ATOCS: Automatic Configuration of Encryption Schemes for Secure NoSQL Databases

David Ferreira
U. Lisboa & INESC-ID
davidjferreira@tecnico.ulisboa.pt

João Paulo
INESC TEC & U. Minho
joao.t.paulo@inesctec.pt

Miguel Matos
U. Lisboa & INESC-ID
miguel.marques.matos@tecnico.ulisboa.pt

2021 17th European Dependable Computing Conference (EDCC) | 978-1-6654-3671-7/21/$31.00 ©2021 IEEE | DOI: 10.1109/EDCC53658.2021.00017

***Abstract*—Secure databases have emerged to securely store and process sensitive data at untrusted infrastructures (*e.g.*, Cloud Computing). To be secure and efficient, the encryption schemes used by these systems must be carefully chosen. Indeed, this task requires expertise both in databases and security, and is currently being done manually, which is time-consuming and error-prone and can lead to security violations, poor performance, or both.**

This paper presents ATOCS, a novel framework that analyses the applications' code and, from the inferred requirements, determines the best combination of encryption schemes and related configurations for the underlying secure NoSQL database. Its design is modular and extensible thus facilitating the support of different applications and database solutions. Our evaluation with real-world applications shows that ATOCS is fast (it takes 44 seconds to analyse more than 12K LoC), accurate, and simplifies the configuration of secure databases.

***Index Terms*—security and privacy, databases, code analysis**

I. Introduction

Database Management Systems (DBMS) are widely used to store and query large amounts of data. Since information stored in a database can be sensitive, data confidentiality is a prime concern for every company resorting to this technology.

Additionally, there is a business benefit for companies to move 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. However, moving sensitive data to the cloud is in direct confrontation with the privacy concern discussed above, as the cloud provider has full control over the infrastructure and data. This conundrum led to the development of secure database systems such as CryptDB [2] and SafeNoSQL [3].

Secure databases allow applications to manipulate and query data residing in an untrusted location (*e.g.*, cloud provider) while offering security and privacy guarantees over the data and queries. This is achieved by leveraging a combination of encryption schemes that support operations over encrypted data such as equality, ordering or arithmetic operations.

Selecting the appropriate encryption schemes for a deployed database is therefore a critical task. Namely, if the encryption schemes are not chosen correctly, the secure database will not be able to perform the operations correctly (*e.g.*, performing a sum with an encryption scheme that does not support algebraic operations). Even if the encryption schemes provide the required functionality, they might still not be the most adequate in terms of performance or security. Additionally, choosing an incorrect encryption scheme might lead to latent bugs, which can be very hard to identify and correct. As an example, if the database performs a comparison between two probabilistic cryptograms that originated from the same plaintext data, the comparison will not fail but will wrongly report that the values being compared are different.

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. For example, if one considers a *simple* application whose database schema has 10 tables, each with 10 columns, choosing the right encryption scheme for the 100 columns quickly becomes a daunting and error-prone task. Note that to select the most appropriate encryption scheme, it is not enough to consider only the column's data type, but also *all* the queries and operations that the application does at each column. We argue that this crucial decision, with far reaching implications in the security, dependability and performance of the system, must not be done manually by humans, but should rather be fully automated.

Therefore, we propose ATOCS, a framework that given the application's code is able to automatically derive the most appropriate encryption schemes for the underlying NoSQL database. This is achieved by relying on code analysis to automatically infer the types of operations performed over data 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 user's goals. By fully automating this process, ATOCS removes human error and leads to more secure and performant database applications.

Our 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. The analysis takes 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.

In more detail, this paper makes the following contributions:

- ATOCS, a framework that automatically analyses the code of applications and determines the most appropriate encryption schemes and related database optimisations;

978-1-6654-3671-7/21/$31.00 ©2021 IEEE
DOI 10.1109/EDCC53658.2021.00017

- an open-source implementation[1] of ATOCS including plugins for two NoSQL databases: HBase and SafeNoSQL;
- a detailed evaluation with three real-world applications.

The rest of the paper is organised as follows. Section II overviews the state of the art of secure databases and code analysis tools. Section III describes the architecture and implementation of ATOCS. Section IV presents the experimental evaluation while Section V concludes the paper.

II. Background and Related Work

The simplest way to provide secure storage and processing at cloud services is to encrypt all the information, with a strong probabilistic encryption scheme (*e.g.*, AES), before sending it to the server [4]. However, this does not allow executing any kind of processing (equality or arithmetic operations over the ciphertext) at the database server. To perform such operations, data needs to be transferred to a trusted environment, decrypted and then processed. Naturally, such approach exhibits poor performance which degrades as the amount of data grows.

