MIRES: Recovering Mobile Applications based on Backend-as-a-Service from Cyber Attacks

Diogo Lopes Vaz

ABSTRACT

Many popular mobile applications rely on the Backend-as-a-Service (BaaS) cloud computing model to simplify the development and management of services like data storage, user authentication and notifications. However, vulnerabilities and other issues may lead to malicious operations on the mobile application client-side and malicious requests being sent to the backend, corrupting the state of the application in the cloud. To deal with these attacks after they happen and are successful, it is necessary to remove the immediate effects created by the malicious requests and subsequent effects derived from later requests. In this paper, we present MIRES, an intrusion recovery service for mobile applications based on BaaS. MIRES uses a two-phase recovery process that restores the integrity of the mobile application and minimizes its unavailability. We implemented MIRES in Android and with the Firebase platform and made experiments with 4 mobile applications that showed results of 1000 operations reverted in less than 1 minute and with the mobile application inaccessible only for less than 15 seconds.

KEYWORDS

Intrusion Recovery, Mobile Computing, Backend-as-a-Service, Cloud

1 INTRODUCTION

Mobile applications are programs that run on mobile devices, typically *smartphones* or *tablets*. Most mobile applications rely on remote services and resources provided by servers, often designated *clouds*, to support their functioning. Recently several frameworks/platforms have been appeared to support the development and execution of mobile applications. These frameworks allow integrating code running on devices with remote services through APIs. These remote services are executed on the cloud and allow storing the state of the application, sending notifications and authenticating users. To simplify the development of these features, a new cloud service model, named *Backend-as-a-Service* (BaaS) [7, 11, 21], has emerged, allowing developers to configure the backend of a mobile application without implementing it from the ground up. In fact, today many popular mobile applications are based on BaaS, e.g., the Duolingo platform for learning languagesand the Lyft car sharing platform.

Mobile applications can contain *vulnerabilities*, e.g., due to improper user input validation, or other errors made by developers in designing and/or writing code. These weaknesses can be explored by malicious users with the intent to corrupt the state of the application stored in the backend, leading to *intrusions*. A 2019 study revealed that 60% of the mobile application vulnerabilities were on the client side, where two thirds were medium/high risk [31].

This paper is about *intrusion recovery*, i.e., about reverting the effects of the intrusion on the state of the application. A simple solution would be to periodically backup the application state, creating a *snapshot*, and, when an intrusion occurs, to replace the state with the last snapshot. However, this solution would lead to data loss, as backups are almost always outdated, e.g., hours or days, depending on their frequency. Database recovery does better by considering not only snapshots but also the statements since the last snapshot, which are stored in a *log* [13]. However, statements are low-level events that are hard to correlate to higher-level operations and databases store only the statements since the last snapshot.

This work follows a more recent line of research on *intrusion recovery* that aims to revert the effects of intrusions on the application layer by logging the requests or higher-level operations made [5]. Our approach involves generating *compensating transactions* [24] based on log analysis, that will revert the effects of the intrusion without loss of legitimate data. Intrusion recovery has been studied in different contexts, such as web applications [1, 8], databases [10, 26], operating systems [19], email services [5], and cloud computing [27, 29]. However, to the best of our knowledge, no previous work focused on recovering *mobile* applications. Also, no previous work focused on recovering applications based on the BaaS model.

To fill this gap, in this work we present the **M**obile Applications **I**ntrusion **Re**covery **S**ervice (MIRES), an intrusion recovery service for mobile applications that use BaaS. The MIRES recovery model is based on a two-phase process that aims to reconstruct the corrupted data concurrently to users' interaction with the backend, by restoring the integrity of the systems' state with a focus on maintaining the availability of the mobile application system. Besides the main intrusion recovery mechanism, MIRES also provides an user recovery mechanism that allows the application users to recover from mistakes.

In terms of security properties [4], the objective is therefore to regain *integrity* after an intrusion and to do it with low impact on *availability*; on the contrary, the objective is *not* to achieve *confidentiality* as MIRES operates after the intrusion happened. Confidentiality protection requires runtime mechanisms that are out of the scope of this paper [6, 15]. Our work also does *not* focus on intrusion detection, that is orthogonal to intrusion recovery; other mechanisms could be used for this purpose [3, 12, 38].

We implemented MIRES in Android and the Firebase platform and evaluated it experimentally using 4 applications: a social network, a messaging app, shopping list app and a contact tracing application. MIRES was able to recover 1000 malicious operations in less than 1 minute, letting the mobile application inaccessible only for less than 15 seconds.

