. To encrypt a message, one can take the public key of the receiver to encrypt a message. Only the receiver can decrypt the message, as their private key is required to decrypt a message. If a message is encrypted under A's public key, only A can decrypt that message. Using PRE, A can ask a proxy to transform the ciphertext such that B can decrypt it, without alowing the proxy to see the plaintext. The first party (A) uses their private key to generate a delegation key for the other party's (B) public key, which enables the proxy to re-encrypt a message such that only the second party (B) can decrypt it with their private key. A visualization of PRE can be found in Figure 2.1.

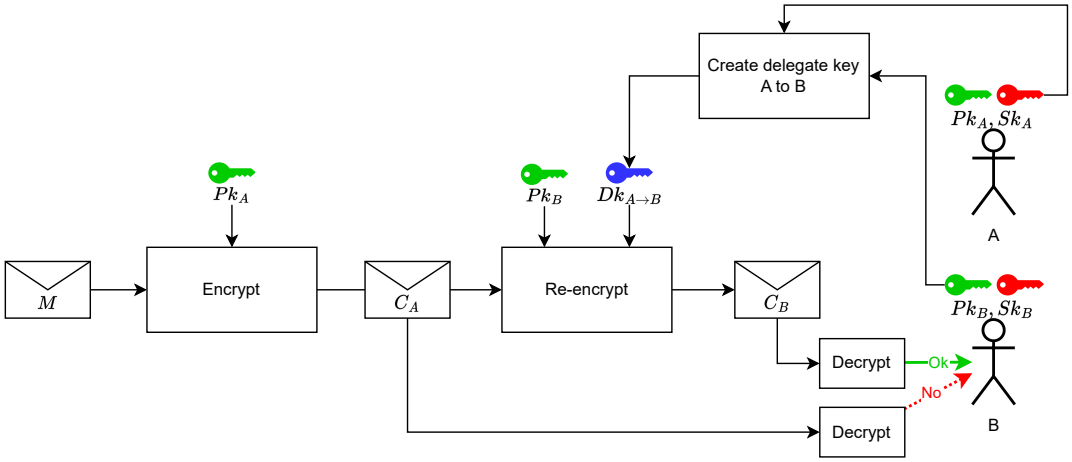

FIGURE 2.1: Proxy Re-Encryption: a ciphertext can be Re-Encrypted without seeing plaintext.

An alternative to public key encryption is symmetric key encryption, which requires all parties involved to have access to the shared private key. If standard symmetric encryption is used to encrypt data in a database, decryption keys have to be given to all third parties to let them decrypt the data. However, now an attacker can simply become a third party, obtain the decryption key, and decrypt all encrypted data. This does not prevent leakage of sensitive data, but only postpones it. A solution would be to encrypt every entry in the database under a different key, such that each third party can only decrypt its own data, however now authorized third parties that require access to many user's data need to store an absurd amount of keys.

PRE allows us to split the data flow from one encrypted data flow into two encrypted data flows with a proxy in-between. The database is encrypted under a private database key. As no third party possesses that private decryption key, they cannot read this data. The application server that interacts with the database can act as a proxy and re-encrypt the data towards the specific third party that requested the data, allowing only that party to decrypt the data.

Zhang et. al. found that TEEs can be used for proxy re-encryption[30]. Instead of having a proxy that re-encrypts from A to B using a delegate key, the proxy uses a TEE to decrypt the ciphertext C_A, and then re-encrypts it with the public key of B into C_B. Assuming the underlying TEE is secure, this satisfies the goal that no plaintext is ever

visible on the systems of the proxy. A visualization of TEE PRE can be found in Figure 2.2.

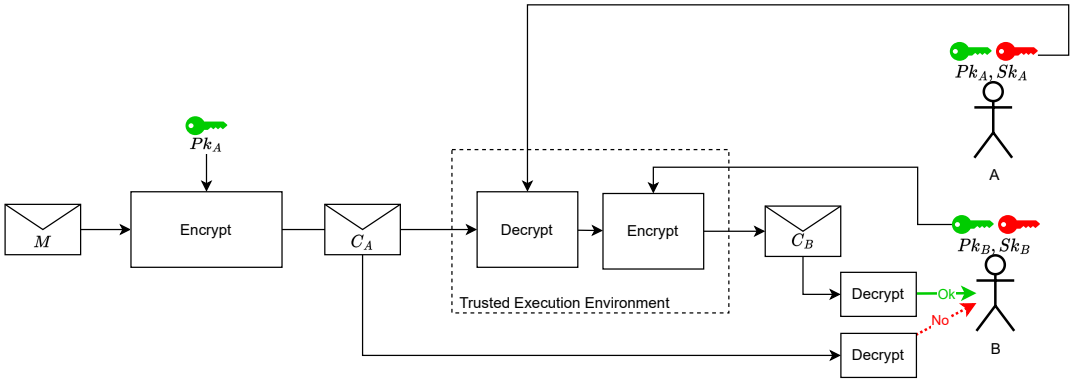

FIGURE 2.2: Proxy Re-Encryption using a trusted execution environment: instead of re-encryption, a ciphertext is decrypted and encrypted within the TEE.

PRE with TEEs is different from cryptographical PRE in several aspects. No delegate keys are involved, but instead, the secret key of the first data flow is used directly in the TEE to perform re-encryption. Furthermore, TEE PRE works with many cryptographic schemes, which can easily be exchanged in the future when one protocol turns out to be insecure. The cryptography used for the first data flow between database and proxy can even differ from the one used in the second data flow between proxy and third party. This property allows for a wide variety of encryption algorithms, which increases the chances of interoperability with the database and third parties and supports functional cryptography such as homomorphic encryption for additional functionality.

2.4 TEE database search

2.4.1 Overview

Various researchers have proposed to use TEEs for searching through encrypted data. Instead of using searchable encryption schemes the TEE decrypts the data and then performs the necessary computations for searching. Bajaj et. al. claim that the computation inside secure hardware processors is orders of magnitude cheaper than equivalent cryptographic operations performed on the provider's unsecured common server hardware[6]. This section shows various proposals from the literature that involve database searching with TEEs. The database search technique used in our proposed architecture is further explored in section 2.4.2.

TrustedDB proposes a trusted hardware-based database where the client transparently encrypts a query, which is offloaded to a Request Handler inside a secure co-processor (SCPU)[6]. This SCPU decrypts, parses, and optimizes the query, after which the private parts of the query are handled by the SCPU database engine, while the public queries are handled by the host server. The final result is assembled, encrypted, signed, and send back to the client by the Query Dispatcher within the SCPU. The performance overhead measured is up to 10x compared to databases without encryption[6].

Priebe et. al. propose EnclaveDB, which as the name implies uses enclaves, the secure parts of Intel SGX, to process database queries[23]. The client establishes a session key with the secure enclave on the server-side and uses this key to encrypt the query parameters.

In contrast to TrustedDB, EnclaveDB does not require additional hardware, as Intel SGX is used as opposed to a special co-processor. Furthermore, EnclaveDB ensures integrity and freshness of computation next to confidentiality, and the system has been designed to keep context switches low to reduce performance overhead.

StealthDB by Vinayagamurthy et. al. uses a TEE as well, but only the primitive comparison operations are outsourced to the trusted part[28]. While EnclaveDB assumes the existence of large enclaves, much greater than existing today, StealthDB is practical to implement on the current generation Intel SGX. Only outsourcing primitive operations to an enclave greatly reduced the size of the trusted code base, counting only 5k lines of which 1.5k running in enclaves. However, this also results in leakage of the plaintext outputs for every primitive comparison operation to the untrusted server.

Fuhry et. al. have proposed EncDBDB, a searchable encrypted, fast, compressed, in-memory database that relies on Intel SGX enclaves[11]. In-memory databases rely mainly on memory instead of hard disks to store data. They focus on range queries, but EncDBDB also supports join, count, aggregation and average calculations. Furthermore, EncDBDB target usage is for heavy read-only usage and is implemented on the column-oriented database Monet, but insertions, deletions and updates are possible as well.

Antonopoulos et. al. have proposed the Microsoft SQL Database supporting Always Encrypted with Secure Enclaves[3]. This is a production-ready database, that supports column-level encryption and uses a TEE to search through the encrypted data. Despite some limitations that can be relevant depending on the application, this system demonstrates the feasibility of TEE search in practice. This database is further explored in section 2.4.2.

2.4.2 Always Encrypted with Enclaves (AE)

Always Encrypted (AE) is a feature of the production-ready Microsoft SQL Server that uses column granularity encryption to provide cryptographic data protection guarantees[3], while maintaining most SQL functionality. The first release of AE used deterministic decryption (DET), which only allowed for equality comparisons. Unfortunately, deterministic encryption turns out to reveal quite some information about the underlying plaintext[19]. Besides the fact that only a small subset of the SQL functionality was available in this first version of AE, the weak cryptography makes it inadequate for our use case.

In the release of Always Encrypted with Secure Enclaves, a Trusted Execution Environment is used at the database engine that can do much more operations, while maintaining the protection guarantees. The application server has an SQL client which transforms the query by encrypting sensitive fields and sends the query to the database server. Decryption keys required are sent over a secure channel from the SQL client to the TEE at the database server. Now the database with the TEE can perform the query operations, and send back the results. Note that all operations on plaintext happen inside the enclave, and thus are invisible to the database server. The on-premise version of AE uses Virtualization-based Security enclaves[18], while the Azure cloud version can use Intel SGX enclaves[8].

Besides the promising features of AE, there are still some limiting factors. Only one encryption scheme is currently supported, namely AES256 with HMAC SHA256. The HMAC is used to distinguish legitimate ciphertext from garbage and does not aim to guarantee data integrity according to the authors[3]. There are more implementation limitations, but their significance depends on the application requirements.

The default setup of AE requires the database connector at the application server to have access to the decryption keys. These keys have to be sent to the TEE at the database provider. As our required trust boundary cannot include decryption keys at the application

server, these keys should either be sent directly from the secure key storage or from a TEE at the database driver. Furthermore, the database driver should not decrypt the data, as this reveals plaintext at the application server. Instead, the data should be re-encrypted. These modifications are the basis for our architecture, decribed in the next chapter.