A. Secure Databases

To overcome these shortcomings, secure databases resort to encryption schemes that preserve different properties of the original plaintext in the corresponding ciphertext. These allow operations (*i.e.*, equality, order, arithmetic) to be performed directly over the ciphertexts stored at the database server.

a) Encryption Schemes: The Standard Encryption (STD) scheme uses a probabilistic cipher such as AES in CBC mode [4]. This scheme has the best security guarantees, however, it does not support any operations over the generated 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 at the database server. Order-preserving Encryption (OPE) [6] guarantees that if a plaintext P1 is greater than a plaintext P2, the respective ciphertext C1 is greater than ciphertext C2. As a result, both equality and order operations are possible over encrypted data. Format-preserving Encryption (FPE) [7] ensures that the ciphertexts maintain the size and type of the plaintext, while some implementations of this scheme also preserve the equality of ciphertexts. Homomorphic Encryption (HOM) [8] enables arithmetic operations among ciphertexts, while Searchable Encryption (SE) [9], [13] enables searching for keywords in ciphertexts.

Some of these schemes leak information to an attacker and hence provide weaker security guarantees. For example, when using DET, an attacker can infer whether two ciphertexts correspond to the same plaintext or not [2], [3].

b) Overview of Secure Databases: The previous encryption schemes are combined by secure database systems to provide data confidentiality and privacy in cloud deployments. As depicted in Figure 1, most of the current solutions consider an honest-but-curious untrusted environment that can inspect

[1]https://github.com/miguelammatos/ATOCS

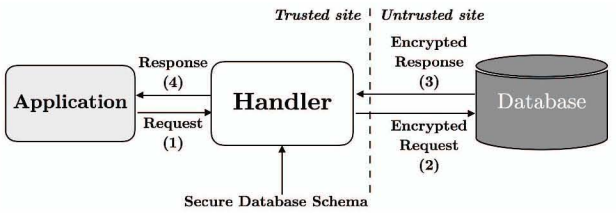

Fig. 1: Generic architecture for a secure database system.

TABLE I: Example of a secure database schema.

Table	Column	Encryption Scheme
User	Name	STD
	Age	OPE
Group	ID	DET

the information stored and being processed at the database server. Contrarily, the database client is deployed on a trusted site where data encryption and decryption can be done safely. The latter holds the application and the Handler which mediates the interactions between both sites. The untrusted site hosts the database server(s) where encrypted data is stored and processed. It is crucial that this data is never decrypted on the untrusted site, so that no information is leaked.

To illustrate a typical flow of execution consider the database schema (Table I) that has two tables `User` and `Group`. When the application issues a query (1) (*e.g.* obtain the names of users with age greater than 40), it is intercepted by the Handler which encrypts the data contained in the query. The Handler resorts to the secure database schema to determine the encryption scheme for each column. In this case, it encrypts the value 40 with OPE as this is the scheme chosen for the `Age` column (Table I). Next, the Handler submits the query to the untrusted database server (2). If the database schema is correctly configured, the ciphertexts retain the required properties from the original plaintexts and operations can be performed transparently over the ciphertexts.

The response from the database server is sent to the Handler (3) which decrypts it (resorting to the database schema once again) before sending it back to the application. In our example, we retrieved the `Name` column, so this will be decrypted with STD (Table I). This is then sent to the application in plaintext (4), which resides on the trusted site.

The secure schema is therefore a key component for these databases, as it enables the Handler to know what encryption scheme must be used for each database column. However, assigning encryption schemes is a complex task that requires knowing exactly what operations are being done over each database column. Thus, this analysis is highly dependent on the targeted application, making a manual approach unfeasible for applications with a large code base.

c) Secure Database Systems: Secure solutions such as CryptDB [2] and SafeNoSQL [3] encrypt database columns with different combinations of encryption schemes to provide a full-fledged database engine. CryptDB is applied to SQL databases and follows an onion encryption approach where each database column is encrypted with multiple encryption

schemes. The encryption scheme with the best security guarantees is in the outer layer while the inner layer holds the encryption scheme with the most functionality. SafeNoSQL is applied to NoSQL databases and was developed with a modular design in mind regarding the addition of new encryption schemes. Each column is protected with a single encryption scheme that is chosen based on the type of computation, security and performance required by operations over that column.