MobiQuitous '20, December 07–09, 2020, Darmstadt, DE
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The main contributions of this paper are: (1) an intrusion recovery service for an emerging cloud service model, focused on mobile applications; (2) a different approach to previous offline recovery models, that aims to increase the availability of the system; (3) a recovery mechanism that allows users to recover their own actions.

2 BACKEND-AS-A-SERVICE

Backend-as-a-Service (BaaS) [7, 11, 21], also know as *Mobile Backend-as-a-Service* (MBaaS), is a cloud service model that provides a set of ready to use application-logic services that automates and speeds up the backend development process of web and mobile applications. In this paper, the focus is only on *mobile applications*. BaaS aims to provide scalable and optimized backend infrastructures, where all responsibilities of running and maintaining the backend infrastructures are outsourced to the BaaS vendor, leaving only the development of the mobile application to the user of the platform. Examples of BaaS platforms are Firebase[*], Back4App[†] and Parse.[‡]

Typically, a BaaS service model provides a set of application-common services like: *data and file storage* for storing structured data and files, *push notification* to send notifications to the application, *user management* to authenticate the users, *application analytics* to scrutinize the crashes and performance of the application, and *cloud functions* [25] to run simple and single-purpose code on the server-side, invoked via *HTTP endpoints* or when specific *cloud infrastructure events* occur, like database changes, for example. BaaS services are integrated by the mobile application via custom *software development kits* (SDK) and *application programming interfaces* (APIs).

Figure 1 represents the architecture of a BaaS platform. Each mobile application, like A and B, running in a mobile device, is pre-associated – usually through a configuration file – with a specific virtual environment called *container*, assuring the use of the containers' services by the mobile application. Containers are virtually isolated from the others and contain all the *resources* – code, services and configurations (e.g., database permissions and settings) – used by the mobile application system. A mobile application system is identified in the platform by a *global unique identifier* that is sent in the mobile application requests and among the resources inside the containers.

2.1 Application Implementation with BaaS

Mobile applications that use a BaaS backend have their *state* distributed between the mobile device and the cloud. This distribution of the state is coordinated by executing remote services such as user management, file or data storage. In this work, we assume that the state of the application is reflected on a database service, which is the recovery object of MIRES. BaaS database services can vary on the database supported, that can be relational or non-relational [18]. Some services allow the integration of external databases (e.g., Back4App allows the integration of MongoDB[§]), while others provide their own database (e.g., Firebase provides Cloud Firestore[¶]). MIRES does not depend on the database or if it is relational or not. However, the prototype uses a NoSQL database.

[*]https://firebase.google.com/
[†]https://www.back4app.com/
[‡]https://parseplatform.org/
[§]https://www.mongodb.com/
[¶]https://firebase.google.com/docs/firestore

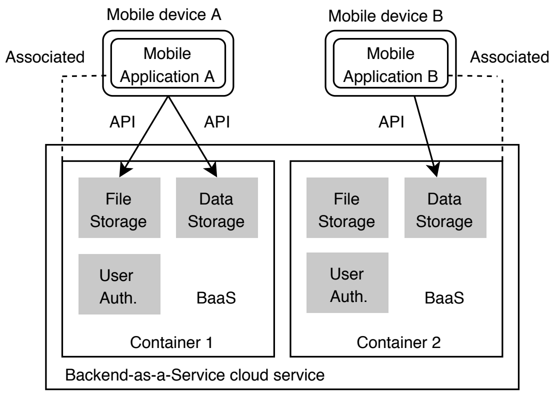

Figure 1: Arquitecture of a BaaS service model.

NoSQL databases are of different types: key-value [33], columnar [35], or document-oriented, as the already mentioned MongoDB and Firestore. In our prototype, we use a document-oriented model, where the database structure is based on *documents* and *collections*, so we present MIRES in terms of that model and often refer to documents. NoSQL databases support CRUD operations: *create*, *read*, *update* and *delete*, but we summarize them in just two: *writes* that modify the content (create, update, delete) and *reads* that do not. These databases support *transactions* that provide the ACID properties[‖]. This allows applications to perform writes – and in some cases reads – atomically in different documents.

In this work we use the term *mobile application* to mean the application running on the mobile device and *mobile application system* to mean the entire system, i.e., both the application and the backend.