Chapter 3

Architecture

This chapter gives a detailed description of our proposed architecture and the prototype implementation. The first section provides an overview of the architecture, after which the individual components are explained in detail. After the technical details, this chapter explains the organizational impact of the proposed architecture.

3.1 Overview

We propose an architecture that consists of a REST API at an application server, a database server, and a third party. Upon third parties request the API retrieves data from a database and responds with JSON formatted data back to the user. Requests can both involve retrieving and inserting data.

Figure 3.2 shows an overview of the architecture. The secure key store and key management at the top right are considered secure and trusted, while everything else within the service provider is considered insecure. The third-party organization itself is out-of-scope for this research.

An example scenario for the proposed architecture is a governmental organization that keeps track of its citizens' driver's licenses. The database consists of three tables: *Users*, *DriversLicenses*, and *DriversLicenseCodes*. Each user has a social security number (BSN), and personal information such as first name, last name, birth date, birth place, postal code, and house number. Each driver's license is linked to one single user and furthermore contains the start date, the expiration date, and a penalty points number. Finally, each driver's license can have multiple driver's license codes, which represent the various categories of vehicles a driver's license is valid for. These codes contain an optional text field for extra information. A diagram of the database can be found in Figure 3.1.

For simplicity, only the BSN, last name, and postal code fields are considered sensitive. Therefore, these are the only fields that are encrypted. For simplicity, these fields are encrypted under the same key. The encryption scheme is randomized since random initialization vectors are used per entry.

The architecture is based on the Microsoft SQL Database (MsSQL), which supports Always Encrypted (AE) with Enclaves (described in section 2.4.2). MsSQL is a mature database that supports a great part of the SQL functionality under encryption of sensitive columns (**SR0**). A cloud version is available on Azure. MsSQL supports the required data functionality such as inserting (**FR2**), updating (**FR5**), and retrieving data (**FR1**). With this database the architecture has a trust boundary as described in section 2.1.2. The database server will not be discussed in detail, since it is already described and implemented by Microsoft[3].

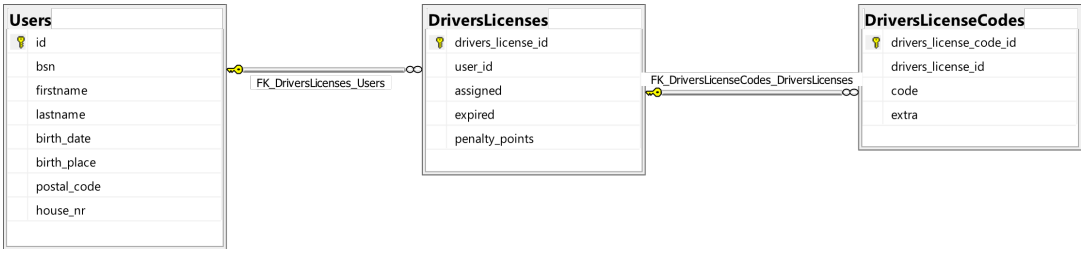

FIGURE 3.1: Database scheme used in our database. The User.BSN, User.lastname, and User.postal_code columns are sensitive and encrypted.

A major modification required in our architecture is the key provisioning to the MsSQL enclave. In the original AE technology the application server provisions the database decryption key to the database server. However, in our design the trusted key management system should provision that database key. Otherwise, the key would be available in plaintext on the application server, which does not satisfy our requirements (**SR1**). The enclave can cache and even store the key using the Intel SGX sealing functionality. Therefore, the key has to be provisioned only on initialization or when keys are rolled-over.

To obtain our desired trust boundary (from section 2.1.3), proxy re-encryption at the application server extends the AE encryption. Instead of encrypting and decrypting data at the application server, the application server merely re-encrypts the data. This makes sure that there is no plaintext on the application server (**SR1**). As explained in section 2.3 a TEE is used for re-encryption. Proxy re-encryption within an enclave is very versatile, as the cryptography between the database and application as well as between the application and the third party can be chosen as desired. This allows insecure cryptography to be changed in the future, or switch databases once other mature databases are developed (**SR7**, **FR9**). Section 3.2.2 explains the modifications made to the database driver in more detail. The prototype has an C# implementation of re-encryption next to the TEE implementation, which is only meant for analysis.

A typical data flow involving both sensitive data in the request and in the response starts at the organization at the bottom of Figure 3.2. The third-party encrypts sensitive request parameters using a randomly generated session key (1). The third-party encrypts this session key as well, using the application server's public key, and sends both the encrypted request and the encrypted session key to the application server (2). The application server calls the TEE to re-encrypt the parameters towards the database (3) and makes an SQL query to the database server with the re-encrypted parameters (4).

The database server uses AE with enclaves to process the query (5) and sends back the encrypted results to the application server (6). The application server now re-encrypts the data inside the TEE using a new session key (7). After this, the application server encrypts the session key under the third parties public key and sends the encrypted data and session information back to the third party (8). Finally, the third party decrypts the session key and uses this key to decrypt the data (9). During this whole process, plaintext is only available within the TEEs, and not anywhere else on the service provider (**SR1**).

As state before in chapter 1, if the service provider as an organization requires access to the plaintext data, we considered that part of the service provider like any other third party. For instance, the data flow of a customer service centre of the service provider is identical to the one described above.

Section 3.2.1 discusses the enclave that re-encrypts the data at the application provider (PREnclave). After this, section 3.2.2 discusses the integration of the PREnclave into the

database driver. Finally, section 3.2.3. explains the integration of this modified database driver into a common DotNet application.

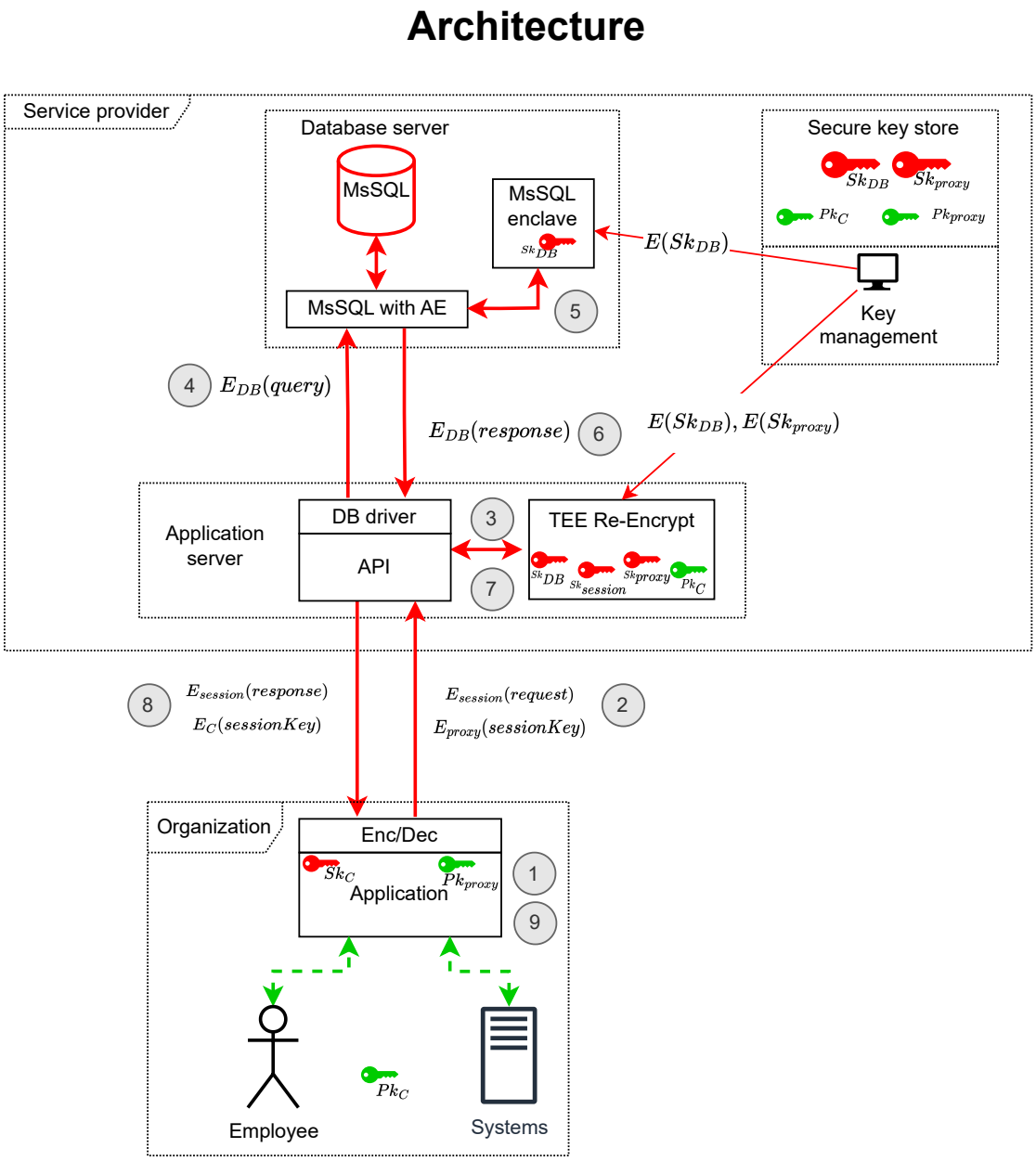

FIGURE 3.2: Architectural overview. Red (solid) indicates encrypted data, green (dashed) indicates plaintext data.

3.2 Components

3.2.1 TEE Re-encrypt (PREnclave)

The PREnclave securely re-encrypts the data from the client to the database server and vice versa. Encrypted request parameters which are re-encrypted towards the database are defined as the *forward* proxy re-encryption, while encrypted response data that is

re-encrypted back to the client is defined as *backward* proxy re-encryption.