Cipherbase [10] adds a trusted hardware module (TM) at the untrusted site. The main idea is to use encryption schemes that provide the best security guarantees while preserving the necessary properties from the plaintext data. The TM is responsible for performing database operations that cannot be achieved through the cryptographic primitives.

The security and performance guarantees of these systems hinge on a careful specification of the database schema and encryption schemes. However, no system automates this crucial step. We address this shortcoming with ATOCS.

B. Code Analysis Tools

Code analysis tools inspect applications' code for various purposes such as detecting bugs, measuring code quality or understanding the applications' functionality. In this paper, we are interested in the application's interactions with the database system. This allows narrowing the scope of the analysis and hence enable a very efficient approach even for applications with a large code base. We consider any application request done to the database system as an interaction.

Regarding code analysis tools, these can either be dynamic or static [11]. Dynamic analysis requires the program to run against previously defined input values that cover the relevant parts of the program. For this reason, dynamic analysis can miss important behaviours generated by inputs that were not accounted for. In static analysis, the program does not need to be executed because only the application's code is analysed. Since this approach does not rely on program inputs, it assumes every possible execution path, hence fully covering the program's logic. Given that dynamic analysis cannot explore every path of an application, we restrict ourselves to static analysis. In particular, we consider two distinct techniques: Data Flow analysis and Symbolic Execution.

Data Flow analysis considers a graph that connects all the possible paths between nodes, where each node represents a program instruction [12]. The connections between nodes represent the possible execution flow of information. The graph is then used to see how values propagate throughout the nodes. One disadvantage of this approach is the fact that it considers every path between the nodes, even if they are not feasible (without taking variable values into account). As a result, Data Flow analysis can lead to false positives.

Conversely, Symbolic Execution distinguishes between feasible and non feasible paths and only traverses the first, which makes it more precise. 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

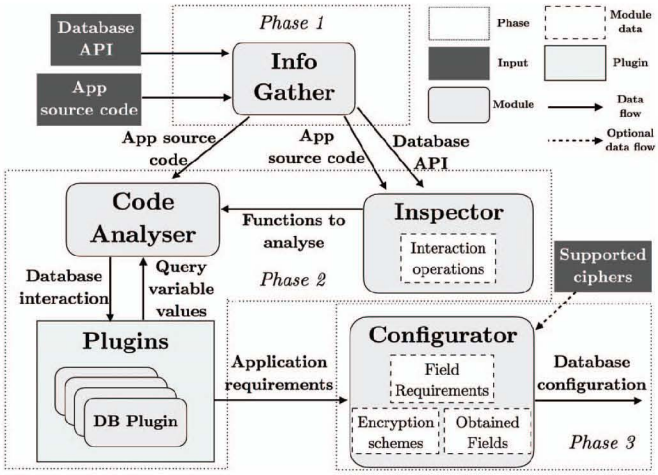

Fig. 2: ATOCS architecture and flow of requests.

of each branch the results are not merged but rather continue as two different execution paths. In each step, 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 several times and greatly increasing the number of paths it has to explore. Therefore, Symbolic Execution can be very complex and its performance can quickly degrade in applications with large code bases such as the ones we target. For this reason, in this work, we rely on Data Flow analysis.

III. ATOCS

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

Figure 2 provides an overview of ATOCS flow. The `Info Gather` module, responsible for the *Phase 1 - Information gathering*, requires as inputs the application code and the target database system, namely its API and the set of encryption schemes. This is the only phase that requires input from users.

The `Inspector` module starts *Phase 2 - Analysis* by automatically gathering the interactions between the application and the database. After this step, the `Inspector` sends the application functions have any interaction 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.

The `Code Analyser` is responsible for determining the database columns that are in fact accessed, and the type of computation done over those columns. Finally, the previously gathered requirements are sent to the `Configurator` module, which starts *Phase 3 - Configuration*. By merging the information obtained from *Phase 2* with the available encryption

schemes, provided in *Phase 1*, the `Configurator` generates the final secure database schema and configurations. Next, we describe each of the phases in more detail.

A. Phase 1 - Information gathering

This is the only phase that requires input from users. To analyse the code of an application, the user has to provide ATOCS full access to the application's source code and specify the used database system. 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` has 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), which can be expanded by users as necessary. Also, ATOCS warns the developer in case the provided encryption schemes are not able to satisfy the application requirements in terms of queries functionality (*e.g.*, if a database column requires an order property but there is no encryption scheme listed that is capable of preserving this property).

B. 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 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, 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 column 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 (*e.g.*, equality and order).

