TÉCNICO LISBOA

ATOCS - Automatic Configuration of Secure Databases

David Jorge Louro Ferreira

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. Miguel Ângelo Marques de Matos
Prof. João Tiago Medeiros Paulo

Examination Committee

Chairperson: Prof. José Luís Brinquete Borbinha
Supervisor: Prof. Miguel Ângelo Marques de Matos
Member of the Committee: Prof. António Manuel Ferreira Rito da Silva

September 2020

Acknowledgments

I would like to thank my girlfriend, my parents and my siblings for all the support they gave me not only during the development of this thesis but also during all my college years.

I would also like to thank my supervisors, Miguel Matos and João Paulo, for all the help and feedback they provided me. They were crucial for the development of this work.

This work was partially funded by Fundo Europeu de Desenvolvimento Regional (FEDER) through Programa Operacional Regional de Lisboa, and Fundos Nacionais through FCT - Fundação para a Ciência e Tecnologia, under project Lisboa-01-0145-FEDER-031456 (Angainor).

Resumo

A computação em nuvem oferece serviços para empresas e indivíduos que necessitam de guardar e processar grandes quantidades de informação de forma escalável e com elevada disponibilidade. Porém, alguns estudos revelam que estes serviços podem ser atacados e revelar informações sensíveis dos seus utilizadores. Isto motivou o trabalho recente em bases de dados seguras, que combinam diferentes esquemas criptográficos para garantir que a informação está cifrada enquanto é processada na nuvem.

No entanto, para que estas bases de dados sejam seguras e eficientes, é necessário que sejam devidamente configuradas pelo administrador da base de dados, algo que requer um profundo conhecimento sobre a aplicação, o esquema da base de dados e sobre as varias propriedades dos esquemas criptográficos suportados pela base de dados segura. Este processo manual não é ideal visto que consome bastante tempo e que precisa de ser feito por um especialista tanto em bases de dados como em segurança, o que pode levar a erros humanos tendo em conta a complexidade das aplicações modernas.

Nesta dissertação propomos o ATOCS, uma ferramenta que automatiza a análise do código de uma aplicação e que é capaz de configurar e otimizar automaticamente a base de dados segura utilizada pela aplicação em causa de forma a atingir a funcionalidade, o desempenho e as garantias de segurança desejadas. A solução proposta é modular e extensível e por isso é simples suportar diferentes aplicações e bases de dados. O ATOCS foi avaliado com três aplicações reais. Os resultados mostram que a solução proposta é rápida, precisa e simplifica a configuração e otimização de bases de dados seguras.

Palavras-chave: segurança e privacidade, bases de dados, análise de código, automatização

Abstract

Cloud providers offer compelling services for individuals and companies that want to store and process large-quantities of data in a highly-available and scalable fashion. However, recent news have shown that these services can be breached and leak sensitive information from users. This has motivated the recent work on privacy-preserving databases that combine different security primitives to ensure that data is encrypted while being stored and processed at the cloud.

However, in order to be secure and efficient, these database systems must be properly configured by a database administrator, which requires deep knowledge about the applications, the database schema and the properties of the different security primitives supported by the database engine. This manual configuration is not ideal since it is a time consuming process that must be done by an expert on both database and security topics and can lead to human errors due to the complexity of modern applications and database engines.

This thesis presents ATOCS, a framework that fully automates the analysis of applications' code and that, from the inferred functional and non-functional requirements, is able to automatically configure and optimise the underlying secure database systems in order to achieve the desired functionality, performance and security guarantees. The proposed design is modular and extensible thus easing the support of different applications and database systems. ATOCS is evaluated with three real applications. Our results show that the proposed solution is fast, accurate and simplifies the configuration and optimisation of secure databases.

Keywords: security and privacy, databases, code analysis, automation

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The amount of information that companies and organisations have to store and process has been increasing exponentially in the past few years [1]. To accommodate this growth, Database Management Systems (DBMS) are commonly used to efficiently store, manage and query large amounts of data. The DBMS is able to perform operations on the data it stores such as ordering or arithmetic operations, which makes data access efficient and versatile. This way, it is possible to retrieve only the desired information from a database, when issuing a request to the DBMS.

Since some of the information stored in a database can be sensitive, data confidentiality is a prime concern for every company and individual. There are many aspects that can be affected by the lack of security mechanisms. Custumer satisfaction can degrade if the costumer's information gets exposed publicly. Companies that invest in Research and Development cannot have their research achievements compromised by a security breach, since that leads other companies to take advantage of their work without the same investment. Nowadays there are regulations that obligate companies to protect their users' data such as the General Data Protection Regulation (GDPR) [1] that companies operating in the European Union have to comply with.

Orthogonal to these security and privacy aspects, there is a business benefit for companies to push their data storage and processing tasks into the cloud due to economies of scale and the improved availability without the need to make up-front investments in private infrastructures [2]. This is very enticing as it enables companies to invest in key business areas rather than investing in infrastructures to store and process their data. Nevertheless, moving the data to a cloud is in direct confrontation with the security, privacy and legal compliance aspects discussed above, as the cloud provider has full control over the infrastructure and data. This conundrum led to the

[1]General Data Protection Regulation, European Commission, https://gdpr-info.eu

[2]Gartner Says the Future of the Database Market Is the Cloud, Gartner, https://www.gartner.com/en/newsroom/press-releases/2019-07-01-gartner-says-the-future-of-the-database-market-is-the

development of secure database systems such as CryptDB [2] and SafeNoSQL [3].

Secure database systems allow an application to manipulate and query data residing in an untrusted location (such as a cloud provider) while offering security and privacy guarantees over the data and queries. The data is kept encrypted in the untrusted location and the queries are converted into secure ones (by encrypting any values used in them). This is achieved by leveraging a combination of encryption schemes that support operations over encrypted data such as equality, ordering or arithmetic operations. Naturally, the right encryption scheme to use depends on the application's data and operations. This decision must be made when the database schema is being designed.

In fact, if the encryption schemes are not chosen correctly, the secure system will not provide the desired functionality as the database will not be able to perform the operations correctly (*e.g.*, requiring a sum operation with an encryption scheme that does not support algebraic operations). Moreover, even if the encryption schemes provide the required functionality, they might still not be the most adequate in terms of performance or in terms of security as they may leak sensitive information.

With the increasing complexity of modern applications, not only due to the application logic itself, but also due to security, privacy and legal compliance requirements, it becomes increasingly difficult - and hence error prone - to select the right set of encryption schemes. As a matter of fact, to make an informed decision the application developer or database administrator needs to be knowledgeable not only of the application and database requirements but also of the security, functionality and performance trade-offs of advanced encryption schemes. We argue that this crucial decision, with far reaching implications in the security and performance of the system, must not be done manually by the application developer, but should rather be fully automated.

In this thesis, we propose ATOCS, an extensible framework that given the application's code is able to automatically derive a set of encryption schemes that best fulfil the developer's security and performance goals. This is achieved by relying on code analysis tools to automatically infer the types of operations performed over data (such as equality or arithmetic operations) and then choosing an encryption scheme that supports those operations. When several possibilities are available, ATOCS proposes the one with the best security or performance trade-offs according to the developer's goals. By fully automating this process, ATOCS removes human error and leads to more secure and performant database applications.

ATOCS can be used not only when designing a new secure application, but it can also analyse an existing application using a standard database system while providing the configuration required for a future integration with a secure database.

The conducted experiments, with three real applications and a secure database system, show that ATOCS can analyse different applications while automatically providing a tailored secure schema according to the performance, security and functionality requirements of each application. Moreover, the results show that this analysis can be conducted efficiently, in less than 45 seconds for applications with thousands of lines of source code, and can even be used to optimise performance and resource usage configurations of secure databases.

Briefly, this thesis makes the following contributions:

- ATOCS, a framework that automatically analyses the application code and determines the most appropriate encryption schemes and database optimisations;
- a detailed evaluation with three real-world applications that showcases ATOCS's applicability to a wide range of scenarios;
- an open-source implementation of ATOCS.

The remainder of this document is organised as follows. In Chapter 2, we present an overview of property-preserving encryption schemes, the state of the art of secure databases and code analysis tools. This is followed by Chapter 3 which describes the architecture and implementation of ATOCS. Chapter 4 presents the experimental evaluation and Chapter 5 concludes the document and discusses future work.

Chapter 2

Background and Related Work

In this chapter, we present the background and related work. In Section 2.1 we discuss some relevant state of the art in secure database systems. Section 2.2 describes the state of the art of code analysis tools.

2.1 Secure Databases

There has been an increasing amount of research work that aims at providing secure data storage and processing for databases deployed at untrusted cloud services. To safely secure data on the cloud, the data has to be encrypted using an encryption scheme. Each scheme has a cipher and decipher mode, both of which are presented in Figure 2.1.

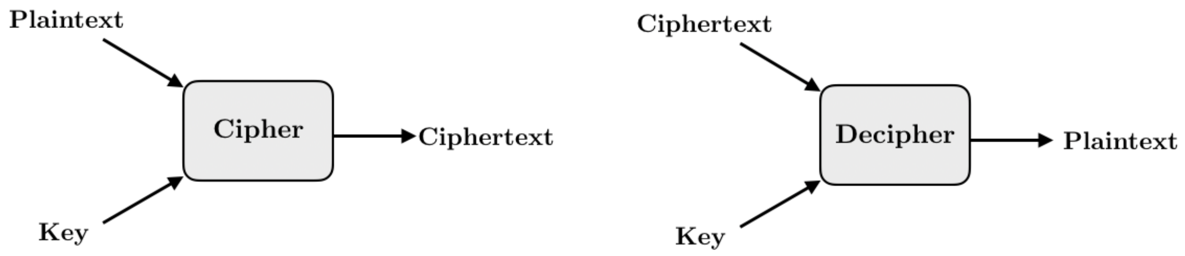


Figure 2.1: Encryption modes. On the left we have the cipher mode that receive the original message as plaintext and a key. It then generates a ciphertext that corresponds to the plaintext received. On the right there is the decipher mode which is the opposite. It turns the ciphertext into the plaintext when provided with the same key that was used to cipher the original plaintext.

The simplest approach is to encrypt all the information, with a strong probabilistic encryption scheme (*e.g.*, probabilistic AES), before sending it to the cloud database server [4]. However, this does not allow executing any kind of processing (equality, order, or arithmetic operations over the ciphertext) at the database server. To perform such operations, data needs to be transferred to a trusted environment, for example the client premises, decrypted and then processed. Naturally,

Table 2.1: Operations supported by different encryption schemes. The encryptions that the table compares are: Plaintext or no encryption (PLT), Standard Encryption (STD), Deterministic Encryption (DET), Order-preserving Encryption (OPE), Format-preserving Encryption (FPE), Partial-Homomorphic Encryption (HOM) and Searchable Encryption (SE).

	Operations				
Encryption Scheme	***Equality***	***Order***	***Format***	***Search***	***Algebraic***
PLT	✓	✓	✓	✓	✓
STD					
DET	✓				
OPE	✓	✓			
FPE	✓		✓		
HOM					✓
SE	✓			✓	

such approach has very poor performance which tends only to degrade as the amount of data grows.

To overcome these shortcomings, secure database systems resort to encryption schemes that preserve different kinds of properties about the original plaintext in the corresponding ciphertext. These schemes allow certain operations (*i.e.*, equality, order, arithmetic) to be performed directly across ciphertexts stored at the database server, thus avoiding the need to perform additional computation at the client premises.

Table 2.1 details the encryption schemes that are most commonly used by existing secure database systems.

Plaintext (PLT) represents data that is not encrypted and thus supports any type of operation.

The Standard Encryption (STD) uses a probabilistic cipher such as AES [4] in CBC mode [5]. This scheme has the best security guarantees, however, it does not support any operations over the ciphertexts.

Deterministic Encryption (DET) uses a deterministic cipher like AES in ECB mode [5] and ensures that if two plaintexts are equal, then their respective ciphertexts are also equal. This allows equality operations to be performed.

Additionally, Order-preserving Encryption (OPE) [6] improves the functionality of DET by also guaranteeing that if plaintext P1 is greater than plaintext P2, the corresponding ciphertext C1 is greater than the ciphertext C2. As a result, not only are equality operations possible, but it is also possible to order ciphertexts and hence get the minimum or the maximum of a set.

Format-preserving Encryption (FPE) [7] ensures that the ciphertexts maintain the size and type of the respective plaintext. Besides this, some implementations also allow the same functionality as the Deterministic Encryption.

Homomorphic Encryption (HOM) [8] is used to provide algebraic operations between ciphertexts. This can be divided into two categories: Fully-Homomorphic Encryption and Partial-Homomorphic Encryption. The first is able to provide multiple algebraic operations with the caveat of having a slow cipher performance which makes it unusable in real-world scenarios. The Partial-Homomorphic Encryption improves the performance of the Fully-Homomorphic Encryption by limiting the operations available, making it a better fit for most applications. In the rest of the document when using HOM, we are referring to the Partial-Homomorphic Encryption.

Searchable Encryption (SE) [9] is used for word search in documents using equality checks. Some implementations also allow regex operations [10].