The cryptography between the third party and the application server, and between the application and database server both use AES, however, in slightly different ways. The third-party uses AES256 in CBC mode with a random session key and IV to encrypt towards the application server. The third-party encrypts the session key and initialization vector (IV) with RSA2048 with OaepSHA256 padding towards the application server, such that the application server can decrypt them using its private key. While AES and RSA are well-established schemes for exactly this use case, other combinations of schemes are possible as well. For instance, if the cryptography should be changed to quantum secure schemes in the future or key lengths should be extended, this is possible in our design (**SR7**). The encryption between the application server and the database server mainly depends on the support of the database server. Currently, MsSQL with AE only supports authenticated encryption based on AES256 in CBC mode, with an HMAC SHA256 hash. There is no key encapsulation mechanism, so the secret key is retrieved at both the application and the database server from a key store.

The PREnclave is implemented in C++ using the Intel SGX SDK. The the Intel SGX SSL library[1] provides the cryptographic functions based on OpenSLL. To allow passing data from and to the DotNet database driver, two additional dynamically linked libraries (DLLs) are implemented. This is required because DotNet with C# is managed code (i.e. it uses a runtime to manage memory) while the TEE code is written in C++ which is unmanaged code. Furthermore, the enclave bridge functions can only contain C linkages, not C++, and should be 100% native code[2]. As explained in section 2.2.2, no commercial license is obtained from Intel for this research. The enclave is build and tested in pre-release mode, which enables all compiler optimizations, but does not enable the hardware isolation features.

The EnclaveLink is a C++/CLI DLL that is mainly responsible for marshaling data between the managed C# code and the unmanaged C++ code. This DLL consists of both a managed and a native part. When the caller moves byte arrays of data in and out of the enclave, the EnclaveLinkManaged class creates a *pint_ptr* from the given byte array which is passed to the EnclaveLinkNative class. This native class is responsible for calling the enclave bridge functions and for managing the running enclave. On the first enclave function call, it creates a new enclave. Subsequent enclave functions calls make use of the same enclave, and a mutex makes sure that per EnclaveLink instance only one enclave can be active and only one thread can access this enclave at a time. This ensures that one enclave represents one client-enclave and enclave-database session at a time, and that different sessions can not influence each other. When the EnclaveLinkManaged class is not used anymore by the caller, the enclave is destroyed as well.

The EnclaveBridge is a DLL containing pure C code and acts as a bridge between the EnclaveLink and the PREnclave. Besides specifying the interface, all linkages functions are generated by the SGX SDK upon compilation.

The enclave itself specifies functions to set-up the enclave and functions to do the re-encryption. The Enclave Definition Language (EDL) file describes all external callable functions and can be found in the appendix A. Data in and out the enclave is passed by a pointer to an unsigned char array, together with the length of the data. The length is required for the enclave, since all untrusted memory is copied to trusted enclave memory before using the data. All functions return a status code, which indicates success or failure

[1] https://github.com/intel/intel-sgx-ssl

[2] https://www.intel.com/content/www/us/en/developer/articles/technical/using-enclaves-with-callbacks-via-ocalls.html

of the operation.

The enclave requires various keys to operate. For simplicity, we allow the private RSA key and the database AES key to be set from the untrusted environment in our prototype. In a real-world deployment such key provisioning should be done using SGX remote attestation features. Therefore our exact implementation of the key provisioning function is not of interest. Once key provisioning using remote attestation is implemented, key caching using SGX sealing features could be implemented as well to minimize the number of times a secret key has to be transferred. Both the RSA private key of the enclave and the AES database key remain constant until keys are rotated.

Communication from and back to the client uses session keys. We define two sessions analogous to the two re-encryption directions: a *forward* session for communication from the third party to the database, and a *backward* session for communication from the database to the third party. Before the re-encryption functions can be called, the sessions should be initialized. The caller provides the encrypted session key, encrypted session IV, and public key of the client to the enclave. The enclave decrypts the key and IV, and initializes an AES object for the forward communication. The enclave creates a new AES object with a random key and IV for the backward communication. The enclave encrypts the backward session key and IV towards the third parties public key, such that they can be retrieved by the untrusted environment.

The forward proxy re-encryption first decrypts the given encrypted data. Since the plaintext size is unknown before decryption, but always equal or less than the ciphertext size, a buffer of the size of the ciphertext is created. After decryption, this buffer is trimmed to the final size. The enclave then encrypts the plaintext towards the database with the cryptography used by Always Encrypted, which involves generating a random IV, encrypting the plaintext, calculating a MAC, and combining everything. Details can be found in the Always Encrypted documentation[3]. The backward re-encryption is similar, but the other way around. Upon decrypting the ciphertext from the database, the enclave calculates a MAC and compares this hash to the given hash.

3.2.2 DB driver

Our architecture uses the Microsoft.Data.SqlClient database driver, which is a popular driver used in the DotNet framework. The driver already supports Always Encrypted functionality, where parameters are encrypted and results are decrypted at the application server. The source code is mainly written in C# and available publicly[3], which allows extension with our PREnclave. Other database drivers could be used instead, as long as they support AE or similar technology.

With typical usage of the database driver, an application creates a *SqlConnection*, creates one or more *SqlCommand*s, possibly sets *SqlParameter*s, executes the command(s), and reads the results.

When the application creates a *SqlConnection*, a connection string that includes various settings such as the database address, credential type, and security settings is provided in the constructor. The Always Encrypted functionality can be enabled by providing the *Column Encryption Setting=Enabled;* setting. Enabling proxy re-encryption in our modified database driver works in a similar way, by setting the option *Column Encryption PRE Setting=BidirectionalTEE;*. The forward and backward re-encryption has been implemented separately as well, but in order to obtain our desired trust boundary both must be enabled hence bidirectional.

[3] https://github.com/dotnet/SqlClient

When a new *SqlConnection* object is created and the PRE setting is set, a new *EnclaveLinkManaged* instance is created, in turn spinning up a new enclave. Each connection, therefore, contains its own enclave, while multiple commands for this connection share the same enclave. Upon disposing of the *SqlConnection*, the enclave is automatically disposed of as well.

The proposed architecture considers each *SqlCommand* as a separate session. Therefore, each command holds information about the encrypted session keys, encrypted session IVs, as well as the client's public key. The enclave session function is called with this information. With this call the enclave sessions as described in section 3.2.1 are initialized.

When PRE is enabled, the database driver re-encrypts instead of the AE encryption and decryption. This is required because the input and output should not be plaintext, but ciphertexts under the session key instead. Parameter values are re-encrypted using the PREnclave in the *TDSExecuteRPCAddParameter(...)* method before the encrypted parameter is added to the remote procedure call (RPC). The query with encrypted parameters is sent to the database server, which processes the requests and responds with encrypted data. When the application extracts the response data from the *SqlDataReader*, the PREnclave backward re-encrypts the values. These locations are chosen for the re-encryption because this is where original AE code encrypts parameters and decrypts values as well. Therefore, code such as obtaining information about which database key should be used and retrieving that database AES key can be re-used. Note that for the first request, the database AES key is retrieved and provided to the enclave, which is insecure and only used for simplicity as explained in section 3.2.1.

For evaluation purposes, the re-encryption functionality is implemented in both the PREnclave and in C#. When PRE without TEE is enabled, the encrypted parameter is decrypted in C#, and encrypted towards the database using the regular Always Encrypted functionality. Backward re-encryption uses the regular Always Encryption decryption functionality and encrypts towards the third party in C#. Note that this is only helpful for our analysis to compare the overhead of re-encryption in general versus the overhead of TEE PRE. The implementation in C# does not contribute to our desired trust boundary.

3.2.3 API

This section shows how a typical DotNet application can implement the modified database driver and how this differs from implementing the default database driver with Always Encrypted enabled. The design goal of the database driver is to require as little developer effort as possible (**FR7**), but unlike the AE functionality, a completely transparent implementation is not possible. Whereas AE does not require any information from the application server, the PRE version needs to know both the forward session information in order to decrypt parameters, as well as the public key of the third party to encrypt the backward session information. This information is provided by the third party and should be available to the PREnclave, and thus passed through the database driver. Furthermore, the query parameters and return values do not have the original data types but will be byte arrays instead. Finally, applications might perform operations on the plaintext values which are not possible on encrypted data. Databases should be designed with encrypted columns in mind, and applications should be adapted to encrypted database designs.

An example implementation of the modified database driver is shown in Listing 1. The connection string in line 3 includes the PRE TEE setting in addition to the default AE connection string. The public key, encrypted key, and encrypted IV properties are set on the *SqlCommand* in lines 5-7. The query contains one single parameter, which is set in lines 10-12. The column type is explicitly set in line 10, while the encrypted value is set

in line 11.

Upon retrieving the values, the types of the encrypted columns are lost. Line 18,20, and 23 are of type byte array, and use the *GetValue(...)* function instead of the typed get function.

```
1  byte[] enc_symmetric_key, enc_symmetric_IV, pk_client, val;
2
3  using (SqlConnection connection = new SqlConnection(connectionstring +
   ↪  "Column Encryption PRE Setting=BidirectionalTEE")){
4    using (SqlCommand cmd = connection.CreateCommand()){
5      cmd.CommandText = @"SELECT * FROM users WHERE bsn = @BSN";
6      cmd.PREPublicKey = pk_client;
7      cmd.PREEncryptedSymmetricKey = enc_symmetric_key;
8      cmd.PREEncryptedSymmetricIV = enc_symmetric_IV;
9
10     SqlParameter p = new SqlParameter("@BSN", System.Data.SqlDbType.Int);
11     p.Value = val;
12     cmd.Parameters.Add(p);
13
14     connection.Open();
15     using (SqlDataReader r = cmd.ExecuteReader(){
16       while (r.Read()){
17         int id = reader.GetInt32(r.GetOrdinal("id"));
18         byte[] bsn = (byte[])r.GetValue(r.GetOrdinal("bsn"));
19         string firstname = r.GetString(r.GetOrdinal("firstname"));
20         byte[] lastname = (byte[])r.GetValue(r.GetOrdinal("lastname"));
21         DateTime birth_date = r.GetDateTime(r.GetOrdinal("birth_date"));
22         string birth_place = r.GetString(r.GetOrdinal("birth_place"));
23         byte[] postalcode = (byte[])r.GetValue(r.GetOrdinal("postalcode"));
24         string house_nr = r.GetString(r.GetOrdinal("house_nr"));
25       }
26     }
27   }
28 }
```

Listing 1: Example implementation of the modified database driver with PRE TEE enabled.