ATOCS's analysis can distinguish which database columns are actually being manipulated by the application and which are not. 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. This refinement is important because it allows suggesting a stronger non-property preserving encryption scheme (*e.g.*, STD) for the columns that are not manipulated by the application.

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 being used, and the plugin is contacted by it when database specific information is required.

1) Plugins: The database plugin is responsible for interpreting the database operations analysed by the Code Analyser. The Code Analyser sends to the plugin the function invocations that represent database interactions. From this, the plugin determines the concrete operations being performed (*e.g.*, insert, filter), as it is aware of the database API. If the values of the operands are required for the plugin to understand on which columns the operations are performed, then the plugin will query the Code Analyser to determine these values. Thus, the plugin does not need to perform any code analysis. It only queries the Code Analyser for the values it requires to infer the database operations being performed and the computational properties that must be preserved.

2) Data structure: ATOCS leverages the Soot (https://github.com/soot-oss/soot) static analysis tool, at the beginning of *Phase 2 - Analysis*, 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 (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 is aware of the class dependencies of the application.

Soot contains a class called Value to represent variables, constants, method invocations, etc. In ATOCS we 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.

The main goal of the *ValueState* class is to encapsulate all the required information (scope method, value, list of method instructions, etc) about a certain value for each module to

Algorithm 1: 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);
```

operate. This is important to improve ATOCS's performance (since it does not need to analyse the same method more than once) and to make the communication between the `Code Analyser` and the database `Plugins` smoother.

3) Example: Consider the sample Java application in Algorithm 1 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 in the `doScan` method. The full flow of operations is detailed in the following section, so for now let us focus on the `doScan` method (lines 6—10). When analysing this method, the `Code Analyser` looks for database operations. It finds on line 10 a `scan` operation which retrieves all the rows that match the given criteria. 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.

To simplify the example, let us assume that the plugin has already determined the table name where this operation is being performed. The plugin now has to determine if there is any filter operation applied by this scan (which is done by invoking the `setFilter`). It contacts the `Code Analyser` and sends the scan *ValueState* for further investigation while asking if any filter operation was applied (also sending the signature of the `setFilter` method). 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). 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.

C. Phase 3 - Configuration

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

In general, 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.

With the information obtained from the analysis phase, the `Configurator` module can also automatically infer possible optimisations. For example, schemes such as OPE require a significant amount of computational time for encrypting and decrypting data when compared to STD or DET schemes [3]. For this reason, and in certain scenarios, it might make sense to duplicate a column and encrypt one copy with OPE and other with STD, for instance. 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 is only relevant if the column encrypted with OPE is retrieved by the client, otherwise it is just wasting storage space. ATOCS is able to detect these patterns and inform users about possible optimisations, as further discussed in Section IV-B.

D. Implementation

ATOCS's prototype is implemented in Java and targeted towards the analysis of applications using the same programming language. Although it is designed to be generic and to support both SQL and NoSQL database systems, our prototype only focuses on the latter. To add support for a new database only requires writing a new plugin, which for NoSQL databases is a matter of extending and adapting the plugins we provide. Given the richness and complexity of SQL, writing a plugin for a SQL database is more challenging, and which we leave for future work. ATOCS supports the analysis of multiple applications using the same database backend. It just requires that all code bases are provided together so that the analysis can span through all of them and produce an aggregated secure schema and configuration. In addition, ATOCS can provide support for the development pipeline of applications. Namely, it can be used to analyse incremental code updates and to validate the requirements with respect to the database schema.

1) Plugins: We implemented a plugin for HBase (https://hbase.apache.org), a NoSQL database. HBase tables are composed by rows that can have several columns and are uniquely identified by a key, The HBase client API supports insert/update (put), delete, read (get) and scan operations. Also, filters can be applied to scan operations in order to only bring rows whose columns match a specific condition (*e.g.*, value of column `C` is greater than `V`).