2.2 Application and Threat Models

A *mobile application* [22, 36] is a type of software application built to run on a mobile device such as a *smartphone* or a *tablet*, that runs a specific operating system, e.g., Android or iOS. The state of a mobile application is composed of a *local state* existing on the mobile device and a *backend state* existing on the backend database. The focus of this work is to recover the backend state of the mobile application, since the recovery of the local state is already supported in many applications, e.g., the backup recovery process supported by WhatsApp and the implicit recovery done by many applications simply by logging out and logging in again. Also, recovering the backend state is more challenging since it is accessed and modified by many different users, while the local state is only accessed in the mobile device. We assume that the mobile applications always use data based on the backend state that is considered the authoritative copy of the data.

When an user performs an action on the mobile application, e.g., by clicking on a button, a set of operations is made to the backend reflecting the users' action. The operations create, read, update or delete database documents. This set of operations that represent a single action is what we call a *transaction*. In this paper we assume that transactions are performed correctly and atomically, as

[‖]MongoDB transactions https://www.mongodb.com/transactions; Cloud Firestore transactions https://firebase.google.com/docs/firestore/manage-data/transactions our focus is not on recovering inconsistent applications' state due to incomplete transactions (we are not concerned with fixing broken applications, but with recovering from intrusions in correct applications). However, in some cases MIRES is able to recover these inconsistent scenarios.

An intrusion occurs when a malicious action performed by an user explores a vulnerability on the mobile application, originating a *malicious transaction*. A malicious transaction consist of a set of, at least, one malicious operation. However, besides the number of malicious operations, when recovering a transaction, the atomic model must be respected, where all the transactions' operations must be undone, both malicious and non malicious.

We also assume that malicious transactions are the only way the state of the system is compromised. We assume that adversaries cannot corrupt the computational infrastructure of MIRES, the mobile application or the BaaS platform. This assumption does not mean that such problems cannot occur in practice, but only that these are outside of the scope of the solution presented in this paper.

3 MIRES

MIRES is an intrusion recovery service for mobile applications that use BaaS. MIRES is focused on recovering the *integrity* of mobile applications' state by *undoing* the malicious intrusions, i.e., to recover the state of the application such as if the intrusion never took place. In this work, we use the term *system administrator* to mean the person the manages the MIRES service and the term *user* to mean the clients of the mobile application system that MIRES protects.

3.1 Types of Recovery

We consider that the state of the mobile application can only be corrupted by transactions originated by users' actions. We consider two possible scenarios that can be recovered by MIRES service:

(1) **Administrator recovery**: when a transaction is recovered by the system administrator, typically due to the detection of an intrusion;
(2) **User recovery**: when an user makes a mistake and wants to undo an action moments later.

An interesting case happens when the user loses control of his device and the application during an interval of time, e.g., because the device was stolen. That case is handled with Administrator recovery, but also implies a manual process for convincing the administrator that the recovery should be done, e.g., showing a police certificate that the phone was stolen and recovered. We do not present a specific solution for this manual process as it is outside of the technical scope of the solution.

3.2 MIRES Architecture

The MIRES recovery service is formed by a set of different components that run in the frontend (mobile application) and backend (BaaS platform). Figure 2 represents the architecture of MIRES. On the mobile application, the *MIRES package* provides the framework needed to configure the mobile application. On the Application container, alongside with the Application Database, three resources are added: *MIRES users tokens* to retrieve the information that allow MIRES to communication with the application; the *MIRES users recovery* for undoing mistakes done by users (see Section 6); and *MIRES users flags* used for tracing the mobile application normal execution (see Section 4).

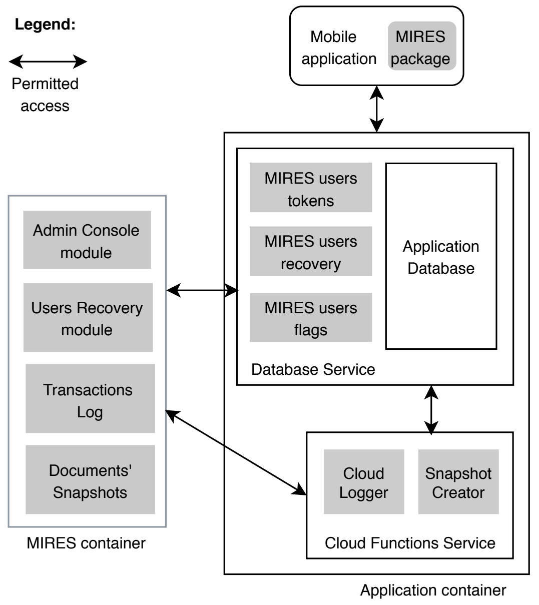

Figure 2: MIRES architecture on a mobile application system (MIRES components in grey).