Note that some of these schemes leak the properties preserved in the ciphertext to an attacker, thus providing weaker security guarantees. For example, if equality is preserved by using a deterministic encryption scheme, an attacker can infer if two ciphertexts correspond to the same plaintext or not. Further details about the guarantees of each scheme can be consulted in previous work [2, 3].

These encryption schemes are combined and used by secure database systems to provide data confidentiality and privacy in untrusted cloud deployments. Most of the current solutions consider an honest-but-curious untrusted environment that is able to inspect the information stored and being processed at the server. As for the database client, it is assumed to be deployed on a trusted environment so that the encryption and decryption of data can be performed there.

In the rest of this section we detail some the state of the art secure database systems. We will also present a discussion and comparison of these systems in the last section.

2.1.1 CryptDB

CryptDB [2] is a system that addresses the challenges previously presented with four main ideas:

- Encrypt the data before sending it to the cloud provider.
- Execute SQL queries over encrypted data.
- Adjust the encryption schemes in run-time.
- Provide a key management system.

As it can be seen in Figure 2.2, the CryptDB system is divided into an Application server, the CryptDB proxy server and the unmodified DBMS server. The queries are issued by the Application and pass through the proxy before being sent to the DBMS. The Application server

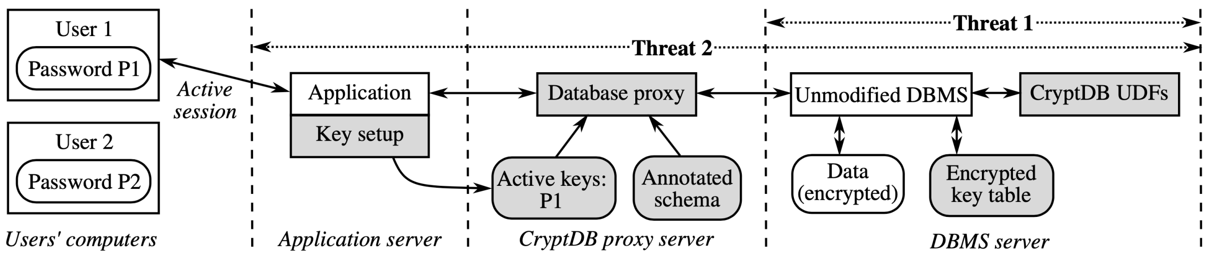


Figure 2.2: CryptDB architecture. The vertical dotted lines separate the different components of the system which run in different machines (although the Application and the proxy can run on the same server). Rectangular boxes are processes while rounded boxes represent data. The grey boxes represent the CryptDB components while the white ones are the standard and unmodified application components. The two threats that CryptDB addresses are represented on the top. Threat 1 is a curious cloud provider that observes the data stored in its server. Threat 2 is a compromise of not only the DBMS server but also the Application server. This figure was obtained from the CryptDB paper [2].

and the proxy are considered to be trusted whilst the DBMS server is considered untrusted so it only sees an anonymised version of the schema, encrypted data, some auxiliary tables from CryptDB and the user-defined functions (UDFs).

All the hard work of encrypting and decrypting queries is done by the CryptDB proxy. The main idea with this is to leverage the fact that SQL queries have a set of well defined primitives that can be rewritten (by changing some query operator while preserving the query semantics) in order to be performed on encrypted data and hence provide confidentiality when stored in the DBMS. The data is encrypted before being sent to the DBMS. CryptDB enables the database to perform its standard operations as if the data was decrypted. However, due to the data encryption, some of the data's properties are lost and for that reason more complex queries (for example search queries with regular expressions or summation queries) cannot be performed over ciphertexts. For this reason, it is possible to use user-defined functions that extend the functionality of the database while still preserving the confidentiality of the information stored. These user-defined functions are stored alongside the DBMS. In order to execute these functions the proxy replaces the queries needed with calls to the UDFs.

The encryption schemes supported by CryptDB are Standard, Deterministic, Order-preserving, Homomorphic and Searchable. Since some of the encryption schemes reveal more information to the DBMS server than others, CryptDB developed a solution to avoid revealing all the possible encryptions *a priori*. To do so, it uses *onions of encryption*, where a column can be encrypted with multiple ciphers at a time by encrypting the data in layers. The outer layer is the one with the most secure encryption and the inner layer with the most functionality. The system starts with the most secure encryption for each column. As queries arrive, some layers of encryption may need to be pealed off to allow for the operations to be performed. The decision of which

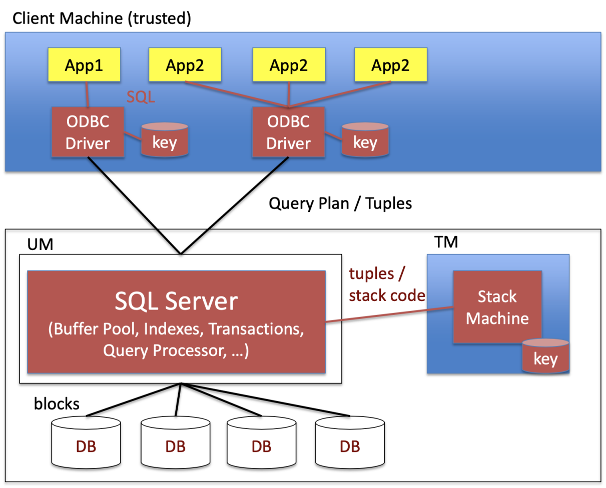


Figure 2.3: Cipherbase architecture. The Client Machine and the Trusted Machine (TM) represent the trusted components while the Untrusted Machine (UM) contains the untrusted ones. Cipherbase maintains the same components as traditional databases despite still adding new components like the TM in the untrusted server and extending the functionality of the Open Database Connectivity (ODBC) drivers. This figure was obtained from the Cipherbase paper [11].

encryption schemes to use for each column must be previously specified manually in the database schema. The proxy stores the database schema as well as the ciphers currently used by each column. In order to remove layers of encryption CryptDB sends the required keys from the trusted site (proxy) to the untrusted site (DBMS server) for the decryption to be performed.

CryptDB minimises the threat of data leakage to the cloud provider where the DBMS resides since all the information stored on the cloud is encrypted and is never decrypted to plaintext. Furthermore, CryptDB also tries to address another threat which is when not only the DBMS server is compromised but also the Application Server. Here CryptDB contributes with a key management system for its users. This system allows the stored data to only be accessed by the users that own the data, which can be a single or multiple users. The data encryption and decryption using the user's key is done in the proxy. However, the proxy only has the keys of the active users which ensures that the data of the logged out users is not leaked if the Application server is compromised. As for the logged users, this assumption cannot be guaranteed.

2.1.2 Cipherbase

Many approaches on secure databases have a tradeoff between security and functionality. As a result, very sensitive information might not be encrypted with a strong encryption because of the operations that are necessary to be performed over it. Cipherbase [11] introduces a different view to solve this issue with a full-fledged SQL database system that is capable of granting high confidentiality and processing over strongly encrypted data.

This system is divided into a Client machine (trusted machine) and a Server (untrusted machine) as depicted in Figure 2.3. The client has an ODBC driver which receives the client's requests before sending them to the database. Then the database processes the requests and responds back to the client, just like a standard database system. Cipherbase differs from normal databases by extending the functionality of the Open Database Connectivity (ODBC) driver. This driver is responsible for encrypting the client's requests before sending them to the database which is assumed to be in an untrusted environment. The decision of which encryption scheme to use depends on the sensitivity of the information in question. The system not only supports plaintext processing but also Standard encryption, Deterministic encryption and Order-preserving encryption. The ODBC driver stores a key for each application in order to encrypt the requests sent and decrypt the responses received. These operations are transparent to the application.

The other addition to a standard database that comes with this system is a secure co-processor, called a Trusted Machine (TM), that resides in the untrusted database server. This is a hardware module that needs to be installed alongside the database (referred to as an Untrusted Machine, UM). The query processing over strongly encrypted data is performed in the TM. The decision of whether to use the TM or the standard database depends on the operation and the encryption scheme of the data. For instance, if the request requires an equality check and the data is encrypted with Deterministic encryption there is no need to use the TM, since this encryption scheme allows equality operations between ciphertexts. However, if the data was stored with Standard encryption, the TM would be required. In the latter, the query is sent to the TM, which decrypts the data in order to perform the desired operation. The confidentiality is guaranteed since the TM has security mechanisms to provide the necessary isolation from the untrusted server so that no information is leaked.

The decision of whether to use the UM or the TM for query processing in done by the ODBC driver. Instead of sending an encrypted query to the database, the ODBC driver sends a query plan specifying the requirements to perform the request's operation. When queries need to be performed in the TM, the UM has to send not only the data but also the required encrypted keys (which are stored encrypted in the UM). This imposes a performance penalty and therefore Cipherbase tries to minimise the interactions between these two modules. This can be achieved by encrypting the data with a weaker encryption scheme when possible.

The main advantage of using a Trusted Module is the ability to perform all the standard database operations even on strongly encrypted data in an untrusted environment. This grants orthogonal security to the system, which means that it provides full functionality regardless of the chosen security policy. Nevertheless, TM has a negative impact on system performance since

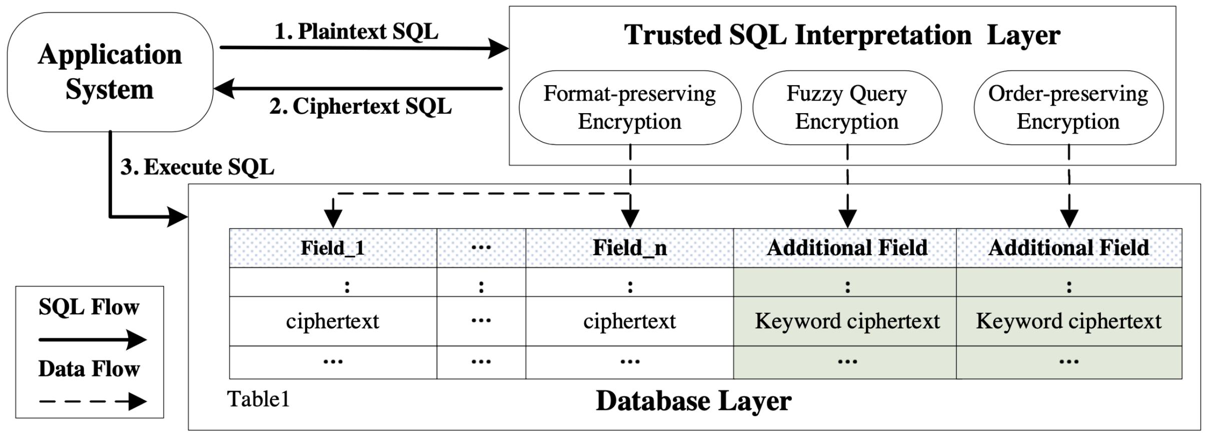


Figure 2.4: L-encDB architecture. The straight arrows denote the SQL query flow. The application layer has a Trusted SQL Interpretation Layer responsible for the requests interpretation and transformation. The requests first pass the Interpretation Layer before being sent to the database layer. The dotted arrows represent the data flow for each encryption type. This figure was obtained from the L-encDB paper [12].

data has to be delivered to it and decrypted before the processing is able to be performed.

2.1.3 L-encDB

Traditional database encryption has the caveat of changing the database structure since the encrypted data does not preserve the format of the original plaintext, so the database has to be adapted to this new format. This is the challenge that L-encDB [12] aims to solve. It is a lightweight framework for privacy-preserving data queries in cloud computing that does not change the database structure, enables operations on all kinds of databases (including text based) and supports SQL operations like advanced fuzzy queries (finding a certain character in a document) and range queries. The architecture is presented in Figure 2.4.

The system has a Trusted SQL Interpretation Interface, that is responsible for interpreting the queries, encrypting constants and creating queries with ciphertexts. This interface is viewed as an API for the application system and can be deployed either at the client side or the application service layer. The application is still the one responsible for sending the requests to the database, however it needs to send them to the Interpretation Interface beforehand for the query conversion to be performed. Regarding the database layer, the developers are not allowed any operation beyond SQL-based functions. This implies that they are not allowed to extend the functionality of the database, for example with stored procedures.

In order to preserve the database structure, L-encDB uses a Format Preserving Encryption (FPE) for every column since this encryption preserves the type and size of the original data in the ciphertexts. This is an important feature that makes the integration of L-encDB much easier in already developed systems as there are no adjustments that have to be made to the structure of the original database, we just need to integrate the process of encrypting/decrypting data with the Interpretation Interface. In addition, a custom FPE was developed, which can preserve both the length and storage size of character strings. This cannot be efficiently achieved with normal FPE schemes. This is the main differentiating factor of this system. The normal fields of each table contain the original data encrypted with FPE although there can be additional fields used for the range and fuzzy queries ciphertexts.

For every request, the Interpretation Interface rewrites it in order to encrypt its constants with FPE. This is the basic procedure for most requests. However, this can differ when dealing with columns that support either range queries or fuzzy queries. In these cases, if the query is an insert or update, there is the need to add an extra field to the query where not only is the original field inserted/updated with the data encrypted with FPE but also an extra field destined to the fuzzy or range query encrypted with FQE (Fuzzy Query Encryption) or OPE, respectively. Therefore, when a fuzzy/range query request is received at the Interpretation Interface, it changes the field on which the operation will be performed to the respective extra field.