Our implementation also supports a database plugin for the SafeNoSQL [3] database. The motivation for supporting both is the following. If the targeted application is using SafeNoSQL, ATOCS can be used to properly configure the database and define its schema. If the application is using a non-secure database (HBase), developers can rely on ATOCS to determine the schema and encryption schemes required by the application and therefore assess what secure databases would provide the needed features to fully support the application.

The implemented plugins have to extend an abstract class *DatabasePlugin*, which has the necessary methods to connect them to the other ATOCS components. The plugins receive a database operation to analyse through the *analyseDbInteraction* method. To obtain concrete values for the operation arguments received, these can use the available methods from the `Code Analyser`. Aside from this, the plugins are responsible for sending to the `Configurator` all the requirements inferred from the operations analysed. The lines of code (LOCs) that must be written to implement a database plugin depend on the dimension of the database API. The HBase API, for example, is rather vast, mainly due to the Filters that can be performed on the scan and get operations (which will influence the properties that must be preserved for each database column). Thus, the ATOCS's HBase plugin has around 1800 LOCs. The SafeNoSQL plugin (because SafeNoSQL itself extends HBase) only requires more 18 LOCs to account for the operation that obtains a database table, which is different in SafeNoSQL.

2) Inputs: 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*). This file needs to be built only once per supported database system. 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 underlying database system by stating their preserved computational properties.

E. Flow of operations

We now illustrate ATOCS's flow of execution by considering the sample Java application depicted in Algorithm 1. The application is inserting data into the database in the `doPut` method and retrieving data in the `doScan` method.

The `Inspector` module gathers the Java methods that contain HBase operations, namely `doPut` and `doScan`. 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), 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.

Scan operation: For the `doScan` method, the `Code Analyser` finds an HBase's `scan` operation at line 10 and sends this interaction to the 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 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 chosen encryption schemes must preserve the order between keys. For a full scan, the relative order between rows does not need to be preserved.

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 order.

TABLE II: ATOCS execution time for different applications.

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

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). If the table (`UserTable`) has other columns that do not need to preserve any processing properties, these are assigned with a probabilistic encryption scheme (*e.g.*, STD).

IV. Evaluation

ATOCS's evaluation is focused on two different aspects: i) ATOCS's accuracy and efficiency when analysing large codebases, and ii) the impact on performance of the chosen configurations. All experiments 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.

A. 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. The other three are real applications using HBase as the storage backend. HBaseTinkerGraph[2] is an implementation of a TinkerPop graph[3]. Twitbase[4] is a simplified clone of the Twitter social network. Yahoo! Cloud Serving Benchmark (YCSB)[5] is a NoSQL database benchmark, which we configured with the workload used in SafeNoSQL [3].

We chose the latter three applications because they are open-source and have 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 (Table II). Given these results, we expect ATOCS to also be efficient for applications with more LoC which, nevertheless, is an interesting future work task.

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. Also, we performed a manual analysis which consisted on carefully reading the code of each application, finding the operations issued to the database system and determining which database columns were affected by each operation – similarly to what any other developer would have to do in the absence of ATOCS. With this information, an encryption scheme was chosen for each database column that would allow all the operations found to be feasible. We considered the following encryption schemes: STD, DET, OPE, FPE, HOM and SE. Results are presented in Table III for TwitBase.

TABLE III: TwitBase interactions with the database system.

Table / Column	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		
followedBy 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

The *Table/Column* identifies the name of the database table and the respective database column, the *Operation* identifies the set of operations done over that column, and the *Property* 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. For the cases where there are no property-preserving needs over a given database column, *i.e.* no operation is done over that column, ATOCS defaults to the most secure encryption scheme available (*i.e.*, STD). We omitted these columns from the table to improve its legibility (there are 8 columns in this situation for Twitbase).

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 same results were found for the other applications, with the exception of one column for the YCSB application (*Appointments:Date*) where ATOCS suggested OPE/STD (*i.e.* OPE plus STD) and the manual analysis suggested OPE. This difference is discussed next.

B. Performance Optimisations

As previously explained in Section III-C, SafeNoSQL adopts a ciphertext duplication strategy where each database column encrypted with OPE is accompanied by the same value protected with STD. This is done because the decryption time for the STD encryption scheme is considerably faster than the one for OPE. However, this optimisation, and the extra storage space it requires, is only worthwhile if the column encrypted with OPE is indeed sent back and decrypted at