The functioning of the MIRES service is supported by the *Admin Console module*, that allows the system administrator to interact with the MIRES service and recover malicious intrusion, the *Users Recovery module* responsible for the functioning of the user recovery mechanism (see Section 6) and 2 modules deployed on the application container: the *Cloud Logger* responsible for logging all the requests made to the database and creating the *Transactions Log* and the *Snapshot Creator*, responsible for creating snapshots of the database documents, stored on the *Documents' Snapshots*.

4 MIRES NORMAL EXECUTION

The communication between the mobile application and the BaaS services, as the database service, is achieved by an API provided by the BaaS platform. MIRES provides intrusion recovery without interposing the communication between mobile applications and the backend, i.e., it does not place a proxy between the application and the backend, as many related work systems do [1, 5, 8, 9, 27, 29]. This is important because it allows preserving all the functional and security properties provided by the BaaS service API.

During normal execution, MIRES captures specific data of each transaction performed to the BaaS that, later, can be used on the recovery process. Figure 3 shows the normal function of a mobile application system when using MIRES. In this section we will explain all the steps performed to a transaction during the normal execution of MIRES.

4.1 Mobile application configuration

When an user interacts with the mobile application, a transaction is performed by the mobile application reflecting the users' action (operations *R1* and *R2* in Figure 3). Each write operation is configured to carry extra-data: an *operation ID* representing the operation itself; a

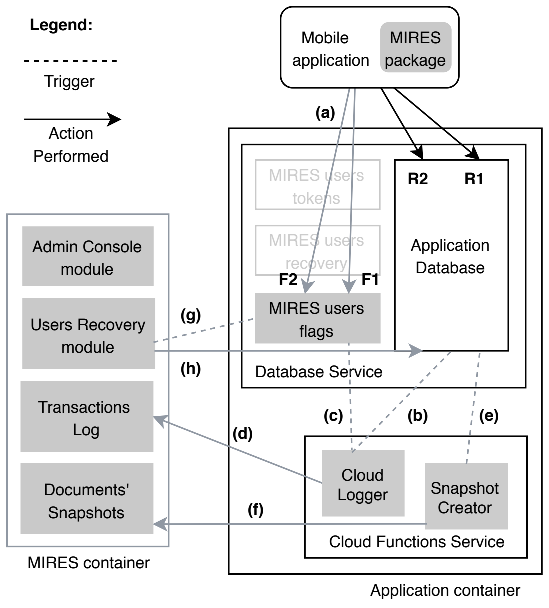

Figure 3: Normal execution flow of MIRES.

locked property used in the recovery process (see Sections 5 and 6.2); and an *ignore* property, used by MIRES to perform requests above database documents, without activating the *Cloud Loggers* functioning. On create/update operations, this extra-data is carried by the operation and stored in each document, while on delete operations the extra-data is carried by the operation's *flag*.

On read operations, the mobile application is configured to forbid reads on *locked* or *blocked* documents.

4.2 Logging Process

MIRES logging process is achieved by using two mechanisms: *flags* and *Cloud Loggers*. Flags carry specific operation information (Section 4.2.1). Cloud Loggers are cloud functions [25] that log the requests made by the mobile applications (Section 4.2.2).

For each operation that alters the state of the database (create, update and delete operations) MIRES gathers the *operation's type*, the *timestamp* associated, the *document changed*, the *data associated with the operation*, a *transaction ID*, that associates all requests of the same transaction (in Figure 3, both *R1* and *R2* are logged with the same transaction ID) and part of the additional information generated by the MIRES package for each write operation (see Section 4.1).

In the rest of this section, we explain how both mechanisms allow MIRES to log each operation made to the database.

4.2.1 Flags. For each write operation made to the database, the mobile application sends a second request (arrow *a*), that we call a *flag* (flags *F1* and *F2* in Figure 3). Each flag is responsible for sending additional information needed to log the operation (see Section 4.2). However sometimes the BaaS API provides function calls where it is not possible to identify the *type* or the *data* performed by the operation on the mobile application, e.g., the *set* operation can be of type create or update, depending if the document exists or not, or backend calls as the Firebase *incrementValue()* made on the mobile application, but where the logic is only executed on the backend.