The main takeaway from this system is the fact that it is easy to deploy, since it does not require any changes to the database layer. This is the consequence of using a FPE for every data item in the database, which means its structure remains intact.

2.1.4 SafeNoSQL

The SafeNoSQL [3] system delivers a modular and extensible architecture that enables query processing over encrypted data for NoSQL databases. Its main contribution is to provide a system that guarantees the confidentiality of the data stored in the databases with a minimum impact on the application's performance.

The architecture, presented in Figure 2.5, is divided into a trusted and untrusted site. The application, NoSQL client and a CryptoWorker reside in the trusted site. This CryptoWorker is the only point of contact from the trusted site to the untrusted. As for the untrusted site, it is composed of the NoSQL database and another CryptoWorker.

CryptoWorkers are a crucial part of this system. They provide a privacy-aware NoSQL API that enables a simple integration in NoSQL systems. All the requests sent by the client are intercepted by the CryptoWorker (number 2 in Figure 2.5) in the trusted site. Then, the query is converted into one with the same semantics but with greater security guarantees by encrypting the information. This conversion can differ depending not only on the query itself but also the cryptographic needs of the data being sent, i.e. a column encryption is chosen based

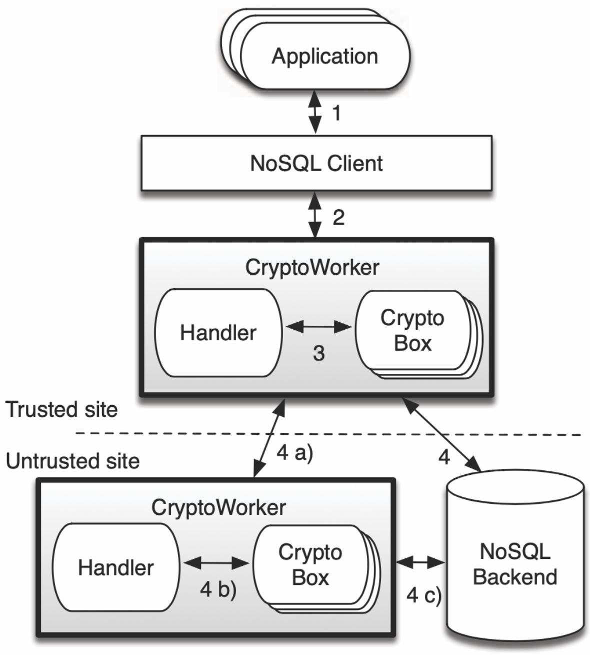


Figure 2.5: SafeNoSQL architecture. The horizontal dotted lines separate the trusted and untrusted sites. On the top part resides the Application, the NoSQL Client and a CryptoWorker to process the requests. On the bottom there in the NoSQL Backend and another CryptoWorker. This figure was obtained from the SafeNoSQL paper [3].

on the operations that are going to be performed over it. For that reason, the CryptoWorker has various CryptoBoxes, one for each encryption scheme supported. SafeNoSQL can easily be extended by adding new CryptoBoxes with new encryption schemes. The CryptoWorker has access to the database schema where it retrieves the information about the encryption schemes required.

After the query is processed in the CryptoWorker, the request is sent to the NoSQL database (number 4 in Figure 2.5) where the query is performed as if it were in plaintext (provided that the required cipher was used). Each database column can have a different type of encryption depending on the functionality required for said column. The cipher used has to be chosen accordingly and be specified in the database schema. The encryption schemes supported by SafeNoSQL are Standard encryption, Deterministic encryption and Order-preserving encryption. However, as stated above, this system is modular and can be extended with other encryption schemes by developing a new CryptoBox that support them.

The majority of the requests sent to a NoSQL database are supported just by using the CryptoWorker in the trusted site. Nevertheless, there are some operations where some additional calculations in the untrusted site are required, for example if we want to perform search queries. For that reason, SafeNoSQL also provides a CryptoWorker in the untrusted site that can be extended with other encryption schemes. When requests of this type arrive, they are sent to the

CryptoWorker in the untrusted site instead of directly to the database (number 4a in Figure 2.5).

With this modular approach, SafeNoSQL ensures that its system can easily be upgraded as the developer sees fit.

2.1.5 Discussion

One of the main contributions of the four systems presented above is to provide confidentiality of the data stored in cloud databases. This is accomplished by encrypting the data before sending it to the cloud provider.

All these systems allow the database to operate on ciphertexts as if they were plaintext. Nevertheless, there are some different approaches to accomplish this. CryptDB and SafeNoSQL are similar in this matter, where both encrypt a column with the encryption required to perform the operations over said column. In addition to this, these two systems allow the developer to extend the functionality of the database by creating new functions that run on the server side. These functions can be used to perform operations over encrypted data that are not possible without additional server side processing (such as search queries or algebraic operations).

Cipherbase leverages the Trusted Module to perform all the operations that the database is not able to provide (due to the fact that the data is encrypted). Since the TM is a secure environment, it can decrypt the data and perform the operations on plaintext. However, this has an impact on performance. Every time a query needs to be executed in the TM, the query data has to be sent from the UM to the TM as well as the keys to decrypt the data (that are stored encrypted in the UM). Cipherbase tries to mitigate this issue by encrypting the data with the encryption that matches the functionality needed for that data so that the UM can process the query without interacting with the TM. Nevertheless, by doing this and avoiding the contact with the TM, Cipherbase will start to behave just like CryptDB and SafeNoSQL. Besides the performance issues, the use of a TM presents an additional component/manufacture that has to be trusted.

As for the L-encDB system, it encrypts every field with FPE which is deterministic and preserves the original data's size and type in the ciphertexts, so it can perform queries over encrypted data that requires equality checks. Alongside the FPE ciphertexts, L-encDB also stores two extra fields of the same data encrypted with OPE and FQE (for the columns that require range and fuzzy queries). This makes L-encDB the easiest to be adapted to an already developed application, since the database fields will preserve their type and size. The caveat of this system is the fact that we cannot add other encryption schemes, which makes operations, such as algebraic ones, unfeasible.

CryptDB differentiates itself by using onions of encryption where multiple encryptions can be used for the same data. With this method the weaker encryptions are not revealed to the cloud provider a priori. However, as the system ages and some encryption layers are peeled off, CryptDB will tend to reveal the same information as the other systems. Besides this, to peel off the encryption layers, the trusted site has to send the required keys to the untrusted site. This is not only insecure but also has an impact on the system's performance. A possible attacker that has access to the cloud provider can keep these keys for future use.

As for SafeNoSQL, it was developed with a modular design in mind regarding the addition of new encryption schemes. In order to add a new encryption scheme the developer only has to create a new CryptoBox that implements it.

However, all these systems can suffer from Inference Attacks [13] where the attacker gathers public information and combines it with the activity monitored in the cloud to infer the information stored in the databases. Although we acknowledge this concern, this will not be addressed in this thesis. Alongside this, there can also be a functionality concern. For applications that require multiple clients, where each client cannot share its information with others, we need a key management system. Although both CryptDB and Cipherbase address this issue, SafeNoSQL and L-encDB do not. We consider this an orthogonal concern to our work which focuses on properly configuring these databases and not improving these systems.

A common requirement of these four systems is the fact that the schema has to specify which encryption schemes to use in each database field. This decision has to be done correctly so that the application yields the expected results. However, none of these systems provide an automated way to accomplish this. A manual configuration is not ideal since it is prone to errors which will affect the behaviour of the application. Aside from that, it is also difficult to reach an optimal configuration taking into account the complexity of modern applications.

2.2 Code Analysis Tools

Code analysis tools inspect an application's code and can be used for various purposes. Such analysis can be used to detect bugs, inspect the quality of the code, verify if the code matches a certain programming style and understand the functionality of the applications.

Although these tools can be very versatile, in the scope of our work we are only concerned with some specific parts of an application, mainly its interactions with the database. Any other information that these tools can provide such as code correction, complexity and bug reports can be discarded and are not considered to be an advantage. Ideally, the tool would be able to only focus its analysis on the database interactions, in order to improve its performance. An interaction with a database is any request that is sent to it. This informs us about the operation that is being requested to the database. It is also important to determine the operands of these requests, which represent the specific database fields that are being accessed, so that we know the fields on which an operation is being performed. These are the details required to be able to determine the encryption and functionality needs for each database field, which can then be used to choose the appropriate encryption scheme. The literature review and discussion below focusses only on these aspects.

Code analysis can either be dynamic or static. Dynamic analysis requires the program to run against previously defined test cases. Then the output generated by each test is analysed. Due to this, the obtained results are guaranteed to be sound, i.e. there are no false positives which ensure that the provided inputs generated the obtained outputs. However, this type of analysis is not complete since it relies on the coverage of the tests available and hence it might not detect every possible outcome. Real-world applications are complex and defining every possible input is in most cases unfeasible. For this reason, dynamic analysis tools can miss important behaviours generated by the inputs that were not accounted for.

In static analysis the program does not need to be executed. Only the code itself is analysed. The fact that it does not rely on the inputs provided by the programmer means that it assumes every possible scenario. As a result, it traverses every path available instead of only following the path of a certain input. This is what makes it account for possible scenarios that the dynamic approach misses and hence be considered complete [14].

Mainly due to the fact that dynamic analysis cannot explore every path of an application, we consider static analysis as the only option to deliver the desired results for this thesis.

The state of the art of code analysis tools is quite vast [15] so we will focus our study on some representative tools.

2.2.1 PMD

PMD [1] is an open-source tool that provides static analysis on Java source code based on rules. There are a set of default rules whose main goal is to spot common developer mistakes in Java programs, such as empty try-catch blocks, variables that are never used and objects that are unnecessary. Besides these default rules, PMD can also be extended with custom rules. A rule is used to analyse part or all of the code and decide whether it violates a certain principle or not.

In order to perform an analysis, PMD first receives the byte-code files to analyse and sends them to a Java code parser which outputs an Abstract Syntax Tree (AST). This tree represents

[1]PMD Source Code Analyzer, PMD, https://pmd.github.io/pmd-6.14.0/index.html

the structure of the byte-code provided. If the rules enabled require it, PMD can also generate control flow graphs and data flow nodes based on the AST. After gathering this information, the rules are applied by traversing the AST and accessing the required data structures previously generated. When a rule is considered to be violated it is reported. In the end, the output will display all the violations found in the inspected code.

2.2.2 Soot

Soot [16] [2] is a framework that enables developers to implement static analysis for Java programs. Soot can receive as input either Java byte-code or Java source code and generally outputs Java byte-code. After receiving the input byte-code, the framework transforms it into an intermediate representation (Jimple [16] is the default one). It does this to simplify the analysis of Java programs since byte-code can be difficult to analyse as the implicit stack can mask the control flow of the program. The Jimple intermediate representation (IR) is a mix of Java byte-code and source code. It stores the stack data in named local variables which makes the flow of data more obvious. For that reason, the analysis are done over the IR.

There are two major types of analysis that can be developed with Soot which are Intra-procedural analysis and Inter-procedural analysis.

Intra-procedural analysis is related to the individual methods of a Java program. Each analysis is independent and executed for a single method. Soot can create Control-Flow Graphs (CFG) for each method required. CFGs are directed graphs where each node represents a Soot Unit (which can be viewed as a Java statement) that points to the Units that follows it in the execution flow. With this information Soot allows the developer to create a Flow Analysis that feeds from the CFG and follows every execution path in it. We can use this to follow the execution flow of certain variables and determine their possible values.

In the Inter-procedural analysis, the whole program is taken into account. Contrary to the previously described analysis, where we have an execution of the same analysis for each method, here the analysis happens only once. Soot is able to create Call Graphs where for each method call it is able to determine all the possible class targets. Not only that but it is also possible to determine all callers of a certain method. For example, in our study, it is possible to determine the application methods that have any interaction with the database by knowing the database API.

Soot supports different forms of outputs. The most common is Java byte-code which can be instrumented by the analysis performed and be used as the new application byte-code. However,

[2]Soot - A Java optimization framework, Sable Research Group, https://github.com/soot-oss/soot

it can also output error messages, HTML files with the analysis results and class files with annotations from the analysis.

The main objective of Soot is to provide the developer with as much information about the program code as it can to improve the precision of the analysis being performed.

2.2.3 Java Path Finder

When analysing a program, one way to discover if the program behaves as expected for any use case is to execute the program with a wide range of inputs and see if any of the inputs result in an undesirable outcome. However, in real world applications, the number of possible inputs is huge, which makes this approach unfeasible.

Java Path Finder (JPF) [3] is a tool that uses Symbolic Execution [14] to explore every possible execution path of a Java program. In Symbolic Execution, a variable can either be assigned to a value or a symbol, depending if the concrete value is known or not. In the normal execution of a program, since we know the concrete values of each input, only one flow path is explored. On the contrary, Symbolic Execution explores multiple paths for all possible values of each input. During this static analysis, the symbolic execution engine (which performs the actual analysis) maintains a map of the variables previously encountered assigned to symbolic expressions or values and a formula that describes the conditions from the branches already taken by the current execution. With this information, a model checker can be used to determine if a certain property has been violated by one of the explored paths and also if all the paths' formulas can be satisfied by assigning concrete values to the symbolic expressions. For example, we can use this to determine if a variable can have a certain value in any state of the program. Although we might not be able to determine the exact value, we can have a range of possible values in a given state.