[2]HBaseTinkerGraph, https://github.com/dpv91788/HBaseTinkerGraph.
[3]TinkerPop, Blueprints, https://github.com/tinkerpop/blueprints
[4]HBase in action, TwitBase, https://github.com/hbaseinaction/twitbase.
[5]Yahoo, YCSB, https://github.com/brianfrankcooper/YCSB

TABLE IV: Impact of SafeNoSQL OPE optimisation.

#	%OPE / %STD	OPE Optimisation	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	100%/0%	On	27.50 ± 2.96	1636	79.90
4	0%/100%	Off	33.69 ± 3.33	1951	74.80
5	1%/99%	Off	7.70 ± 13.92	459	74.80
6	100%/0%	Off	0.03 ± 3.33	2	74.80

the client premises. Therefore, after analysing the application code, ATOCS only suggests this optimisation when it infers that the client can potentially read the OPE column.

To showcase the impact on enabling or disabling the previous optimisation, we deployed the YCSB application with the SafeNoSQL database in 6 servers in our cluster: one server holds the YCSB application and SafeNoSQL client, another server holds the HBase Master and the remaining 4 servers hold an HBase RegionServer each (SafeNoSQL uses a Vanilla HBase cluster deployment as the backend).

We performed distinct experiments resulting from the combination of enabling/disabling the OPE optimisation discussed above and the percentage of times that the same OPE column is retrieved by the client. Each experiment consists of a scan-only workload that runs for one hour. We ran half of the tests with the optimisation enabled and the other half with the optimisation disabled. For each configuration, we measured the throughput (op/min), the total number of operations performed and the database size (MB).

As depicted in Table IV, it is clear that when the application does not read any OPE columns (experiments 1 and 4) then the optimisation brings no throughput benefits (both experiments achieve approximately 33 operations/minute) and the total number of operations performed is very similar as the client does not need to pay the cost of decrypting an OPE cipher.

With the optimisation enabled, the throughput value is very similar for all tests. As described before, with the optimisation enabled the system duplicates the OPE columns and encrypts the same value with STD. Due to this, in tests 1 to 3, the system is only decrypting STD values at the client and hence obtaining similar performance results.

However, even with a small percentage of OPE columns, the throughput with the optimisation disabled greatly decreases when compared to the test with the optimisation enabled. As an example, for the tests where only 1% of the columns obtained are encrypted with OPE, the throughput decreases from 32.77 operations/minute to 7.70 operations/minute (tests 2 and 5, respectively). This leads us to conclude that, for all tests where at least 1% of the columns obtained are using OPE, the optimisation results in high performance gains.

The setup using the optimisation in Table IV (experiments 1 to 3) 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 greater if one considers a large database with several columns using OPE.

V. Conclusion

This paper presents ATOCS, a framework that automates the schema configuration of secure NoSQL database systems. As applications become more complex and need to address data security, privacy and compliance concerns, the process of selecting the most appropriate encryption schemes becomes increasingly difficult, time-consuming and error prone.

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. Moreover, the detailed analysis performed by ATOCS allows it to suggest possible optimisations to the application and database backend.

We believe the framework proposed in this paper opens the door to several interesting research directions, such as the addition of a new plugin that supports SQL databases. Aside from this, while the detailed static analysis can spot some optimisations, enriching the analysis with information about the workload such as the relative frequency of operations over a column could result in fine grained optimisations that explore different performance, security, and resource usage trade-offs. A trace analysis could also be used to provide ATOCS with dynamically obtained inputs, which would improve the accuracy of the framework.

Acknowledgements: Work supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under projects Angainor (LISBOA-01-0145-FEDER-031456) and UIDB/50021/2020, and by project AIDA (POCI-01-0247-FEDER-045907), co-financed by the European Regional Development Fund (ERDF) through the COMPETE 2020 Program and by FCT under CMU Portugal. We thank our shepherd Marco Vieira for his help on improving the paper.

References

[1] Reinsel et al., The Digitization of the World From Edge to Core, tech. rep., International Data Corporation, 11 2018.
[2] Popa et al. CryptDB: protecting confidentiality with encrypted query processing. SOSP, 2011.
[3] Macedo et al. A practical framework for privacy-preserving NoSQL databases. In SRDS, 2017.
[4] Pub, N. F. Specification for the Advanced Encryption Standard. In FIPS 197, 2001
[5] Dworkin, M. Recommendation for block cipher modes of operation. methods and techniques. In NIST-SP-800-38A
[6] Agrawal et al. Order preserving encryption for numeric data. In ACM SIGMOD 2004.
[7] Black et al. Ciphers with arbitrary finite domains. In RSA Conference, 2002.
[8] Fontaine et al. A survey of homomorphic encryption for nonspecialists. IN EURASIP JIS, 2007.
[9] Song et al. Practical techniques for searches on encrypted data. In SSP 2000.
[10] Arasu et al. Orthogonal Security with Cipherbase. In CIDR, 2013.
[11] Baldoni et al. A survey of symbolic execution techniques. In ACM CSUR, 2018.
[12] Lam et al. The Soot framework for Java program analysis: a retrospective. In CETUS 2011.
[13] Salehi et al. Reseed: Regular expression search over encrypted data in the cloud. In CLOUD 2014.