As mentioned briefly before, our prototype provisions the database key via the original Always Encrypted functionality in an insecure way. Similarly our prototype provisions the PREnclave private RSA key from the application server. In a real-life implementation of the modified database driver, the developer and administrator should make additional effort for key management.

3.3 Organizational impact

3.3.1 Implementation at service provider

Besides technical changes, the implementation of our architecture also has an organizational impact on the service provider. The off-the-shelf MsSQL database and existing applications require little changes (see section 3.2.3), thus maintenance of these parts is not affected much (**FR10**). The database driver on the other hand requires more development and maintenance, which requires a thorough understanding of low-level programming, SGX development, and cryptography. As this is a niche in software development, current knowledge at a service provider's IT personnel might be limited. Furthermore, if the database driver has been customized, every update of the original database driver has to be integrated into the customized version. This is important, especially for security and performance updates, and can be time-consuming. If the original developers of the DotNet database driver would include our proposed additions in the official code-base, the maintenance is completely outsourced and not an issue for the service provider.

Before an organization can implement the proposed architecture, good key management practices should be in place. When the organization already uses Transparent Data Encryption, which provides encryption at rest, key management probably is already in place. However, when no data encryption is used yet, such key management should be designed and implemented. This includes access control to the keys and back-ups. As the database server itself is untouched in our architecture, existing backup technology can be used. Currently Always Encrypted supports encrypted back-ups. Data can be both exported and imported in encrypted or plaintext form. For key management, several options are available such as Hardware Security Modules, or software key vaults.

As described in the introduction, any internal part of the service provider that uses the data is considered a regular third party. Therefore, any internal processes that use the data should be adapted as well. This includes technical applications such as logging and metrics (**FR12**, **FR11**), but also includes business intelligence and customer services. When a customer service requires access the plaintext sensitive data, it should behave as an authorized third party and thus make similar modifications as the other third parties including encrypting requests and decrypting responses.

While the security of a system is improved by implementing the proposed architecture, the functionality and performance might be reduced (see chapter 4 for more details). Whether the additional security out-weights the disadvantages is mainly a business decision. Chapter 4 aims to provide an adequate security, performance, and functionality analysis to support such decisions.

3.3.2 Key management

This section provides an overview of all cryptographic keys used in the proposed architecture, of which a summary is given in Table 3.1.

Always Encrypted requires two keys for every encrypted column in the database. Column Master Keys (CMK) are used to encrypt Column Encryption Keys (CEK), such that the latter can be stored at the database server. The CMKs are stored securely in a separate key store such as Azure Key Vault, Windows Certificate Store, or Hardware Security Modules (**SR2**). Always Encrypted uses AES, which is CPA secure (**SR5**).

Encryption between the service provider and a third party requires a public-key pair for both sides since key encapsulation is used with a random session key for symmetric AES encryption. The session key is encrypted using the public key of the other party,

such that only that party can decrypt the session key. This approach provides forward secrecy and is faster since symmetric encryption generally has better performance than asymmetric encryption. Note that the session key is for one-time use, and does not have to be stored.

A backup (**FR4**) can contain either a plaintext version or an encrypted version of the database. Microsoft SQL server supports backups with encrypted content, and this option has the least risk in terms of unauthorized data access. However, the availability of the data depends completely on the availability of the keys. If the main private key is lost, neither the original database nor the backup can be decrypted, meaning that all data is lost. Measures could be taken, such as backing up the private key as well. Good key management practices include such key backups, for instance encrypting the key and replicating that encrypted version of the key over multiple locations. A plaintext backup cannot be obtained using the regular backup functionality of MsSQL. Instead, the export data functionality should be used that does support plaintext exports of the data. Eventually, the risk of a plaintext backup against the risk of losing the private key should be weighed against each other.

Key rollover is supported for both the AE and the session encryption (**SR4**). Key rollover is important to limit the risk of the leakage of a single key. If a key is compromised, it can be changed (rolled-over), such that the compromised key cannot be used to decrypt the data anymore. Roll-over of the Always Encrypted CMK only requires re-encryption of the corresponding CEKs. Roll-over of the CEKs requires re-encryption of all corresponding columns in the database. During these operations the client sees no downtime[3]. Key rollover for the key between the application provider and 3rd party boils down to changing the 3rd parties key pair and sharing the new public key with the application provider. As for the data encryption session keys are used, they roll over each session by definition.

Key	Owner	Amount	Usage	Rollover
Column Encryption Key (CEK)	service provider	One for each encrypted column	Enc/Decrypt database columns	Re-encrypt DB column
Column Master Key (CMK)	service provider	One for each CEK	Enc/Decrypt CEK	Re-encrypt CEK, provide to TEE
Session key	service provider	One for each data request, volatile	Enc/Decrypt client-service provider data	-
Service provider public/private key pair	service provider	One	Enc/Decrypt session key	Publish to 3rd parties
3rd Party public/private key pair	3rd party	One for each third party	Enc/Decrypt session key	Share to service provider, provide to TEE

TABLE 3.1: Overview of the keys required in our system

3.4 Summary

The proposed architecture extends database encryption using trusted execution environment based proxy re-encryption, such that the data remains encrypted from the database all the way to the third parties. This ensures that no plaintext data is present on either

the database server and the application server, which answers our main research question (**RQ0**) on how leakage of sensitive data can be mitigated at a trusted service provider.

The Microsoft SQL Server with Always Encrypted is used as our database. This database supports SQL functionality search as searching through data, logging, and back-ups, even with encrypted columns. The original AE enabled database driver encrypts query parameters and decrypts query results. However, this results in plaintext data on our application server. Our modified database driver instead re-encrypts query parameters and query results. This re-encryption is performed within an Intel SGX enclave.

Data is protected *in transit* between third parties and the application server using session keys, and between the application server and the database server by the Column Encryption Key. The data data remains encrypted under the CEK *at rest* at the database server as well. Since data is not stored at the application server and the third party is out-of-scope, data protecting at rest at these parts is not applicable. Data *in use* at the database server is possible since AE technology uses a TEE. Data remains encrypted during all operations, and when plaintext data is required, for example when comparing rows on an encrypted value, the data is decrypted within the TEE. Data *in use* at the application server always stays encrypted as well. The data is encrypted under the CEK when received, re-encrypted within the TEE, and encrypted under a session key when formatting the API response.

Chapter 4

Results

This chapter presents the results of our prototype analysis. First, the functionality of the architecture is compared to the requirements from section 1.2. After this the performance of the various components is evaluated. Finally, a security analysis of the architecture is presented.

4.1 Functionality

Table 4.1 shows an overview of the requirements introduced in section 1.2. The majority of the requirements have been met or can be met in future extensions of the architecture. The previous chapter showed which parts of the architecture contribute to which requirements, while this section focuses on the requirements that are not completely met in our prototype.

Updating data (**FR5**) is not implemented in the prototype. However, updating data is a combination of selecting a row and inserting data which both have been implemented and tested separately. Therefore, updating data should be possible in the architecture as well.

Third parties need to encrypt and decrypt all incoming and outgoing sensitive data. While this effort is not negligible, it is the minimum required to eliminate plaintext data at the service provider (**FR6**). To help third parties implement this functionality, a library or example code could be provided. Furthermore, third parties need to have some key management for their asymmetric private key.

The prototype implementation is not easy to maintain for a service provider (**FR10**). The main reason is that a custom version of the database driver is required, that needs to be kept up to date. Furthermore, implementation and maintenance of the enclave requires specialised knowledge of low-level programming and cryptography. If the architecture would be integrated in readily available solutions, just as Always Encrypted is integrated in the Microsoft.Data.SqlClient driver, maintenance would be greatly improved.

Clear error messages (**FR12**) have not been implemented in the prototype. However, the PREnclave returns a status code for each function. These status codes could be formalized to clear error messages. The database driver error system could be extended with additional errors thrown by the custom implementation or the PREnclave.

Logging data access (**FR13**) is not implemented in the prototype. However, all logging solutions that work for the regular MsSQL database setup should also work for the proposed architecture. Since encryption is on a per-value basis, the sensitive values in the logs will be encrypted as well.

The proposed architecture can meet all functional requirements for our system, which proofs the feasibility of the design. The gained security will be analysed in section 4.3, but

first the performance costs are analysed in the next section.

Requirement	Description	Full-filled
FR0	Store data records	Yes
FR1	Request data	Yes
FR2	Insert data	Yes
FR3	Guaranteed availability	Yes
FR4	Support backups	Yes
FR5	Update data	Possible, not implemented
FR6	Minimal 3rd party IT changes	Yes, additional encryption/decryption and key management
FR7	Minimal application server changes	Yes, set encrypted Key, IV, public client key
FR8	Integrate with existing services	Yes
FR9	Modular components	Yes
FR10	Maintainable system	Partly
FR11	Measurable performance	Yes
FR12	Clear error messages	Possible, not implemented
FR13	Log data access	Possible, not implemented

TABLE 4.1: Functional requirements overview for the proposed architecture.

4.2 Performance

An important factor that contributes to the decision whether or not to implement security measures is the overhead they impose. This section describes performance tests of the various components involved. These tests are performed at three levels: at the PREnclave, at the SQL driver that uses the PREnclave, and at the demo API that uses the modified SQL driver.