As a result of exploring every flow path of a program, Symbolic Execution is considered to be sound (all possible unsafe inputs are guaranteed to be found) and complete (input values considered unsafe are actually unsafe, which corresponds to bugs in the program). Nevertheless, this approach may become unfeasible for real-world applications where the number of paths to explore is greater than the resources available to store and process those paths. Due to this caveat, many analysis based on Symbolic Execution trade soundness for performance. This can be achieved by either mixing Symbolic and Concrete Execution (where we consider concrete values as inputs and not symbols), exploring first the paths considered to be more relevant or resorting to a backwards approach to Symbolic Execution instead of forward [14].

[3] Java Path Finder, https://github.com/javapathfinder/jpf-core

2.2.4 Discussion

We have now presented different tools for code analysis. In all of them, the objective is to gather information about a program's behaviour. The information obtained with these analysis can be used to better understand the programs' functionality and requirements. As we have discussed above all the tools described perform static analysis, since we consider dynamic analysis to not be ideal for our goal.

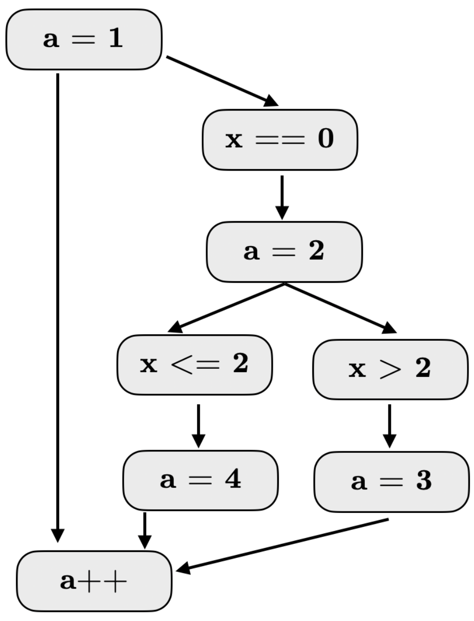
From the three tools presented we have two distinct techniques. PMD and Soot use Data Flow analysis and Java Path Finder uses Symbolic Execution.

With the Data Flow analysis we have a graph that connects all the possible paths between the nodes (which are the program instructions). The connections between nodes represent the possible execution flow of information. We can then use this graph to see how some values propagate through the nodes. However we are only inspecting the flow of data and not actually considering every possible value of every variable in each node and making our branch decision based on them. The associations are generated just by analysing what are the nodes that can be reached from a certain node, without considering the previously taken paths. To better understand this consider the code in Algorithm 1 and the respective Data Flow graph in Figure 2.6. The algorithm presents a method that creates a local variable *a* and updates it based on the value of the input parameter *x*. We are able to conclude, by reading the code, that the branch where *x>2* is never taken since the previous branch stated that *x==0*.

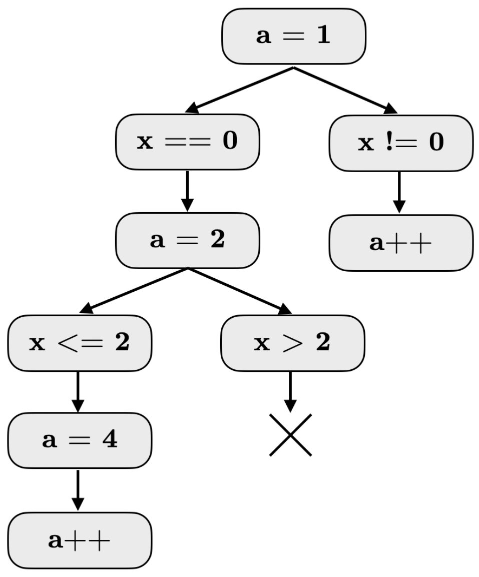
Algorithm 1: Function example.

```
 1 method f (int x)
 2     int a = 1;
 3     if x == 0 then
 4         a = 2;
 5         if x > 2 then
 6             a = 3;
 7         else
 8             a = 4;
 9         end
10     a++;
```

One disadvantage of Data Flow analysis is the fact that it considers every path between the nodes, even if they are not feasible. As a result, this approach can lead to some false positives. As for Symbolic Execution, it is able to distinguish between feasible and non feasible paths and only transverses the first, which makes it more precise. This can also be observed in Figure 2.6. If the first if statement (Listing 1, line 3) is true then we know that variable x is zero. For this reason, the second if statement (Listing 1, line 5) has to be false. The Symbolic Execution is



(a) Data Flow graph.



(b) Symbolic Execution tree.

Figure 2.6: These are two different representations of the analysis of the Java code presented in Algorithm 1. On the left we have the Symbolic Execution tree and on the right the Data Flow graph.

able to notice this and only traverses the path where the second if is false. As for the Data Flow graph it represents every path without taking any variable values into account.

Symbolic Execution assigns variables to symbols instead of concrete values so that it considers every possible path to be explored. When a branch occurs, the analysis splits to inspect each branch independently and at the end of each branch the results are not merged but rather continue as two different execution paths. In each step taken, the symbols are updated with the result of the instruction performed or branch taken. By doing so, this type of analysis is prone to path explosion. This happens when there are many consecutive branches or loops, making the analysis split every time and greatly increasing the number of paths it has to explore. In the Data Flow analysis this is not a problem since when a loop occurs we are not actually simulating its execution, we just view it as a possible flow of data.

Symbolic Execution is better than Data Flow analysis if precision is a key factor. However, it can also be very complex and its performance can easily degrade with a complex application. Besides that, database applications tend to be very complex which makes Symbolic Execution unable to finish its analysis due to the amount of paths it has to consider. Therefore, this thesis relies on the Data Flow analysis technique.

Due to the complexity of database applications, it is difficult to configure them the correct way. However, the tools just discussed above are able to infer the execution paths of applications which can be leveraged to determine the database operations performed. With this in mind, we make the process of configuring a database much easier. In the following section we discuss our approach to accomplish this.

Chapter 3

ATOCS

This chapter discusses the design and implementation of ATOCS, whose main goal is to automate the configuration of secure databases through the analysis of application's code.

Given an arbitrary application, ATOCS needs to address the following challenges: i) identify the database operations performed by the application (*e.g.*, insert, read, update or delete); ii) determine the type of computation done at the column fields being accessed by these operations (*e.g.*, equality, order, sum, mean); iii) map the functionality requirements of each column to the encryption schemes available in the target secure database; and finally iv) output the corresponding secure database schema and configuration to the application developer or database administrator.

The general architecture of ATOCS is presented in Figure 3.1. The arrows show the complete flow of execution for the framework. ATOCS requires, as inputs from the developers or database administrators, the application source code and the targeted database system, namely its API and the set of available encryption schemes. Note that the database information can be reused for different applications using the same underlying database. This information is gathered by the `Info Gather` module, responsible for the first phase of the ATOCS pipeline, namely *Phase 1 - Information gathering.* This is the only phase that requires input from users.

The `Inspector` module is then responsible for starting *Phase 2 - Analysis* by automatically gathering the interactions between the application and the database system. This requires knowing the API of the database in order to distinguish the interaction of the application with other components (*e.g.*, the file system, remote procedure calls). After this step, the `Inspector` sends the application functions, that interact with the database (*i.e.*, that call database operations), to the `Code Analyser` module. By reducing the number of functions sent to the latter module, we are improving the performance for the next phase of the analysis.

The `Code Analyser` is responsible for determining the database columns that are indeed

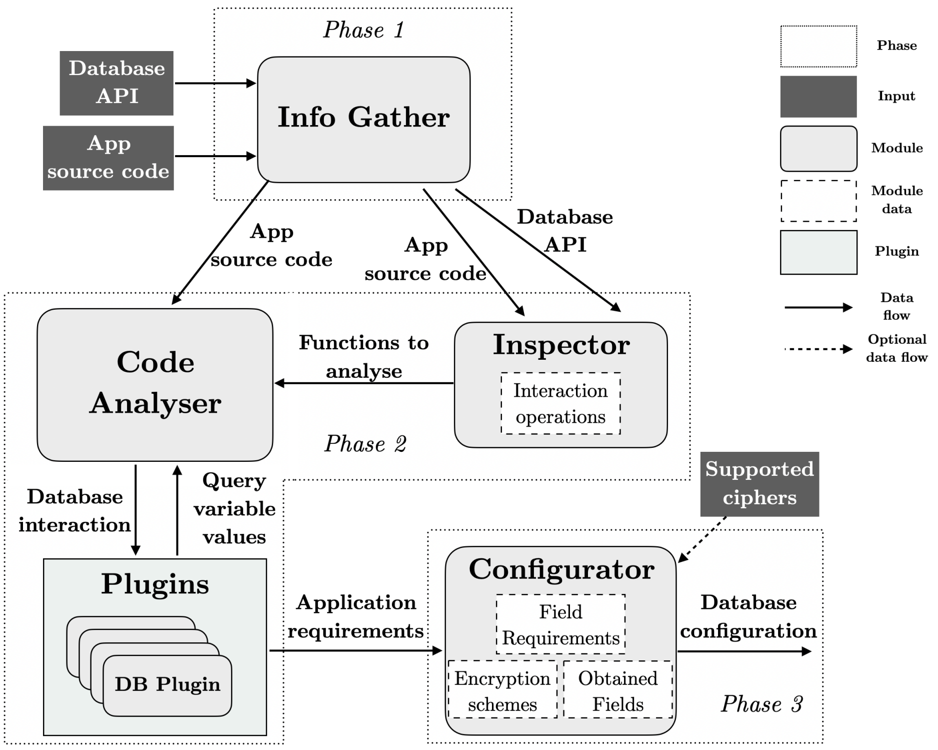


Figure 3.1: ATOCS architecture. Each of the dotted boxes represent the pipeline phase from where the module belongs to. The arrows show the data flow of the framework. The dotted arrows represent an optional flow.

being accessed, within a given function, and the type of computation done over those columns. For example, if an application function is calling a database operation that must find all the cells of a given column that match a specific value, then this column must support equality checking. Note that a single field can have multiple requirements, for example may need to support both equality and ordering.

In order to know the previous information, the `Code Analyser` needs to determine information specific to each database system (*e.g.*, the type of operation being performed and what type of arguments this operation requires). ATOCS design is modular and extensible and therefore to support a new database system it is only necessary to implement a new database plugin without requiring changes to the remainder of the modules.

Finally, after gathering all the previous requirements, these are sent to the `Configurator` module, which starts *Phase 3 - Configuration*. By merging the information obtained from *Phase 2* with the available encryption schemes, passed as user input in *Phase 1*, the `Configurator` generates the final secure database schema and configurations. Next, we describe each of the phases in more detail.

3.1 Phase 1 - Information gathering

As previously said, this is the only phase that requires input from users. Naturally, to analyse the code of a given application, the user has to provide ATOCS full access to the application's source code. Furthermore, users need to specify the database system that is being used by the application.

Note that ATOCS supports both secure databases and traditional databases, which do not necessarily include privacy-preserving storage and computation features. The motivation for including the latter is to allow developers to assess the viability and effort of porting an existing insecure application to a secure database. As an example, consider an application that uses SafeNoSQL [3] as the underlying database. In this case, ATOCS can be used to properly configure this database. However, we can also have an application that uses a traditional database system like HBase [1]. In this case, the developer may want to port his application to SafeNoSQL. Before doing so, he can use ATOCS to assess the encryption schemes required by the application. With this information, the developer can check whether the secure database system (in this case, SafeNoSQL) supports all the requirements of the application, before starting any implementation.

To be able to assess the interactions with the database, ATOCS must know the database API. However, it does not need the complete API but rather only the requests that perform operations over database columns. For example, a request to delete a database table is part of the API but it does not perform any operation over a database column, so it does not need to be considered by ATOCS. As another example, a read operation that filters the database rows by the value of a certain column (*e.g.*, retrieve database rows where column `C` as a value equal to `V`) needs to be considered.

ATOCS ships with a list of common encryption schemes and their computational properties (*i.e.*, equality, order, arithmetic operations), detailed in Table 2.1. Naturally, this can be expanded by users as necessary. If the application being analysed resorts to a non-secure database back-end, then the generated schema will choose the encryption schemes from this list that best suit the application needs. Otherwise, if the application is using a secure database system, then the user can specify, as input, the encryption schemes supported by that database. This way, ATOCS will only consider those encryption schemes when generating the final database schema and configurations. ATOCS warns the developer in case these encryption schemes are not able to satisfy the application requirements in terms of queries functionality (*e.g.*, if a database field requires an order property but there is no encryption scheme capable of preserving order in the ciphertext).

[1]Apache HBase, Apache, https://hbase.apache.org

3.2 Phase 2 - Analysis

After collecting the user's input, the `Inspector` analyses the application's code and collects all the application functions that call database operations as specified by the API.

These functions are then sent to the `Code Analyser` module, that analyses the application's code in order to identify the database operations (*e.g.*, insertion, read, update or delete) being called in each function. Then, for each database operation, the module also extracts the type of computation (*e.g.*, equality, order, arithmetic) done over different database columns. For example, let us consider a database read operation that obtains a set of rows ordered by the value of column `C`. ATOCS determines the computation, which is an order comparison, then finds the database field associated with it (in this case column `C`) and creates a property stating that, when encrypted, column `C` must preserve its original ordering. This methodology is used for every interaction between the application and the database. Note that it is possible for two different interactions to access the same column `C` while performing operations that require different properties. For example one might have the operation in the previous example that requires ordering in column `C` while other operation might require equality.

ATOCS keeps, at this stage, the required property for each operation. The decision of which encryption scheme to use is only done in the next phase.

Moreover, ATOCS's analysis can distinguish which database columns are actually being manipulated by the application and which are not, for each database interaction. As an example consider an application that inserts a row with several columns while subsequent operations only access and filter the value of a single column in that row. In this situation, ATOCS only analyses the column accessed by the application. This refinement is important because it allows to encrypt the columns that are not manipulated by the application, if any, to have the most secure encryption scheme supported by the target secure database system (*e.g.*, STD encryption).

Note that the `Code Analyser` requires processing logic that is dependent of the target database. This is achieved by our plugin architecture which keeps most of the logic generic and delegates the database specific logic to the plugin. The plugin is specific for a given database system and its role is to interpret the database operations found by the `Code Analyser`. This way, the `Code Analyser` is responsible for the generic application analysis, which is independent of the database system used, and the plugin is contacted by it when specific database information is required.

After collecting the previous information, for all the interactions with the database, ATOCS moves to the next phase.

3.3 Phase 3 - Configuration

Phase 3 starts when the application's code analysis is complete. The main goal of this phase is to match the requirements of each database column with the most appropriate encryption scheme supported by the target database. The key is to ensure that the chosen encryption scheme supports the different types of computation that may be performed over that database column.

It is possible that multiple encryption schemes are viable. For that reason, the user can order the encryption schemes by preference (*e.g.,* by the most secure or performant ones). With this input information, ATOCS is able to generate a database configuration based on the user's preferences.

Additionally, with the information obtained from the analysis phase, the `Configurator` module can also automatically infer possible optimisations. For example, certain schemes such as Order-preserving encryption (OPE) require a significant amount of computational time for encrypting and decrypting data when compared to probabilistic or deterministic schemes [3]. For this reason, in solutions such as SafeNoSQL, each database column encrypted with OPE is duplicated in another column encrypted with STD. Then, when a value encrypted with OPE must be retrieved to the client, instead of decrypting the OPE value, the STD one is decrypted instead. Such optimisation trades additional storage space for a considerable performance improvement.

However, this optimisation only makes sense if the column encrypted with OPE is indeed sent back to the client and decrypted. Otherwise, one will be just using additional storage space without having any performance benefits. ATOCS analysis can be used to determine whether a database column protected with OPE is going to be sent back to the client application or not. If not, the optimisation should not be used for that specific column. The `Configurator` module informs the developer of these situations, allowing her to select the most appropriate configuration to the target application. We further discuss this trade-off in the evaluation (Section 4.2).

3.4 Implementation

We implemented an open-source prototype of the above design that is able to analyse Java applications. The system itself is also implemented in Java. Although ATOCS is designed to support both SQL and NoSQL database systems, the prototype developed only focuses on the latter.In Section 3.7, we further discuss the challenges that need to be addressed in order to support SQL-based databases.

3.4.1 Plugins

We also implemented a database plugin for HBase [2]. HBase is an open-source distributed NoSQL database composed by two main components: the HBase Master and the RegionServers. The Master component is responsible for coordinating the cluster of Region Servers and balancing database requests across them. The RegionServers are responsible for persisting the database's data which can be sharded and replicated across different servers to ensure a scalable and highly-available design.

HBase tables are composed by rows and columns. A row is uniquely identified by a key and can have several columns associated with it. HBase client API supports insert/update (put), delete, read (get) and scan operations. It is also possible to define filters for scan operations in order to only bring rows whose columns match a specific condition (*e.g.*, column `C` has a value greater than `V`).

Moreover, ATOCS implementation supports a database plugin for the SafeNoSQL [3] database, which is based on the HBase system. Figure 3.2 presents the SafeNoSQL architecture resorting to HBase. This way, the proposed prototype can analyse both applications that use a secure database (SafeNoSQL) and a non-secure database (HBase). For the latter, ATOCS can report possible secure schemas and configurations that could be leveraged to port this set of applications to SafeNoSQL.

3.4.2 Inputs

As previously explained, ATOCS requires user input at *Phase 1.* This is done via three distinct YAML files. In the first file, users specify the directory containing the application's compiled Java classes, the entry points of the application (*i.e.*, the name and package of the main methods) and the database system being used. The second file is dedicated to the database API and must specify the database interaction methods by stating their name, declaring class and the type of operation they correspond to (*e.g.*, for a put operation in the HBase API, we provide the name of the operation: *put*, the declaring class: *org.apache.hadoop.hbase.client.Table* and state that this invocation corresponds to the operation: *PUT*). Note that this file needs to be built only once per supported database system. In the current prototype, we ship ATOCS with such a file for HBase and SafeNoSQL. Finally, a third YAML file contains the list of the encryption schemes, to be considered by ATOCS analysis, ordered by the user's preference. When two or more encryption schemes can be used for a given column, ATOCS will consider the user's preferences to select one of them. This file can be extended to specify new encryption schemes supported by the

[2]Apache HBase, Apache, https://hbase.apache.org

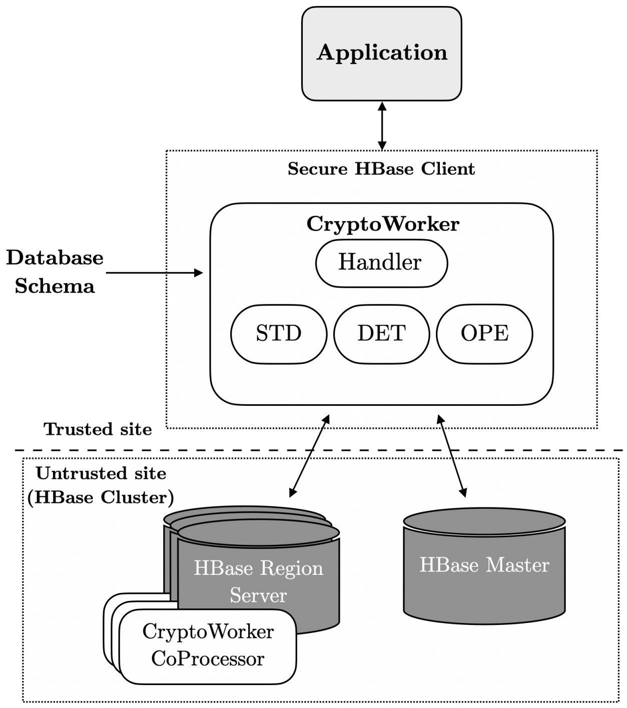


Figure 3.2: Implementation of SafeNoSQL [3] resorting to HBase. White modules belong to SafeNoSQL while the dark grey ones are from HBase.

underlying database system by stating their preserved properties. All three files are presented in the Appendix A.

3.4.3 Data structures

ATOCS leverages the Soot static analysis tool [3] to perform the low-level program analysis. Soot is used at the beginning of Phase 2 to analyse the application's source code and to generate an intermediate representation called Jimple. This representation is an improved version of Java byte code, containing all the classes of the application, that is more amenable for program analysis. From this representation Soot generates the method body graphs, builds dependency graphs and constructs a class hierarchy. However, Soot is not able to perform the Data Flow analysis itself, but rather provides detailed information about the application's dependencies and flow of data.

The `Inspector` queries the Soot dependency graph (called Call Graph) to obtain all the methods that contain a given database interaction. The Call Graph contains all the invocations performed by the application and knows in which method they are located. As for the `Code Analyser`, it requires the method graphs generated by Soot as well as the class hierarchy to perform a more tailored analysis of each method. The method graph contains all the method instructions (which are the nodes) and the paths between them (the edges). The class hierarchy

[3]Soot - A Java optimization framework, Sable Research Group, https://github.com/soot-oss/soot

Table 3.1: ATOCS ValueState subclasses. The first column states the class name and the second column the type of value it stores.

Class	**Value type stored**
LocalState	variable
ConstantState	constant
StringValueState	string constant
FieldRefState	class attribute
InvokeExprState	method invocation
NewExprState	constructor
ParameterRefState	method argument
StaticFieldRefState	enumerate

is aware of the class dependencies of the application.

Soot contains a class called Value to represent variables, constants, method invocations, etc. As a result, there was the need to introduce the notion of ValueState. A ValueState is a class that contains a Value object (Soot object), the scope method (the method where this value resides) and the current state of the scope method, alongside other information that might be useful during the analysis phase. The state is in fact a list of the method instructions that were analysed. There is a specific ValueState subclass for each of the values it can represent. Table 3.1 contains all the ValueState subclasses developed.

The main goal of the ValueState class is to retain the required information (scope method, value, list of method instructions, etc) about a certain value so that it can be later used to further analyse it. This is important to improve ATOCS's performance and to make the communication between the `Code Analyser` and the database `Plugins` smoother.

Consider the sample Java application in Algorithm 2 that is using HBase as the underlying database.

The application is only accessing a single table, and it is inserting data into the database in the `doPut` method, and retrieving data from the database in the `doScan` method. The full flow of operations will be detailed in the following section, so for now let us just focus on the `doScan` method (starting on line 6). When analysing this method, the `Code Analyser` looks for database operations. As a result, it finds on line 10 a `scan` operation. Because it requires specific database information to determine the requirements of this operation, it contacts the HBase plugin. To do so, it creates an *InvokeExprState* (which is a subclass of *ValueState* that represents a method invocation, in this case the `scan` method) and sends it to the database plugin. Notice that the `Code Analyser` does no further investigation on this method invocation, as it is delegated to ATOCS' HBase plugin.

Algorithm 2: HBase application example.

```
1  method doPut(String family, String qualifier, int value)
2      Table table = connection.getTable(TableName.valueOf("UserTable"));
3      Put p = new Put(Bytes.toBytes("row1"));
4      p.addColumn(Bytes.toBytes(family), Bytes.toBytes(qualifier),
         Bytes.toBytes(value));
5      table.put(p);
6  method doScan(String startRow, String stopRow)
7      Table table = connection.getTable(TableName.valueOf("UserTable"));
8      Scan s = new Scan();
9      applyFilter(s);
10     return table.scan(s);
11 method applyFilter(Scan s)
12     Filter f = new SingleColumnValueFilter(Bytes.toBytes("UserInfo"),
         Bytes.toBytes("Age"), CompareOp.GREATER, Bytes.toBytes(18));
13     s.setFilter(f);
```

The *Scan* object is found as an argument of the invocation (which can be obtained from the *InvokeExprState* object and is also represented as a *ValueState*). Although the plugin has to determine the table name where this operation is being performed (stated in line 7), let's assume it has already determined this, to simplify this example. The plugin now has to determine if there is any filter operation applied by this scan (which is done by invoking the `setFilter` method on the *Scan* object). For this reason, the plugin contacts the `Code Analyser`, sending it the scan *ValueState* for further investigation and asking if any filter operation was applied (also sending the signature of the `setFilter` method for reference). The `Code Analyser` will received a *ValueState* representing the *Scan* object, which in this case is a variable value (named *s*) and represented by a LocalState. With only the state information present in this *LocalState*, the `Code Analyser` is able to determine that *s* is used as an argument in the method `applyFilter` (line 9). Notice that if the `Code Analyser` only received the scan value instead of the *ValueState*, it would need to analyse the `doScan` method again in order to determine this. The `applyFilter` is then analysed and the `setFilter` invocation is found in line 13. An *InvokeExprState* for this invocation is created and sent back to the database plugin, which will inspect it to determine the type of filter applied.

This way, all the communication between the `Code Analyser` and the database plugins is done with the exchange of *ValueStates*. The diagram presented in Figure 3.3 shows the *ValueState* objects exchanged in the previous example between the `Code Analyser` and the database plugin. The *ValueStates* encapsulate all the required information for each module to operate, without the need to analyse the same method more than once.

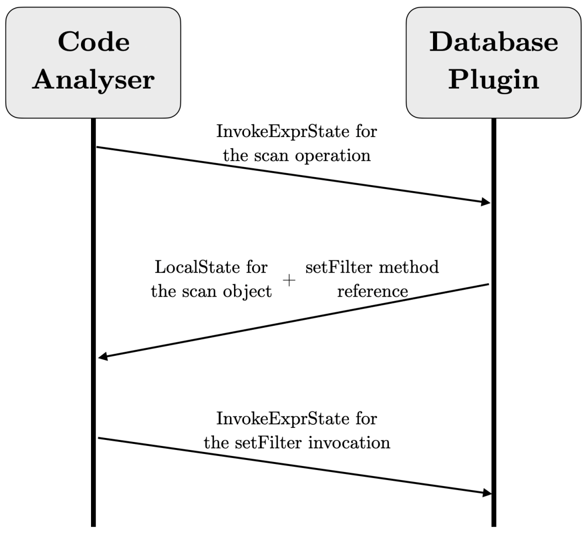

Code Analyser
Database Plugin
InvokeExprState for the scan operation
LocalState for the scan object
+
setFilter method reference
InvokeExprState for the setFilter invocation

Figure 3.3: *ValueStates* exchanged by the `Code Analyser` and database plugin when analysing the scan operation in Algorithm 2. The *ValueStates* exchanged to obtain the table name are omitted for simplicity.