All tests are performed on a HP ZBook Studio G5 with an Intel Core i7-8750H 2.20GHz CPU and 16GB RAM. The computer runs Windows 10 Home 22H2 64-bit. Both the database and the application server run on the same device. A typical deployment of our architecture would be on specialized server hardware, and therefore results of the performance analysis should not be interpreted as absolute values. However, comparison of the various tests with and without our security components as described in the next sections can give valuable insights in the feasibility of our architecture. Both our implementation and evaluation do no not use multi-threading, and during our tests both CPU and memory usage were well below system limits. An advantage of running both the database and the application on the same device is that any network latency is excluded from the analysis.

For all tests that involve the database, our database scheme introduced in Figure 3.1 is used. All queries are performed using various database sizes (1000, 10.000, 100.000, 1.000.000, 10.000.000). For tests that are not significantly influenced by the database size only the results that correspond to the largest database size of 10 million records are shown, which directly shows compliance with requirement **FR0**.

The database is initialized with randomly generated data. Each user contains exactly one drivers license with one drivers license code. The first name, last name, birthplace, and postal code are random strings of length 6, 8, 10, and 6 respectively. The ID is auto

generated, and the BSN is i for the ith row. The house number is a random number below 1000, and the dates are randomly generated dates. The driver's license penalty points is a random number between 0 and 2, and the driver's license code is a random single character.

All tests that are dependent on execution time are executed with a minimum of 1 data item or row, and a maximum of 1000 items or rows. Based on our example scenario we assume that most requests will not contain more than 1000 rows or re-encryptions.

4.2.1 PREnclave

The goal of performance testing the proxy re-encryption enclave is to measure the overhead of re-encryption in isolation. Besides measuring the TEE implementation, similar functionality in C# is implemented such that the two approaches can be compared. In other words, this analysis shows the additional overhead of re-encryption in a TEE compared to re-encryption in untrusted code.

The benchmarks are made using the BenchmarkDotNet[1] library. All re-encryption functions of the TEE are called from C#, such that the context switching overhead is included.

The PREnclave contains both functions that are used for setting-up the enclave and functions to do the actual re-encryption. The functions that perform the re-encryption are measured using various input amounts. In this way, the performance of batch operation is compare against other batch sizes. Furthermore, a full PREnclave invocation including setting up the entire enclave, re-encrypting towards the database, and re-encrypting back towards the client is measured, since this represents the real-world usage of the PREnclave.

Set-up

The execution time of the one-time enclave functions are shown in Table 4.2. Creating an enclave takes the majority of the set-up time, with an average of 31.6 ms. Initializing a new forward and backward AES session for communication between the third party and the application server adds another 5.9ms. Retrieving the encrypted AES key and IV from the enclave only entails passing data from the TEE to the managed code, which is a sub 1 ms operation. Setting and retrieving the private RSA key are only implemented in our prototype for simplicity, and in a real-world deployment other mechanisms such as SGX remote attestation should be used. Furthermore, these operation only happen infrequently as the keys can be cached by the enclave. Therefore, the results are of less importance. The total column is the time required to set-up and initialize an enclave, thus excluding the time that setting and getting the RSA key takes.

An enclave can be set-up on average under 37 ms, and 95% all tests created and initialized an enclave under 42ms. Since in C# no enclave has to be created, the same functionality in untrusted code is significantly faster. However, if we assume an enclave is already created and provisioned with the long-term keys, setting the AES session takes takes about 5.9 ms on average.

Re-encryption

Table 4.3 shows the execution time of forward proxy re-encryption, backward proxy re-encryption , and a full PRE invocation for various amounts of data. The latter contains the set-up execution time as explained above in addition to a forward and backward re-encryption. The # data column shows how much data entries are re-encrypted in the test.

[1] https://github.com/dotnet/BenchmarkDotNet

Functionality	Mean TEE	Mean C#	P95 TEE	P95 C#
Create enclave	31.6	0	33.3	0
Create AES session	5.9	2.5	7.7	3.3
Retrieve encrypted AES session	0.2	0.3	0.2	0.3
Set private RSA key	0.1	2.7	0.1	3.5
Get public RSA key	0.1	77.3	0.1	120.2
Total setup	37,9	2,8	41.2	3,6

TABLE 4.2: PREnclave set-up performance comparison between C# and TEE (in ms)

The results show an increase in runtime with an increase of re-encryption invocations. This makes sense, because the cryptographic operations are the most expensive, and they are invoked more often when more data is re-encrypted.

The time it takes to compute a single forward or backward PRE is sub 0.1ms, excluding any enclave initialization. The backward re-encryption is slightly faster than the forward re-encryption. This can be explained by the forward encryption being a bit more complex, as a new random IV has to be created, and the ciphertext for the database has to be composed in a specific format. Furthermore, upon decrypting the client value, the original plaintext value is obtained.

For an SQL query, each parameter and each encrypted column value is re-encrypted separately and counted as 1 data entry. For example, an SQL query with 2 parameters only has two forward re-encryption, which takes about 1ms without setting up and initializing the enclave. If a table with 3 encrypted columns is returned, each returned row requires 3 backward re-encryptions. One such row requires 0.1ms to re-encrypt backwards. The main overhead for a full run lies in creating an enclave and initializing the enclave. As Table 4.2 shows this takes about 37.9 ms altogether, which is the majority of the mean execution time for PRE full runs.

Functionality	# data	Mean TEE	Mean C#	P95 TEE	P95 C#
PRE forward	1	0.1	0.1	0.1	0.1
PRE forward	10	0.4	0.2	0.5	0.2
PRE forward	100	3.3	1.3	3.7	1.8
PRE forward	1000	31.7	9.4	35.8	12.0
PRE backward	1	0.1	0.1	0.1	0.1
PRE backward	10	0.3	0.1	0.4	0.2
PRE backward	100	2.2	0.8	2.3	0.9
PRE backward	1000	22.6	7.3	27.1	7.7
PRE full run	1	37.7	6.3	39.6	7.3
PRE full run	10	38.5	7.4	41.6	8.4
PRE full run	100	42.4	11.0	44.2	13.2
PRE full run	1000	108.5	31.4	136.4	46.3

TABLE 4.3: PREnclave proxy re-encryption performance comparison between C# and TEE (in ms)

4.2.2 SQL driver

The goal of performance testing the SQL driver is to determine the performance difference between a plaintext query, a query that uses AE, and a query that uses TEE PRE.

Various types of queries are measured where the queries target only plaintext columns, only encrypted columns, or both. Not all types of queries are available in all scenarios. For example, any query targeting encrypted columns in a *WHERE* clause is unsupported by the default plaintext SQL driver. With the unconditional *SELECT* of an encrypted columns the plaintext SQL driver just obtains the encrypted byte values to allow for some sort of comparison. However, insertion of plaintext data is not possible on encrypted columns and therefore INSERT queries without AE is excluded.

SELECT

A simple select query allows us to compare a query that requires backward re-encryption with a query that does not require any re-encryption. To measure the execution time of a query involving three plaintext columns the following query is executed:

```
SELECT TOP(@Limit) id, firstname, birth_place FROM users;
```

A similar query selecting three encrypted columns is executed as follow:

```
SELECT TOP(@Limit) bsn, lastname, postal_code FROM users;
```

The execution time of both queries can be found in Table 4.4. The limit value shows how much rows are returned per query which influences the amount of data that is re-encrypted backwards. As the queries do not contain any sensitive arguments, no forward re-encryption is performed.

Selecting three plaintext columns does not involve any re-encryption. Therefore, results for the three versions should be similar. When AE is enabled, the application server makes an additional round trip to the database to request if the query contains any encrypted columns, to which the answer will be negative. This round trip overhead cannot be deduced from the results. This is probably because there is no network latency in our local set-up. The results show an additional 31.8 ms for the TEE version compared to the other implementations, which is unexpected. This additional execution time can be attributed to the creation of an enclave. Our prototype does not check whether a PREnclave is required, but it creates a PREnclave anyway. The developer can prevent this by excluding the PRE setting in the connection string for queries that do not need it. Alternatively, the database driver can use the information about encrypted columns for the specific query before deciding whether to create a PRENclave or not. When no enclave is created when it is not needed, the query time is reduced by 31.6 ms (see Table 4.2) from 32.0 to 0.4 ms for a single row, and from 33.4 to 1.8 ms for 1000 rows. Microsoft has similar recommendations for excluding the AE setting from the connection string if the developer is sure that the query does not involve any encrypted columns, as this prevents an additional round trip to the server.

The second query targeting encrypted columns takes longer than the query targeting plain columns, except for the plain implementation. This makes sense, because for TEE and AE additional cryptographic operations are performed. Especially for a query that returns one row, the average overhead per row for the TEE version compared the the plain version is significant. The setup costs of the enclave are mainly responsible for the significant overhead, as shown in section 4.2.1. When multiple rows are processed, the

setup costs can be amortized over the total processing time as they only happen once for each *SqlConnection*.

Queries targeting encrypted columns can be further optimized to reduce execution time. Instead of creating a new enclave each time a new *SqlConnection* is created, the database driver can manage a pool of enclaves that stay online. When enclaves do not have to be created and initialized for each new *SqlConnection* the query time can be reduced by 31.6 ms (see section 4.2). Querying a single row then requires 14.1 instead of 45.7 ms, and selecting 1000 rows take 68.6 instead of 100.2 ms. As each session still requires re-encryption and an AES session to be set-up, there is overhead compared to the always encrypted version.

Query	# rows	Mean TEE	Mean AE	Mean plain	P95 TEE	P95 AE	P95 plain
SELECT 3 plain columns	1	32.0	0.2	0.2	32.3	0.2	0.2
SELECT 3 plain columns	10	32.1	0.2	0.2	33.0	0.2	0.2
SELECT 3 plain columns	100	32.5	0.6	0.6	32.7	0.6	0.6
SELECT 3 plain columns	1000	33.4	1.5	1.5	33.6	1.5	1.5
SELECT 3 encrypted columns	1	45.7	0.2	0.2	46.5	0.2	0.2
SELECT 3 encrypted columns	10	46.2	0.4	0.2	46.7	0.4	0.2
SELECT 3 encrypted columns	100	51.2	1.8	0.6	51.6	1.8	0.6
SELECT 3 encrypted columns	1000	100.2	14.9	2.2	101.3	15.3	2.2

TABLE 4.4: Comparison between a *SELECT* query targeting plaintext or encrypted columns, for various amount of rows and the three implementations (in ms)

SELECT WHERE

By introducing an additional *WHERE* clause, a query that requires bidirectional re-encryption can be compared to a query that only requires backward re-encryption. To measure the execution time of a query with a plaintext *WHERE* clause the following query is executed:

```
SELECT * FROM users WHERE id = @Id;
```

A similar query targeting an encrypted column in the *WHERE* clause, thus requiring forward re-encryption, is executed as follows:

```
SELECT * FROM users WHERE bsn = @BSN;
```

The execution time of both queries can be found in Table 4.5. The results are similarly to the *SELECT* queries without a WHERE clause, where the TEE version also has quite some overhead. The query with an encrypted *WHERE* clause is a bit more expensive than the query with a plaintext *WHERE* clause. This can be explained by the fact that the encrypted *WHERE* clause requires the database to search through an encrypted column, which involves Always Encrypted enclave computations.

Since it is possible to generate an index on an encrypted column, the overhead for searching through the encrypted column is limited. Table 4.6 shows a comparison between queries where the encrypted WHERE clause has or has no index for various database sizes. The difference between 0.5 and 0.6 ms for the indexed version can be explained by slight variations in performance and rounding to one decimal.

Query	Mean TEE	Mean AE	Mean plain	P95 TEE	P95 AE	P95 plain
WHERE plain	45.0	0.2	0.2	45.7	0.2	0.2
WHERE encrypted	45.7	0.5	x	46.4	0.6	x

TABLE 4.5: Comparison between a SELECT query with a plaintext or encrypted column in the WHERE clause, for the three implementations (in ms). The WHERE column is indexed.

Query	DB size	AE (indexed)	AE (no index)
WHERE encrypted	1,000	0.5	6.8
WHERE encrypted	10,000	0.6	82.3
WHERE encrypted	100,000	0.6	611.5
WHERE encrypted	1,000,000	0.5	2510.0
WHERE encrypted	10,000,000	0.5	21634.1

TABLE 4.6: Comparison between a Always Encrypted SELECT query with an encrypted column in the WHERE clause, for various database sizes both with and without an index (in ms)

JOIN

A common operation on relational databases is joining multiple tables. A query that joins two tables on a plaintext column is executed as follow:

```
SELECT TOP(@Limit) * FROM users JOIN driversLicenses ON
↪  Users.id=driversLicenses.user_id;
```

This number of rows returned is limited. The execution time has similar performance to the *SELECT* encrypted columns query. Results are shown in Table 4.7. The additional join operation that AE performs has limited impact compared to the *SELECT* encrypted columns query. A few ms required for this *JOIN* can be observed in all three implementations. In our experiments the *JOIN* is performed on a plaintext column. For a *JOIN* on an encrypted column, additional overhead for performing the table searches within the AE TEE might be present. However, section 4.2.2 shows that this overhead is sub 1 ms.

Query	Limit	Mean TEE	Mean AE	Mean plain	P95 TEE	P95 AE	P95 plain
JOIN	1	45.6	0.4	0.3	46.1	0.4	0.3
JOIN	10	46.6	0.6	0.4	47.2	0.6	0.5
JOIN	100	51.8	2.3	0.9	52.9	2.3	0.9
JOIN	1000	104.4	18.4	5.7	106.1	18.7	5.8

TABLE 4.7: Comparison between the various implementations for a JOIN query for various row limits (in ms)

INSERT

In order to meet requirement **FR2**, the architecture supports inserting data. An insert query is executed as follows:

```
INSERT dbo.Users ([BSN], [firstname], [lastname], [birth_date],
↪  [birth_place], [postal_code], [house_nr]) VALUES (@BSN, @FirstName,
↪  @LastName, @BirthDate, @Birthplace, @PostalCode, @HouseNr); SELECT
↪  CAST(scope_identity() AS int);
```

The values used for insertion are randomly generated, similar to those used for database initialization. Execution times for *INSERT* queries are shown in Table 4.8. As one might note, the execution time is quite extensive. Inserting 1000 rows takes almost half a minute using the TEE approach. A theoretical estimation would assume an additional overhead of 3000 PRE forwards (1000 rows, 3 columns per row) and the PREnclave set-up time. According to tables 4.2 and 4.3 this should take less than $31.7 * 3 + 37.9 = 133$ ms extra compared to the AE version. Not only the TEE version scales linearly in time, this increase is also visible in the AE version.

Query	# Rows	Mean TEE	Mean AE	P95 TEE	P95 AE
INSERT	1	55.8	1.1	56.4	1.2
INSERT	10	316.7	7.8	320.1	8.6
INSERT	100	2857.5	72.8	2909.8	76.9
INSERT	1000	28173.8	676.0	28358.9	694.0

TABLE 4.8: Comparison between the various implementations for a INSERT query with various amounts of rows inserted (in ms)

4.2.3 Demo API

The goal of the final performance tests is to measure the impact of the proposed architecture in a realistic scenario. Besides the performance overhead of the modified SQL driver, the web API modifications are consideredd as well. For instance, the session keys should be included in the request and the response. The response time of the API will be measured using JMeter[2]. Traditional plaintext queries that use the default Always Encrypted functionality are compared to encrypted queries and responses that uses the modified SQL driver. A prototype without AE enabled is omitted, as such a prototype cannot give the desired functionality for our demo API.

Table 4.9 shows the response times of various queries to our demo API server. Since all services are deployed on a single machine the response times do not include any network latency.

Three types of API requests are executed. The first request retrieves a user by their BSN. This query contains both sensitive (encrypted) data in the query and in the response. This is similar to the *SELECT WHERE encrypted* query. The second request performs a join over three tables and only contains sensitive columns in the response. This is a combination of the *SELECT WHERE encrypted* query and the *JOIN* query. The third request is a join over three tables and contains a sensitive column in the response.

The AE version responds within at most 2 ms to the requests. The TEE version takes 43-56 ms longer, which is expected given the results from the previous section. The third requests, getting a drivers license by the corresponding users BSN has the longest response time. This is expected as this query requires both forward and backward re-encryption, a join over multiple tables, and searching through an encrypted column.

[2] https://jmeter.apache.org

While the TEE version is significantly slower than the TEE version, in certain scenarios the additional performance overhead might be acceptable. Especially with further improvements as suggested in section 4.2.2, TEE proxy re-encryption in combination with technology such as Always Encrypted is feasible in real-world scenarios.

Request	Mean TEE	Mean AE	P95 TEE	P95 AE
Get user by BSN	54	1	66	2
Get driverslicense by ID	44	1	54	3
Get driverslicense by BSN	56	2	70	4

TABLE 4.9: Average response time comparison between the Always Encrypted and the TEE PRE version of requests to a demo API (in ms)

4.2.4 Summary

The performance tests show that our architecture has quite some overhead compared to the plaintext and AE versions. The majority of this overhead comes from initializing the PREnclave, which especially in the case of a single re-encryption is relatively time consuming. As discussed in 4.2.1, the set-up costs could be reduced by having enclaves ready to go in a managed pool of enclaves. In our demo API, the performance is well below 60 ms which can be considered adequate for many real-world scenarios. There is some performance overhead to consider, but for various applications the additional gained security might justify this sacrifice. Furthermore, with the increase of processing power and TEE performance, the performance overhead could be further reduced in the future.

4.3 Security

This section explores the additional security various sub parts of our architecture provide. The communication between third parties and the application server is assumed to be encrypted in all scenarios, i.e. API request are made over HTTPS. We start with a scenario where the data is only encrypted at rest, which is common nowadays. After this, the AE functionality is added which protects the database server. Finally, proxy re-encryption is added to come to our proposed architecture. The security issues at the database server that are present in the Always Encrypted scenario are also present in the final scenario, as the database encryption part is provided by Always Encrypted.

Various attackers are considered, including an attacker with access to the application server, an attacker with access to the database server, an attacker with privileged access to either of these, and an attacker that watches network traffic at the service provider. Any malicious database administrator or server operator is also considered, which might be helpful in scenarios where certain parts of the architecture is outsourced to an untrusted cloud service provider.

4.3.1 Initial scenario

If no data is encrypted, access to the database server, both physical and digital, allows an attacker to copy all data. Transparent Data Encryption (TDE) is often used to encrypt data at rest, mitigating theft of data from the files system or physical disks. However, TDE does not protect in any way against attackers who can query the database, as the

encryption is transparent to database queries. As decryption keys are present in-memory at the database server, any attacker with privileged access can dump these keys from memory and steal the encrypted data with it. Any malicious database administrator has access to all data, thus outsourcing to an untrusted cloud service provider is not secure.

Always Encrypted allows the database server to be removed from the trust boundary. The database server only contains keys and plaintext data within the Trusted Execution Environment, which is assumed to be secure. Not even a privileged attacker or database administrator can access plaintext data, as there are no keys or plaintext data in memory and the administrator has no access to the keys either. However, if encrypted columns contain an index, this index reveals information about the order of rows. The order of the underlying plaintext values is preserved in the index, and thus leaked.

As traffic between the application server and the database server is not encrypted, any attacker listening on the network can watch all queries and responses in plaintext. Sometimes it is possible to encrypt queries and responses, e.g. MsSQL supports TLS for queries and responses. AE additionally encrypts sensitive query parameters and response columns.

An attacker with access to the application server could perform SQL queries, if the attacker has obtained similar permissions to the running applications. This would results in plaintext retrieval of all data. A sophisticated privileged attacker could see all data and keys going through memory before being send to third parties.

4.3.2 Proposed architecture

When a trusted execution environment is used for re-encryption, such as our proposed architecture describes, no plaintext data or keys are present at the application server outside the TEE. Assuming these TEEs are secure, no attacker at the application server can access any plaintext data. Not even privileged attackers or server administrators have access to the plaintext data. The entire database server also does not have to be trusted, as explained by the Always Encrypted security guarantees above.