3.5 Flow of operations

We now illustrate the flow of execution in ATOCS by considering a sample Java application, depicted in Algorithm 2. Once again, the application is inserting data into the database in the `doPut` method and retrieving data from the database in the `doScan` method.

The `Inspector` module gathers the Java methods that contain HBase operations, namely `doPut` and `doScan`. Note that the `applyFilter` does not interact with the database and hence it is not further analysed. Each of these methods is passed to the `Code Analyser` module that examines the method body, *i.e.*, goes through every instruction until it finds an HBase operation.

Put operation During the analysis of the `doPut` method, the `Code Analyser` pauses its analysis on line 5, when it reaches HBase's `put` operation. Since this is an interaction with the database that requires a specialised analysis, it is sent to the HBase plugin. The plugin determines that this is an insert operation and, as the next step, it needs to find on what table this operation is being done. Since the plugin is aware of the HBase API, it knows that the table name can be consulted when the *Table* object is obtained by calling the `TableName.valueOf` method.

Thus, the *Table* object is sent back to the `Code Analyser` in order to get the value for the first argument of the `valueOf` method. The `Code Analyser` keeps the history of operations already visited and finds that the table name was previously assigned (line 2). From this assignment operation, it obtains the value for the `TableName.valueOf` method, which is

UserTable, and returns it back to the plugin.

The plugin is aware that a put operation requires checking if the key being inserted already exists or not, *i.e.*, equality checking, while the columns do not require any sort of computation over their values. This way, the plugin defines that the keys for the table named UserTable must preserve equality checking.

Scan operation For the doScan method, the Code Analyser finds an HBase's scan operation at line 10 and sends this interaction to the HBase plugin. The table where the scan is being applied is discovered in a similar fashion to the one described for the put operation. A scan can be restricted to a range of table rows or it can be a full scan across all the rows of a table. If a range is specified, *i.e.*, a start key and an end key, the database table must preserve the order between keys. For a full scan, the relative order between rows does not need to be preserved since all rows will be retrieved by the client.

The sample application we are considering is doing a full scan with a filter operation (line 13). This filter is defining that the client will only receive scanned rows whose Age column has a value higher than 18. The HBase plugin is aware that a scan operation may define filters so it queries the Code Analyser to determine if any filters were assigned to this scan by sending the *Scan* object and the setFilter method. Although the *setFilter* on the scan object is present in another method (line 11, method applyFilter), the Code Analyser is able to track it by following the code interactions with the *Scan* object.

After reporting to the HBase plugin that the scan operation contains a filter, the plugin sends another query to the Code Analyser in order to find the three arguments of the filter constructor (line 12). Each value is then sent back to the HBase plugin. The first two arguments describe the table (UserTable) and column (Age) where the filter is going to be applied. The third argument shows that an order comparison (GREATER) is being done. This way, the plugin defines that the column Age of table UserTable must preserve the order of its values.

Note that the analysis of an HBase get operation is similar to the one conducted for the scan operation, while the analysis of a delete operation is similar to the one conducted for the put operation. When the analysis phase is completed, the Configurator module receives the information about what table properties must be ensured and combines this information with the available encryption schemes and the properties that these preserve. In this example, keys must preserve equality, thus requiring a deterministic encryption scheme (*e.g.*, DET), while the column Age must preserve order, thus requiring an order-preserving encryption scheme (*e.g.*, OPE). For this example, the Configurator will produce the final configuration schema depicted in Table 3.2. As discussed previously, if the table (UserTable) has other columns that

Table 3.2: ATOCS output for the HBase sample application.

Table	Field	Encryption Scheme
UserTable	keys	DET
	UserInfo:Age	OPE
All other columns		STD

do not need to preserve any processing properties, these will be assigned with a probabilistic encryption scheme (*e.g.*, STD).

3.6 Optimisations

As described in Section 3.3 we implemented an optimisation for the OPE scheme used in the SafeNoSQL secure database. In short, this optimisation is used since the OPE decrypting process requires a significant amount of computational time. The idea is that a column that requires OPE is duplicated and also stored encrypted with STD (which has faster decryption time). When an OPE field is requested by the client application, only the STD version is decrypted. However, the field can still be used for order operations due to the OPE scheme. This optimisation is not required if the OPE field is never retrieved from the database back to the client as it would only add additional storage space to the database without any performance benefits. This is what ATOCS tries to determine, whether or not the fields that require OPE are ever retrieved from the database to the client.

To do so, we need to establish how this is represented in the application code. In the case of HBase, the `get` and `scan` operations are the main way to obtain data from the database. In both operations, the fields to be obtained are specified using the `addColumn` or `addFamily` methods. If no fields are specified, then all fields are obtained. When an OPE field is obtained we mark this field so that the optimisation is applied to it.

Algorithm 3: HBase application simple scan operation example.

```
method doScan(String family, String qualifier)
    Scan s = new Scan();
    if !family.isEmpty() then
        s.addColumn(Bytes.toBytes(family), Bytes.toBytes(qualifier));
    table.getScanner(s);
```

Nevertheless, accomplishing this is not as straightforward as it may seem at first. As an example, consider the example in Algorithm 3. ATOCS uses static analysis, so it has to consider every possible execution path, from the `addColumn` invocations to the execution of the operation itself (`getScanner`, at line 5). In this case there are two paths. The first is in case the *if* condition is false, so the addColumn method is never executed, hence all the fields will be obtained. The second occurs if the condition is true, in which case only the column represented by the family and qualifier provided will be obtained. Since we have two different paths, we have to take a worst case scenario approach, *i.e.* we consider the union of the obtained fields from each possible path to account for any possible code execution. In this case, we consider that all fields are obtained and if any of them require OPE, then the optimisation is applied.

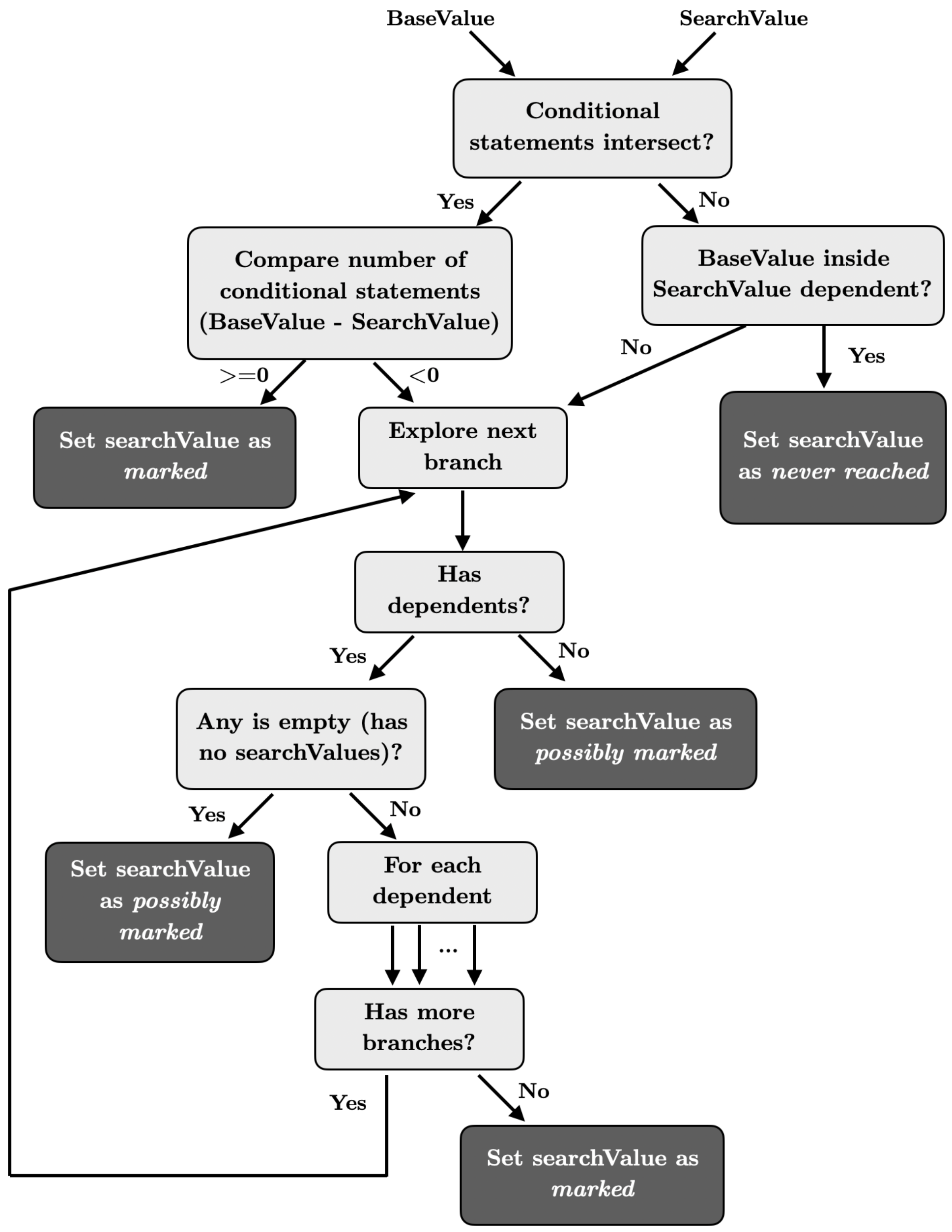
In order to do this kind of analysis we are required to know whether a certain instruction is inside a conditional block or not. However, the Soot framework is not able to provide this information. For this reason, while a method is being analysed by the Code Analyser (to minimise the impact on performance), ATOCS determines whether each instruction is inside a conditional block or not by inspecting the method bytecode. To accomplish this, when ATOCS finds any instruction that begins a block (named a `Conditional Statement`), it alerts the instruction which comes afterwards that they are inside this conditional block. Once again, this information is stored in the ValueState object that represents each instruction so that is can be later obtained.

With this information we were able to develop an algorithm, described below, that is capable of determining what are the fields obtained from the database by the application in simple cases such as Algorithm 3, but also in more complex ones like Algorithm 4. In the latter, because every path executes the invocation in line 10, there are no possible paths where no `addColumn` method is invoked. Remember that if a path does not specify a column to obtain then all columns are obtained. As a result, the values obtained are the union of the three `addColumn` invocations in this method.

Algorithm 4: HBase application complex scan operation example.

```
1  method doScan(String family, String qualifier)
2      Scan s = new Scan();
3      if family.isEmpty() then
4          //doNothing();
5      else if qualifier.isEmpty() then
6          s.addColumn(Bytes.toBytes(family), Bytes.toBytes("Qualifier1"));
7      else
8          s.addColumn(Bytes.toBytes(family), Bytes.toBytes(qualifier));
9      end
10     s.addColumn(Bytes.toBytes("Main"), Bytes.toBytes("Qualifier0"));
11     table.getScanner(s);
```

The diagram in Figure 3.4 presents the main logic of our algorithm. The BaseValue is compared to every SearchValue. In the problem presented in Algorithm 4, the SearchValues are the `addColumn` method invocations and the BaseValue is the execution of the database



BaseValue
SearchValue
Conditional statements intersect?
Yes
No
Compare number of conditional statements (BaseValue - SearchValue)
BaseValue inside SearchValue dependent?
>=0
<0
No
Yes
Set searchValue as *marked*
Explore next branch
Set searchValue as *never reached*
Has dependents?
Yes
No
Any is empty (has no searchValues)?
Set searchValue as *possibly marked*
Yes
No
Set searchValue as *possibly marked*
For each dependent
...
Has more branches?
Yes
No
Set searchValue as *marked*

Figure 3.4: State diagram representing the algorithm to determined the obtained database fields. The BaseValue is compared to every SearchValue. A Conditional statement is an instruction that starts a conditional block (*eg.* an if or switch statement). A Dependent is a conditional statement which execution depends on the execution of another conditional statement, *eg.* an if statement may have a dependent, which would be its else statement. The grey boxes represent the final states, which are merged in the end to provide a unique output.

operation, in this case the `getScanner` method. A Conditional statement is an instruction that starts a conditional block (*eg.* an if or switch statement). A Dependent is a conditional statement which execution depends on the execution of another conditional statement, *eg.* an if statement may have a dependent, which would be its else statement. In a switch block, each case is a dependent of every other case.

In short, the algorithm compares the BaseValue to the SearchValue, in order to determine if there is any feasible path between the two, and hence decide whether an `addColumn` method was invoked before the scan operation was performed. This is done by comparing the conditional statements of each value. The conditional statements represent the conditional block (*eg.*, an if, else, for or while statement) where the value is inserted in. Each value has a list of conditional statements. Depending on the conditional statements of each value, there can be many different execution paths. With this approach our algorithm is able to account for all of them.