The only part that is trusted in our architecture is the key management system. This part of the architecture serves as a root of trusts, and attests the TEEs before providing the keys. Compromise of the proxy RSA keys allows an attacker to decrypt any third parties request. Compromise of the database keys allows an attacker to decrypt sensitive database columns.

As long as the TEEs are secure and the key management system is not compromised, no attacker on the service provider is able to obtain any plaintext data.

4.3.3 Cryptography

The system uses AES in CBC mode, and RSA as cryptographic schemes. Both schemes are used in the industry for quite some time already, and up till now are considered secure. The encryption between the application server and database server could be changed by Microsoft in the feature if this is required. The encryption between any third party and the application server could also be changed in the future if required. The minimum requirement of CPA security (**SR5**), where an attacker cannot learn anything about the cryptography while he is able to craft ciphertext of known plaintexts, is met for all encryption used.

4.3.4 Summary

The security analysis is summarized in Table 4.10. Most of the requirements have been met. The proposed architecture assumed that TEE are secure and that the service provider can securely store private keys. Secure software development best practices have been followed where possible, but the implementation merely serves as a prototype. For real-world deployments the exact implementation should be reviewed by experienced security experts.

The cryptography used can be exchanged by other schemes. However, the cryptography between the application server and the database depends on the supported encryption by the database. Currently only one scheme is supported.

As shown in section 4.3.1, order information can leak when an index exists on the encrypted column. Therefore, the requirement that a persistent attacker does not learning anything about sensitive data is not entirely met. Such an attacker could learn information about the order of records with an encrypted index.

Detection is not implemented specifically in the proposed architecture.

Requirement	Description	Full-filled
SR0	Encrypt sensitive attributes	Yes
SR1	No plaintext	Yes, assuming TEE security
SR2	Securely store keys	Assumed
SR3	Unauthorized parties learn nothing	Yes
SR4	Key rollover	Yes
SR5	CPA security	Yes
SR6	Secure software development	Partly, prototype only
SR7	Cryptographic agality	Yes, partly depended on Microsoft
SR8	Against persistent attacker	No
SR9	Detection	No, not implemented

TABLE 4.10: Security requirements overview for the proposed architecture.

Chapter 5

Discussion & future work

This chapter starts with a discussion on our research. This includes our achievements and contributions. After this, limitations of our research, implementation, and evaluation are presented. Finlay, this chapter concludes with suggestions for future work.

5.1 Discussion

Our results show that it is possible to design an architecture that meets the requirements (Section 1.2) drafted for a scenario in which a service provider stores and manages data for third parties. Furthermore, the performance analysis on our implemented prototype shows that such an architecture has an overhead in the order of tens of milliseconds, which could be acceptable in various scenarios considering the gained security.

This research shows that the use of Trusted Execution Environments for eliminating plaintext sensitive data at a service provider is feasible and promising. Previous research showed how TEEs can be used to implement database search [3] and proxy re-encryption [30], and this research in addition shows how these components can be combined to compose a secure architecture. Before the use of TEEs, such an architecture was either extremely slow, or limited in functionality. We showed how the security of an architecture can be improved at a feasible performance costs and with the preservation of functionality.

Our results once more show the potential of TEEs and open a new direction in secure architecture design. This potential makes TEE-based technology more attractive, which can stimulate research in the direction of TEE technology, TEE database search, and architecture design based on TEEs. Furthermore, the industry can continue developing TEE based technology as use-cases and with that the potential customer base keep expanding. Our advance in secure architecture design is valuable in a digitizing world where protection of sensitive data becomes more and more important.

5.2 Limitations

5.2.1 Architectural limitations

The proposed architecture aims to meet all requirements from section 1.2, however some limitations cannot be circumvented.

The design is based on MsSQL Always Encrypted with Enclaves, which has some limitations in functionality. Clustered indexes cannot be created on encrypted columns, and encrypted columns cannot be primary key or referenced by foreign key or unique key constraints either. For example, the social security number (BSN) column cannot be

defined as unique in the demo database (Figure 3.1). Encrypted columns cannot have the IDENTITY property, have default or check constraints, or be of the types *xml, timestamp, rowversion, image, ntext,text,sql_ variant,hierarchyigeography,geometry* or any user defined type. Furthermore, in-memory tables and computed columns do not support encrypted columns. A complete overview of all Always Encrypted limitations can be found in the documentation [1].

The hardware used for the application server and the database server should support TEE technology, with in particular Intel SGX for our prototype. Modern Intel processor have support for SGX, and cloud service providers such as Microsoft Azure also support SGX enabled virtual machines. However, being limited to particular hardware can cause vendor lock-in effects. Moreover, the requirement of a commercial license to use Intel SGX makes a company dependent on Intel. Deployment pipelines such as Azure App Services do not support Intel SGX. Compared to existing deployment pipelines, deploying to virtual machines or dedicated hardware requires more effort in both deployment and maintenance.

The proposed architecture's security heavily depends on the security of the underlying TEEs. The underlying cryptographic schemes can easily be exchanged, but TEEs lay at the heart of the architecture. The TEE security considerations shown in section 2.2 should be taken into account.

Compared to traditional approaches the proposed system makes extensive use of cryptography, which makes key management an important topic. When keys are lost, all data is lost as well. Therefore, good backup strategies for both data and keys are of utmost importance.

While the sensitive columns are encrypted with randomized encryption, the order of rows can be leaked. Especially if such a column contains an index, this index could be scanned to find determine the order of rows.

Lastly, the proposed design relies on the existence of a database that supports enclave technology to provide functionality over encrypted data. Currently, only Microsoft provides such a database and therefore vendor lock-in is a risk. In the future, more databases might support TEE technology such that our design can be applied universally, but at the time of writing this is not the case.

5.2.2 Research limitations

This research includes an implementation and evaluation of the architecture proposed in chapter 3. Due to time constraints, the implementation deviates slightly from the described architecture. Techniques already implemented in the AE enclave could be re-used for the re-encryption enclave, but the source code of MsSQL and the corresponding enclave is not available. While the prototype gives a first feasibility prove, a real-world implementation is left to future work.

First of all, our architecture should come on top of traditional authorization and authentication (see **NR2**). For simplicity, the prototype does not contain any authentication and authorization. The prototype allows anyone to make API requests, while in a real-world implementation the application server should authenticate and authorize all requests. As in our prototype the requests give a public key to which data is encrypted, any attacker can simply asks for a re-encryption towards that attacker, which makes the re-encryption a decryption oracle. In a real-world implementation, the public key towards which data is encrypted should be validated.

[1] https://learn.microsoft.com/en-us/sql/relational-databases/security/encryption/always-encrypted-enclaves?view=sql-server-ver16

The proposed architecture includes a secure key management system. This component is not implemented, thus the related functionality is not implemented or implemented differently. The first functionality is attestation of the re-encryption enclave at the application server, which is omitted in our implementation. A real-world system should always attest enclaves to meet the security guarantees. Implementing attestation is not a technical issue, as Always Encrypted already implements such attestation to the database server. Secondly, keys are provisioned to the database and re-encryption enclave from the application server. This renders all of our security statements useless, since the combination of keys and ciphertext on the application servers is no better than having plaintext on these servers (violating **SR1**). In a real-world scenario, keys should be provided from a secure key management system. The fact that Always Encrypted can securely provision keys remotely shows that such a remote key provisioning is possible. Since keys can be securely cached at the TEEs, the secure key management only has to provision on enclave creation and when keys are rotated. Therefore, the additional performance overhead is minimal compared to our prototype.

For simplicity the implemented prototype only uses one key to encrypt all sensitive columns. In practice, one might choose to encrypt each column under a different key. This requires more keys to be provisioned to and stored in the re-encryption enclave. Furthermore, the re-encryption enclave should be aware of which key to use for every re-encryption. Slight changes in the design of the PRE enclave are required to allow this extra metadata to be used.

As shown in section 3.2.3 type information on the encrypted values is lost in our prototype. Type information and other constraints on encrypted data can only be verified within the enclave, as this verification requires access to the plaintext values. Such type checking or constraint validation is not implemented in the prototype.

Finally, the implemented prototype uses the Microsoft Virtualisation Based Security TEE instead of Intel SGX at the database server. The tests have been performed with all components running locally to eliminate network delays. However, the local version of Microsoft SQL Server does not support SGX as enclave technology. Therefore, using VBS was the only option in our set-up. Hosted versions of MsSQL on Microsoft Azure do support SGX.

5.3 Future work

The previous section listed the limitations and shortcomings of the architecture and prototype. While some limitations are inherent to the used technology, for instance the Always Encrypted limitations, others can be solved to come to a more reliable prototype. Our implemented prototype can serve as a starting point and inspiration.

For instance, integration with a secure key management system would make the prototype actually reach the trust boundary aimed for. Once such improvements are in place, further practical analysis could take place, for instance by penetration testing the architecture, or running a forensic analysis on the servers to see what sensitive data is leaked to an attacker. Note that for the latter a commercial license for Intel SGX should be obtained, in order to enable the hardware security guarantees.

The proposed architecture has chosen specific technology for the TEE and the database server. Alternative architectures could use a different database, different cryptography, or a different TEE technology. Various combinations of technologies and implementations are possible, all with their own advantages and disadvantages. Advancements in these individual components would improve the entire architecture as well.

The performance of the PREnclave and the architecture as a whole could be improved in further iterations. As mentioned in section 4.2.2, a pool of enclaves could be managed by the application server. This allows the database driver to re-use existing enclaves, reducing the execution time. Furthermore, multi-threading support could be added for re-encryption as these calculations do not depend on each other. Multi-threading could also speed up queries that only use backward re-encryption, as the enclave could be initialized while the query is already send to the database server. When the data returns, the PREnclave is ready for backward re-encryption. For queries where the PREnclave is not needed, it should not be created. To reduce the overhead of key provisioning, the long-term keys could be cached and stored using the SGX sealing technology.

Our research focuses on the scenario of a service provider that allows third parties to store and retrieve data. While this is a common use-case, various other scenarios could be thought of. Designing and implementing a similar architecture for other scenarios could show feasibility for other use-cases. For example, computations on the encrypted data using homomorphic encryption or TEEs could be included. The results section of our research is very focused on the example scenario, while future work could evaluate the architecture more generally. For instance, an industry standard benchmark such as TPC-C [2] could be used to do a performance analysis that allows comparison with other architectures.

For any organization to implement the proposed architecture, the maintenance and implementation effort should be minimal. This can be achieved when existing database drivers implement the desired re-encryption technology, just like the Microsoft DotNet database driver implements the Always Encrypted functionality. Libraries to help implementing the technology, including the secure key management system, should be available and easy to implement. The architecture makes use of generic software components and therefore it does not make sense for each service provider to individually develop and maintain these components. Instead, software companies that provide databases with encryption technology or communities should centrally develop the technology.

Chapter 6

Conclusion

To summarize our contributions, this work has presented an architecture that combines various state-of-the-art components in order to mitigate leakage of sensitive data at a trusted service provider. With this proposed architecture no sensitive plaintext data is present at the service provider at any time, which makes sure any possible data breaches can only contain encrypted data. The architecture protects against various types of attacks, including internal attackers. This increases privacy of data owners and prevents any possible fines for service providers.

A list of requirements (**RQ1**) is compiled in section 1.2 and evaluated in the results chapter (4). The cryptographic tools available to meet these requirements are explored in chapter 2, with in particular Intel SGX (2.2.2), proxy re-encryption (2.3), and Always Encrypted (2.4.2). These components are combined into the proposed architecture (**RQ2**) in chapter 3. The proposed architecture builds on existing work for database encryption and extends the provided encryption using proxy re-encryption in a customized database driver. Both database encryption and proxy re-encryption make use of trusted execution environments.

We implemented a prototype of this architecture which shows feasibility in practice. The functionality (**RQ3.1**) is similar to an architecture without encryption, with minor limitations. Most of these limitations are inherited from Always Encrypted. In terms of performance (**RQ3.2**) there is some overhead compared to a plaintext architecture or an Always Encrypted architecture without re-encryption. However, this performance overhead is in the order of tens of milliseconds, which can be considered worth the gained security benefit depending on the scenario. Furthermore, this work presents suggestions that can greatly improve performance of this first attempt.

Our prototype has some limitations and is not ready to be used in the industry yet. However, this initial feasibility check makes TEE technology more attractive and can stimulate further research and development. The architecture and implementation can serve as a starting point for further research. Finally, our analysis gives insight in the trade-off between functionality, performance, and security of our proposed architecture, which can help business owners to better predict the potential of the presented technology.

To answer the main research question (**RQ0**), leakage of sensitive data can be mitigated at a trusted service provider by using database encryption such as Always Encrypted and extending the encryption using trusted execution environment based proxy re-encryption. This allows the service provider to only handle ciphertext data on its servers, eliminating the possibility of any plaintext sensitive data leakage.

Bibliography

[1] Passmark cpu benchmarks - amd vs intel market share. URL: https://www.cpubenchmark.net/market_share.html.

[2] Tpc-c homepage. URL: https://www.tpc.org/tpcc/.

[3] Panagiotis Antonopoulos, Arvind Arasu, Kunal D. Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, Ravi Ramamurthy, Jakub Szymaszek, Jeffrey Trimmer, Kapil Vaswani, Ramarathnam Venkatesan, and Mike Zwilling. Azure sql database always encrypted. page 1511–1525, 2020. doi:10.1145/3318464.3386141.

[4] ARM. Arm security technology building a secure system using trustzone ® technology, 2009. URL: https://www.arm.com/technologies/trustzone-for-cortex-a/tee-reference-documentation.

[5] Giuseppe Ateniese, Kevin Fu, Matthew Green, and Susan Hohenberger. Improved proxy re-encryption schemes with applications to secure distributed storage. *ACM Trans. Inf. Syst. Secur.*, 9(1):1–30, feb 2006. doi:10.1145/1127345.1127346.

[6] Sumeet Bajaj and Radu Sion. Trusteddb: A trusted hardware-based database with privacy and data confidentiality. *IEEE Transactions on Knowledge and Data Engineering*, 26:752–765, 3 2014. doi:10.1109/TKDE.2013.38.

[7] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. *Journal of Cryptology*, 33:1914–1983, 2020. doi:10.1007/s00145-020-09360-1.

[8] Victor Costan and Srinivas Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016. URL: https://eprint.iacr.org/2016/086.

[9] Anwar Pasha Deshmukh and Riyazuddin Qureshi. Transparent data encryption - solution for security of database contents, 2013. URL: http://arxiv.org/abs/1303.0418, arXiv:1303.0418.

[10] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. Hardidx: Practical and secure index with sgx. 3 2017. doi:https://doi.org/10.1007/978-3-319-61176-1_22.

[11] Benny Fuhry, H. A. Jayanth Jain, and Florian Kerschbaum. Encdbdb: Searchable encrypted, fast, compressed, in-memory database using enclaves. *Proceedings - 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021*, pages 438–450, 6 2021. doi:10.1109/DSN48987.2021.00054.

[12] William G Halfond, Jeremy Viegas, Alessandro Orso, et al. A classification of sql-injection attacks and countermeasures. 1:13–15, 2006. URL: https://www.cc.gatech.edu/fac/Alex.Orso/papers/halfond.viegas.orso.ISSSE06.pdf.

[13] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. 2013. doi:10.1145/2487726.2488370.

[14] Seny Kamara and Tarik Moataz. Sql on structurally-encrypted databases. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11272 LNCS:149–180, 2018. doi:10.1007/978-3-030-03326-2_6.

[15] Nadim Kobeissi. An analysis of the protonmail cryptographic architecture. *Cryptology ePrint Archive*, 2018. URL: https://eprint.iacr.org/2018/1121.

[16] Dongxi Liu and Shenlu Wang. Query encrypted databases practically. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, page 1049–1051, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2382196.2382321.

[17] Joshua Lund. Signal - blog - technology preview for secure value recovery, 12 2019. URL: https://signal.org/blog/secure-value-recovery/.

[18] Microsoft. Virtualization-based security (vbs) | microsoft learn. URL: https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs.

[19] Muhammad Naveed, Seny Kamara, and Charles V Wright. Inference attacks on property-preserving encrypted databases general terms. *Proceedings of the 2015 ACM Conference on Computer and Communications Security*, pages 644–655, 2015. doi:10.1145/2810103.2813651.

[20] Alexander Nilsson, Nikbakht Bideh, and Joakim Brorsson. A survey of published attacks on intel sgx, 2020. doi:https://doi.org/10.48550/arXiv.2006.13598.

[21] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: An encrypted database using semantically secure encryption. *Proc. VLDB Endow.*, 12(11):1664–1678, jul 2019. doi:10.14778/3342263.3342641.

[22] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. page 85–100, 2011. doi:10.1145/2043556.2043566.

[23] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using sgx. *Proceedings - IEEE Symposium on Security and Privacy*, 2018-May:264–278, 7 2018. doi:10.1109/SP.2018.00025.

[24] Proton. Set account recovery methods in case you forget your proton password | proton. URL: https://proton.me/support/set-account-recovery-methods.

[25] P Ruffio. Dark web price index 2022 - dark web prices of personal data, 9 2020. URL: https://www.privacyaffairs.com/dark-web-price-index-2022/.

[26] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: What it is, and what it is not. 1:57–64, 2015. doi:10.1109/Trustcom.2015.357.

[27] Murugiah Souppaya, Karen Scarfone, and Donna Dodson. Secure software development framework (ssdf) version 1.1: Recommendations for mitigating the risk of software vulnerabilities. 2 2022. URL: https://csrc.nist.gov/publications/detail/sp/800-218/final, doi:10.6028/NIST.SP.800-218.

[28] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. Stealthdb: a scalable encrypted database with full sql query support. *Proceedings on Privacy Enhancing Technologies*, 2019:370–388, 2019. doi:10.2478/popets-2019-0052.

[29] B Wolford. What are the gdpr fines? - gdpr.eu. URL: https://gdpr.eu/fines/.

[30] Fan Zhang, Ziyuan Liang, Cong Zuo, Jun Shao, Jianting Ning, Jun Sun, Joseph K. Liu, and Yibao Bao. Hpress: A hardware-enhanced proxy re-encryption scheme using secure enclave. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40:1144–1157, 6 2021. doi:10.1109/TCAD.2020.3022841.

Appendix A

Enclave EDL

```
enclave {
    trusted {
        public int e_PREForward(
            [in, size=data_len] unsigned char* data,
            size_t data_len,
            [out, size=result_len] unsigned char* result,
            size_t result_len
        );
        public int e_PREBackward(
            [in, size=data_len] unsigned char* data,
            size_t data_len,
            [out, size=result_len] unsigned char* result
            size_t result_len
        );
        public int e_session(
            [in, size=key_len] unsigned char* key_enc,
            size_t key_len,
            [in, size=iv_len] unsigned char* iv_enc,
            size_t iv_len,
            [in, size=pk_len] unsigned char* publickey_client,
            size_t pk_len
        );
        public int e_get_session_client(
            [out, size=key_enc_len] unsigned char* key_enc,
            size_t key_enc_len,
            [out, size=iv_enc_len] unsigned char* iv_enc,
            size_t iv_enc_len
        );
        public int e_set_key_db_insecure(
            [in, size=key_db_len] unsigned char* key_db,
            size_t key_db_len
        );
        public int e_set_private_key_proxy_insecure(
            [in, size=key_proxy_len] unsigned char* key_proxy,
            size_t key_proxy_len
            );
        public int e_get_public_key_proxy(
            [out, size=publickey_len] unsigned char* publickey,
            size_t publickey_len
        );
    };
};
```

Listing 2: The enclave EDL file, which specifies all function with their incoming and outgoing parameters.