Let us consider Algorithm 4 and analyse it with the help of the diagram in Figure 3.4. As previously stated, the SearchValues are the `addColumn` methods (lines 6, 8 and 10) and the BaseValue is the `getScanner` method (line 11). The algorithm presented in the diagram has to be executed for each SearchValue and the BaseValue is always the same. We start with the SearchValue in line 6 and compare it to the BaseValue. The two values intersect if their conditional statements match, even if one has more conditional statements than the other. An empty value intersects with any other value. In this case, the BaseValue has no conditional statements and the SearchValue has one (which is the *else if* statement). For this reason we consider that they intersect. Comparing the size of the conditional statements lists we determine that their difference is -1 (BaseValue - SearchValue). Then we explore the next branch, that is defined by the first conditional statement that only belongs to the SearchValue. In this case, the *else if* statement. This branch has two dependents, the *if* and *else* statements. The *if* statement has no SearchValues inside and for that reason we set this SearchValue (line 6) as *possibly marked.* Because the SearchValue in line 8 is a dependent of the previously analysed SearchValue, we can consider that it was already analysed and also set as *possibly marked.* This is done to improve the algorithm performance. Lastly, we have the SearchValue in line 10. Here, the conditional statements also intersect, since both lists are empty. Comparing the two results in a difference of 0, thus setting this SearchValue as *marked.*

To conclude this algorithm, we just need to merge the results obtained. Here we only have to consider two situations. The case where no SearchValue is set as *marked* or the case where there is at least one set as *marked.* In the first scenario all SearchValues, and hence all addColumn invocation, may not occur and for that reason we assume all database columns are obtained. As for the second scenario, since there is one SearchValue that is executed, we consider that all the database columns specified by the SearchValues that are *marked* and *possibly marked* are obtained. Recall that if one column is specified to be obtained (by the *addColumn* method), then only the specified columns are obtained. However, if no columns are specified then all of them are obtained. For this reason, the output of the algorithm in this example is a merge of the specified columns in lines 6, 8 and 10. However, if line 10 was deleted, then the output would be all the database columns.

3.7 Discussion

ATOCS was designed with the intent of automating the configuration of secure databases. As previously stated, there is no automated process to configure these systems in the current state of the art. The solution presented leverages code analysis tools to statically analyse an application and infer the database operations performed. With this information ATOCS is able to determine the required properties for each database field that need to be preserved when the field is encrypted. Furthermore, ATOCS can also leverage code analysis to automate database's tuning. The modular and extensible design of the framework makes it easier to add support for new database systems.

Both SQL and NoSQL database plugins can be added to the framework. All the modules were designed and implemented with this in mind. Nevertheless, the ATOCS prototype developed only contains NoSQL plugins (for HBase and SafeNoSQL databases). This is mainly due to the fact that the implementation of a SQL plugin requires further research. SQL queries can be very complex and this plugin would be required to interpret them, even with the presence of prepared statements for example. We believe ATOCS opens a new area of research with the implementation of such plugins.

Chapter 4

Evaluation

In this chapter, we present ATOCS's evaluation. We focus on two different aspects: i) ATOCS's accuracy and efficiency when analysing large codebases (Section 4.1), and ii) the impact on performance of the optimisations suggested by ATOCS (Section 4.2).

All the tests below were performed on a cluster of servers equipped with an Intel Xeon CPU with 8 cores at 2.13GHz, 40GB of RAM and a 900GB HDD disk. All the results are the average of five independent runs.

4.1 Efficiency and Accuracy

We start by assessing ATOCS's efficiency by measuring the time it takes to analyse the codebase of four applications. The first is a *synthetic* HBase application developed by us for testing purposes. We also evaluate ATOCS with the following three real applications. HBaseTinkerGraph [1] is an implementation of a TinkerPop graph [2] using HBase as the storage backend. Twitbase [3] [4] is a simplified clone of the Twitter social network implemented on top of HBase. Yahoo! Cloud Serving Benchmark (YCSB) [5] is a NoSQL database benchmark, which we configured with the workload used in the SafeNoSQL [3] secure database system.

We analysed the four applications following the workflow introduced in Section 3, and present the execution time in Table 4.2. Interestingly, despite having applications with disparate codebase sizes and complexity, ATOCS is able to perform all the analysis in less than 45 seconds, which is a very good value when compared to the time a programmer would take to do the same analysis manually — which is prone to be incomplete and/or incorrect.

[1]HBaseTinkerGraph, Dpv91788, https://github.com/dpv91788/HBaseTinkerGraph
[2]Blueprints, TinkerPop, https://github.com/tinkerpop/blueprints
[3]TwitBase, HBase in action, https://github.com/hbaseinaction/twitbase
[4]TwitBase, MapR Demos, https://github.com/mapr-demos/hbase-to-maprdb/tree/master/sync-api
[5]YCSB, Yahoo, https://github.com/brianfrankcooper/YCSB

Table 4.1: Encryption schemes available for each application tested.

Application	**Encryption schemes**
Demo App	STD, DET, HOM, FPE, SE, OPE
HBaseTinkerGraph	
Twitbase	
YCSB	STD, DET, OPE

Table 4.2: ATOCS execution time for each application.

Application	**Lines of Code**	**Execution Time (sec)**
Demo App	~270	36.41 ± 0.43
HBaseTinkerGraph	~1400	40.67 ± 0.66
Twitbase	~1300	41.96 ± 0.24
YCSB	~12.5K	44.49 ± 0.40

Next, we focus on assessing the accuracy of ATOCS *i.e.*, the extent to which it can identify the interactions with the database and infer the appropriate encryption schemes. To achieve this, we manually analysed the codebase of the three real applications and determined the right encryption scheme for each database column — similarly to what a developer would have to do nowadays in the absence of ATOCS. The available encryption schemes for each application are depicted in Table 4.1. The results are presented in Table 4.3, Table 4.4 and Table 4.5 for HBaseTinkerGraph, TwitBase and YCSB applications, respectively.

For each table, the *Table/Field* column identifies the name of the database table and the respective field (key or column name), the *Operation* column identifies the set of operations done over that field, and the *Property* column identifies the property required by the encryption scheme. Finally, columns *Required Encryption* and ATOCS *Output* identify the encryption scheme selected by us manually and by ATOCS, respectively. As it is possible to observe, despite the application's large codebase and complexity, ATOCS was able to determine the correct encryption scheme to be used, matching our manual configuration. The only exception is in the `usertable/Appointment:Date` column of YCSB (Table 4.5) where ATOCS suggested a different configuration than our manual analysis. This difference is discussed in detail in the next section. Note that for the cases where there are no property-preserving needs over a given database field, ATOCS defaults to the most secure encryption scheme available in the target secure database (*e.g.*, STD encryption). We omitted these fields in the tables to improve their legibility. Moreover, in case multiple encryption schemes provide the required properties for a certain field (*eg.*, DET and OPE, in case of an equality property), ATOCS chooses one according

Table 4.3: HBaseTinkerGraph interactions with the database system.

Table / Field	**Operation**	**Property**	**Required Encryption**	**ATOCS output**
graphs keys	put	Equality	OPE	OPE
	range scan	Order		
	get	Equality		
graphs qualifier names	qualifier filter	Partial Comparison	SE	SE
properties keys	put	Equality	OPE	OPE
	range scan	Order		
properties vertex	equality filter	Equality	DET	DET
properties edge	equality filter	Equality	DET	DET

to the developer preferences (which are present in the input files).

4.2 Performance Optimisations

The automated and systematic code analysis performed by ATOCS opens the door to a series of optimisations at the application and database schema levels. In this section, we discuss one possible optimisation that can be made to the YCSB application, when considering SafeNoSQL as the database back-end, and evaluate its impact on performance. In the manual analysis discussed in the previous section, we concluded that the YCSB database column *Appointments:Date* should use the OPE encryption scheme while ATOCS suggested to used instead OPE/STD, *i.e.* OPE plus STD.

The rationale for this is the following. As previously explained in Section 3.3, decrypting OPE columns is expensive in terms of computational resources and time. Thus SafeNoSQL adopts a duplication strategy where each database column encrypted with OPE is accompanied by the same value protected with STD. Then, when the client needs to decrypt an OPE field, it decrypts instead the STD value. As previously discussed, recall that this optimisation only makes sense if the column encrypted with OPE is indeed sent back to the client and decrypted there. Otherwise, one will be just using additional storage space without having any performance benefits.

Therefore, after analysing the application code, ATOCS only suggests this optimisation when it infers that the client can potentially read the OPE column. This was the case for the YCSB application considered in the previous section.

To showcase the impact on enabling or disabling the previous optimisation, we conducted the

Table 4.4: TwitBase interactions with the database system.

Table / Field	**Operation**	**Property**	**Required Encryption**	**ATOCS output**
users keys	put	Equality	DET	DET
	full scan	None		
	get	Equality		
twits keys	put	Equality	OPE	OPE
	range scan	Order		
	get	Equality		
folows keys	put	Equality	OPE	OPE
	range scan	Order		
folowedBy keys	put	Equality	OPE	OPE
	range scan	Order		
users info:password	regex filter	Partial Comparison	SE	SE
users info:tweet_count	increment	Algebraic Operation	HOM	HOM

Table 4.5: YCSB interactions with the database system.

Table / Field	**Operation**	**Property**	**Required Encryption**	**ATOCS output**
usertable keys	put	Equality	DET	DET
	full scan	None		
	get	Equality		
	delete	Equality		
usertable Appointment:Date	order filter	Order	OPE	OPE/STD

Table 4.6: Impact of OPE optimisation in application performance.

#	OPE Optimization	Reading OPE field	Throughput (op/min)	Total Op	DB size (MB)
1	On	Yes	27.49 ± 2.95	1636	79.90
2	On	No	33.93 ± 2.88	1968	79.90
3	Off	Yes	0.03 ± 0.45	2	74.80
4	Off	No	32.70 ± 3.33	1951	74.80

following experiment. We deployed the YCSB application with the SafeNoSQL database in 6 servers in our cluster: one server holds the YCSB application and SafeNoSQL client, one server holds the HBase Master and the remaining 4 servers hold an HBase RegionServer each[6].

We performed four experiments resulting from the combination of enabling/disabling the OPE optimisation discussed above for two distinct scenarios. In the first one, the YCSB application does not read the OPE encrypted field (*Appointments:Date*) while in the second scenario this field is read (*i.e.*, it is decrypted at the client). Each experiment consists of a full scan-only workload that runs for one hour. The results are presented in Table 4.6.

If the application reads the column encrypted with OPE (experiments 1 and 3) then we observe that the SafeNoSQL OPE optimisation brings a substantial advantage by increasing throughput from 0.03 operations/minute to 27.49 operations/minute. When the application does not read the column encrypted with OPE (experiments 2 and 4) then the optimisation brings no throughput benefits (both experiments achieve approximately 33 operations/minute) as the client does not need to pay the cost of decrypting an OPE encrypted value. The last column in Table 4.6 presents the database size of each test, where the tests with the optimisation on use more space due to the duplication of the OPE fields. However, here we are only considering the case of always obtaining an OPE field or never obtaining one, which may not be realistic. To complement this, we performed a second analysis where, for each test, we varied the percentage of OPE fields obtained. Table 4.7 presents the results. Notice that test 1, 5, 6 and 10 correspond to the test 2, 1, 4 and 3 of Table 4.6 respectively.

Once again, it is clear that when the optimisation is turned on, the throughput does not take a huge impact even when we increase the percentage of OPE fields obtained. Without the optimisation (tests 6 to 10) we can conclude that, even with a small percentage of OPE fields, the throughput greatly decreases. This leads us to conclude that the OPE optimisation is useful in most scenarios (all tests where at least one OPE field is obtained).

In Table 4.7 some of the standard deviation values of the throughput without the optimisation

[6]SafeNoSQL uses a Vanilla HBase cluster deployment as the back-end.

Table 4.7: Impact of varying the OPE fields obtained in application performance, with the OPE optimisation.

#	%OPE obtained	%STD obtained	OPE Optimization	Throughput (op/min)	Total Op	DB size (MB)
1	0%	100%	On	33.93 ± 2.89	1968	79.90
2	1%	99%	On	32.77 ± 2.99	1952	79.90
3	10%	90%	On	32.25 ± 3.01	1920	79.90
4	50%	50%	On	30.30 ± 2.44	1803	79.90
5	100%	0%	On	27.50 ± 2.96	1636	79.90
6	0%	100%	Off	33.69 ± 3.33	1951	74.80
7	1%	99%	Off	7.70 ± 13.92	459	74.80
8	10%	90%	Off	0.27 ± 2.45	17	74.80
9	50%	50%	Off	0.08 ± 0.71	5	74.80
10	100%	0%	Off	0.03 ± 3.33	2	74.80

are high (experiment 7, which obtains OPE fields 1% of the times). This is mainly due to the discrepancy of the throughput values obtained during the one hour test (which were measured every 10 seconds). During the decryption of an OPE field the throughput was 0, since this decryption is very time consuming. However, when decrypting the STD fields (which are much faster) the throughput would greatly increase. For this reason we had two very different values for the throughput measured during the one hour test, depending of whether decrypting OPE or STD, which results in this high standard deviation.

We now analyse the resource consumption at the client node for every test in Table 4.7, presented in Table 4.8. As expected, when the client retrieves a field encrypted with OPE and has to decrypt it (experiments 7 to 10), CPU consumption increases. The decrease in the network bandwidth consumption is a consequence of the slow performance (operations per minute) of this setup. Resource consumption is identical across the remaining experiments (experiments 1 to 6) because these setups are only decrypting values encrypted with STD.

We also measured resource usage at the server side. All experiments yielded similar results in terms of RAM consumption, namely 1.5 GB. CPU usage was similar across all setups, except for experiments 7 to 10 where the usage dropped from 7% to 4%, mainly due to the decrease on the number of operations per minute done by the client.

The setup using the optimisation in Table 4.7 (experiments 1 to 5) occupies additional storage space for the database (79.9 versus 74.8 MB). Although this may not seem like a big discrepancy, this size would be even greater if one considers a large database with several columns using the OPE encryption scheme. Also note that the values presented not only include the space occupied by the database fields but also the metadata and other information required for the

Table 4.8: Client resource consumption for each test presented at Table 4.7

#	CPU (%)	RAM (MB)	Network recv (KB)	Network send (KB)
1	4.91	929.03	6341.34	62.72
2	4.87	929.77	6353.22	63.58
3	5.11	978.12	6377.54	64.56
4	6.46	989.79	7465.53	64.72
5	5.20	986.00	8374.20	62.70
6	4.90	932.21	6042.92	60.10
7	8.55	1154.16	1434.83	14.42
8	9.76	1078.86	56.31	0.62
9	9.50	905.12	19.52	0.35
10	9.54	925.57	9.62	0.21

Table 4.9: Database size overhead based on the percentage of OPE fields.

OPE (%)	Overhead (%)
0%	0%
10%	15%
20%	28%
30%	40%
40%	51%
50%	61%
60%	70%
70%	78%
80%	86%
90%	93%
100%	100%

correct function of the database system. We present in Table 4.9 the overhead of the database size with regards to the percentage of OPE fields stored in the database. This table considers a database schema with 20 different fields (like the ones used in the SafeNoSQL [3] experiments) where each field's plaintext occupies 100 Bytes. Here we are only considering the space occupied by the database fields and not considering any metadata. The percentage of overhead is always greater or equal to the percentage of OPE fields in the database. If we apply this to a large database with thousands of rows the impact is very significant.

Since the STD scheme uses AES256 and an OPE ciphertext occupies double the size of the corresponding plaintext, the database size when the optimisation is turned on can be obtain from the following formula:

$$db\_size = \sum_{i=0}^{nOPE} (2 \cdot plt_i) + \sum_{j=0}^{nSD+nOPE} (plt_j + (32 - (plt_j \bmod 32)))$$

where:

- db_size is the size of the database with the OPE optimisation turned on.
- $nOPE$ is the number of OPE fields.
- nSD is the number of STD or DET fields.
- plt_i is the plaintext size of field i in bytes.
- plt_j is the plaintext size of field j in bytes.

Note that ATOCS analysis is able to detect if the OPE column is being retrieved to the client or not. Thus, it can assess if the SafeNoSQL optimisation is potentially desirable, due to performance reasons (experiments 1 and 3 from Table 4.6), or if enabling this optimisation does not bring any performance advantage while requiring extra storage space (experiments 2 and 4 from Table 4.6).

To sum up, in this chapter we demonstrated that ATOCS yields an accurate analysis of arbitrary applications, creating the best configuration for the application's underlying database system with the encryption schemes available, similar to what an expert programmer would do, but automatically. Results show that ATOCS performs its analysis and outputs the proper database configuration in less than 45 seconds even for complex applications with thousands of lines of code. Moreover, during this analysis, ATOCS is able to gather information about the application which can be used to guide the user in the proper configuration of database optimisations. In the case of the SafeNoSQL OPE optimisation, ATOCS can assess if it is potentially desirable, due to performance reasons (experiments 1 and 3 from Table 4.6), or if enabling this optimisation does not bring any performance advantage while requiring extra storage space (experiments 2 and 4 from Table 4.6).

Chapter 5

Conclusions

Secure database systems provide security and privacy guarantees over data that resides in an untrusted location (such as a cloud provider). Data is kept encrypted and operations are performed over the corresponding ciphertexts. This is possible by combining different encryption schemes which support operations over encrypted data such as equality, ordering or arithmetic operations. By exploring the state of the art of these systems, it is clear that they all have a common shortcoming which is the fact that their configuration has to be done manually. This means assigning each database field with a certain encryption scheme that respects the functionality requirements of the application while providing the best security and performance guarantees. As applications become more complex and need to address data security, privacy and compliance concerns, the process of selecting the appropriate encryption scheme for each of the database fields becomes increasingly difficult, time-consuming and error prone.

To mitigate these issues, this dissertation presents ATOCS, a framework that automates the configuration of secure database systems. ATOCS is able to analyse applications with large codebases in a few dozens of seconds and suggest the appropriate set of encryption schemes without requiring the manual labour of an expert programmer. This greatly reduces the possibility of the occurrence of human errors while assuring that the system is configured to provide the best security guarantees for the operations performed by the application. Recall that ATOCS only requires a small number of inputs from the developer or system administrator about the application and database in use at the beginning of its execution and fully automates the rest of the analysis.

Not only is ATOCS useful to configure a secure database system but it is also useful to reconfigure it in case any changes are made to the application code. Due to the fast execution time of the framework, this can easily be added at the end of the development pipeline of an application to ensure that it is always deployed with the best configuration available. Moreover, the detailed analysis performed by ATOCS allows it to suggest possible optimisations to the application and database back-end. Aside from this, ATOCS also supports the analysis of applications that use traditional databases, which do not necessarily include privacy-preserving storage and computation features. This allows the system administrators to assess the viability of porting their application to a secure database system.

This way ATOCS automates the configuration a secure database system, making this a simpler process that does not require an expert in databases and security, less prone to errors, more accurate and efficient.

5.1 Future Work

5.1.1 Research Directions

We believe ATOCS opens the door to several interesting research directions, which we discuss next. While the detailed static analysis can spot optimisations such as the ones identified in Section 4.2, enriching the analysis with information about the workload such as the relative frequency of operations over a column can result in fine grained optimisations that explore different performance, security, and resource usage trade-offs. As an example, consider the case where an application performs an operation that retrieves an OPE field from the database to the client only 0,1% of the times. In the current analysis, our framework will assume that the OPE optimisation is required, since we are retrieving an OPE field. However, since this only happens 0,1% of the times, it might not be worth to increase the database size just due to this operation but rather pay the cost of the expensive OPE operation once in a while. These type of decisions are only possible with workload information.

Aside from this, a trace analysis can also be helpful to improve ATOCS accuracy. Although the static analysis assures that the framework is considering every possible scenario and hence guaranteeing that every operation performed by the application will be feasible, it requires that the operands of the database interactions are present in the application's code and are not dynamically obtained, for instance through a configuration file obtained when the program starts. However, with traces we are able to account for application inputs that are dynamically obtained and only consider the executions that are actually traversed by the application, hence improving the precision of the analysis and yielding a finer grained database configuration. Nevertheless, we need to take into account that, since we are considering the input values from the traces and not all the possible scenarios, we may end up with incomplete results.

As stated in Section 3.4, the ATOCS prototype developed only focuses on NoSQL database systems. However, the design of ATOCS is intended to be suitable for either SQL or NoSQL. Although the analysis of applications using SQL databases considerably differs from NoSQL ones, the main structure and modules of the current ATOCS prototype would not need to be altered. To support a SQL database one would only need to develop a new plugin capable of analysing such system. This might seem simple at first but is actually a research challenge when we consider that it needs to interpret complex SQL queries (with prepared statements, for example).

5.1.2 Engineering Improvements

In terms of the OPE optimisation developed, the ATOCS output can also be extended in order to provide the database size overhead imposed by using this optimisation. To do so, a detailed database schema with the size of each field would be required as an input of the framework. The developer could then decided if she considers the optimisation worthwhile or not. This would be an extension of the developed framework as no changes to the current code would be required.

Following this extensibility idea, a new module could be added to the ATOCS architecture to generate the specific configuration file for different secure database systems. In the current state of the framework, ATOCS outputs the required encryption schemes for each database field. However, this new module could generate the specific database schema file to be used by the secure database system. This file would be specific for each secure database, so different plugins should be developed to account for every system supported.

Due to the modular design of ATOCS, it is also easy to add new plugins to the system in order to support new database systems. Although both SQL and NoSQL database plugins can be added, the implementation of a SQL plugin requires further research as discussed in the previous section.

Bibliography

[1] David Reinsel-John Gantz-John Rydning. The digitization of the world from edge to core. *Framingham: International Data Corporation*, 2018.

[2] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100. ACM, 2011.

[3] Ricardo Macedo, João Paulo, Rogério Pontes, Bernardo Portela, Tiago Oliveira, Miguel Matos, and Rui Oliveira. A practical framework for privacy-preserving nosql databases. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 11–20. IEEE, 2017.

[4] Joan Daemen and Vincent Rijmen. Specification for the advanced encryption standard (aes). *Federal Information Processing Standards Publication*, 197, 2001.

[5] Morris Dworkin. Recommendation for block cipher modes of operation. methods and techniques. Technical report, National Inst of Standards and Technology Gaithersburg MD Computer security Div, 2001.

[6] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order preserving encryption for numeric data. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 563–574. ACM, 2004.

[7] John Black and Phillip Rogaway. Ciphers with arbitrary finite domains. In *Cryptographers' Track at the RSA Conference*, pages 114–130. Springer, 2002.

[8] Caroline Fontaine and Fabien Galand. A survey of homomorphic encryption for nonspecialists. *EURASIP Journal on Information Security*, 2007:15, 2007.

[9] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*, pages 44–55. IEEE, 2000.

[10] Mohsen Amini Salehi, Thomas Caldwell, Alejandro Fernandez, Emmanuel Mickiewicz, Eric WD Rozier, Saman Zonouz, and David Redberg. Reseed: Regular expression search over encrypted data in the cloud. In *2014 IEEE 7th International Conference on Cloud Computing*, pages 673–680. IEEE, 2014.

[11] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. Orthogonal security with cipherbase. In *CIDR*. Citeseer, 2013.

[12] Jin Li, Zheli Liu, Xiaofeng Chen, Fatos Xhafa, Xiao Tan, and Duncan S Wong. L-encdb: A lightweight framework for privacy-preserving data queries in cloud computing. *Knowledge-Based Systems*, 79:18–26, 2015.

[13] Muhammad Naveed, Seny Kamara, and Charles V Wright. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 644–655. ACM, 2015.

[14] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018.

[15] Anjana Gosain and Ganga Sharma. Static analysis: A survey of techniques and tools. In *Intelligent Computing and Applications*, pages 581–591. Springer, 2015.

[16] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, page 35, 2011.

Appendix A

ATOCS input files

Listing A.1: Application config file example for SafeNoSQL.

```
# App Configuration file

# Database system used by the application.
database: SafeNoSQL

# Application directories containing the .class files to analyse as well as the
    libraries' .class or .jar files.
directoriesToAnalyse:
    - app/src/classes

# Main classes of the application, which contain a main method.
entryPoints:
    - Main
```

Listing A.2: Database interactions config file example for SafeNoSQL.

```
#SafeNoSQL version 2.2.0
#
#Format:
#---
#className: fullname
#methods:
#  - name: methodname
#    operation: type of database operation performed
#    args:
#      - - fieldType # - - represents a set of arguments for the method defined
    above
#        - fieldType
#      - - fieldType

className: pt.uminho.haslab.safeclient.secureTable.CryptoTable
methods:
  - name: put
    operation: PUT
    args:
      - - org.apache.hadoop.hbase.client.Put
      - - java.util.List
  - name: get
    operation: GET
    args:
      - - org.apache.hadoop.hbase.client.Get
      - - java.util.List
  - name: getScanner
    operation: SCAN
    args:
      - - org.apache.hadoop.hbase.client.Scan
  - name: delete
    operation: DELETE
    args:
      - - org.apache.hadoop.hbase.client.Delete
      - - java.util.List
  - name: append
    operation: APPEND
    args:
      - - org.apache.hadoop.hbase.client.Append
  - name: mutateRow
    operation: MUTATE
    args:
      - - org.apache.hadoop.hbase.client.RowMutations
  - name: checkAndMutate
    operation: CHECKMUTATE
    args:
      - - byte[]
        - byte[]
  - name: batch
    operation: BATCH
    args:
      - - java.util.List
        - java.lang.Object[]
```

Listing A.3: Ciphers config file example for SafeNoSQL.

```
# Ciphers Configuration file

# Encryption schemes ordered by preference.
cipherPreferences:
    - STD
    - DET
    - OPE

# [Optional] Encryption schemes supported by the secure database system to be
    used. Requires the specification of the properties preserved by each scheme.
supportedCiphers:
  - name: STD
    properties:
  - name: DET
    properties:
      - EQUALITY
  - name: OPE
    properties:
      - EQUALITY
      - ORDER
```