In these cases, the log information is completed by the *Cloud Logger* (see Section 4.2.2) . On delete operations, this conflict does not occur: delete operations are well defined on the type – delete – and do not generate new data on the database, only delete. Thereby, delete operation flags are always completed and, consequently, can be directly logged by Cloud Loggers.

Besides their transport property, flags are also used to know when the recovery process must be initiated. On rare occasions, Cloud Loggers can take some time to activate and log the operations. However, MIRES can only start the recovery process when it contains the entire log of all operations made to the database. To circumvent this scenario, since each flag represents an operation made to the database, the recovery process can only begin when all flags are processed and the MIRES users flags resource is empty.

Flags follow an ACID model with its associated operation, i.e., each flag can only be sent to MIRES users flags if the associate operation is also performed.

4.2.2 Cloud Loggers. *Cloud Loggers* are MIRES resources that listen to two specific events: *create/update operations* on the Application Database (arrow *b*), and *delete operation flags* on the MIRES users flags (arrow *c*).

When a create/update operation is performed, a Cloud Logger is activated to catch that operation. Then, the Cloud Logger accesses the MIRES users flags in order to get the flag associated with the operation. As previously explained, since the information needed to log the operation cannot always be defined by the mobile application, Cloud Loggers are used to gather the rest of the information needed to log the operation, more precisely, the *type* and *data* handled by the operation.

To gather the data written by the operation, the Cloud Logger compares the document after and before the operation effect. This presents a limitation: an update that writes data already on the document cannot be gathered by the Cloud Logger. For that reason, the operations' data structure is sent on the flag, in order for the Cloud Logger to know each operations' data. Interestingly, this process has an advantage: Cloud Loggers can capture the *direct and indirect effects* of an operation, i.e., a set operation without the merge option replaces the entire document by the new data: a direct effect. Nevertheless, there is data on the document that is discarded: an indirect effect. Cloud Loggers can capture both effects, which allows to reconstruct the documents independently from the type of update made.

On delete operations, the logging process is performed using a different approach. When a new flag is added to MIRES users flags, a Cloud Logger is also activated that accesses the flag to see if it is a delete operation flag and logs directly the operation only in that case. As previously explained, delete operations do not generate new data on the database, which means that flags are the only way to provide information about delete operations. Thereby, delete operation flags always contain all the information needed.

After analyzing the flag and/or the operation performed, the Cloud Logger creates the operation log record (arrow *d*) and deletes the flag on the MIRES users flags.

When the logging process is finished, the log can be accessed by the *Admin Console*, allowing the system administrator to recover the state of the application. Arrows *e*, *f* relate to snapshots and are explained in Section 5.3.2; and arrows *g*, *h* relate to user recovery and are explained in Section 6.1.

4.3 Read operations

Mobile applications change their state by performing write operations on the database. The information sent on each operation may come directly from the user, e.g., user input, or can be based on data already existing on the database. In this last case, mobile applications perform *read* operations in order to retrieve information from the database.

Since this operation type does not change the state of the database, there is no need to log all read operations made by mobile applications. The idea is to log only the read operations that can originate *dependencies* between transactions (see Section 5.2). To achieve this, MIRES package is used to configure the mobile application in order to send the information about the read operation. Thereby, the package offers the possibility to send the information about the read operation through the operation's flag: the *name of the document read*, the *field-values read* (a document can contain both legitimate and illegitimate data, and so it is important to know the data accessed) and the *operation ID* present on the document. MIRES cannot define a timestamp for when the read operation occurred, so the operation ID property allows to know which *version* of the document was accessed; different versions of the document are created by each operation made to that document.

After gathering the data related to the read operation, that information is passed to the Cloud Loggers through the operations' flag of each operation that is influenced by the read operation, in order to be logged alongside with the operation affected.

Besides the read operations made to the database, sometimes dependencies are not strictly defined: for example, function calls that abstract the necessity to perform a read operation like the Firebase *incrementValue()* call, where there is a dependency on the value incremented. In this cases, the dependency exists, so it is necessary to configure these special scenarios in order to increase MIRES recovery efficiency.

5 MIRES ADMINISTRATOR RECOVERY

MIRES follows an approach where *intrusions* and their *effects* are directly removed by *compensating transactions*. The recovery process is divided in two phases: a *locking phase* responsible for identifying the malicious transactions and the affected documents and a *reconstruction phase* responsible for reconstructing the affected documents. This section explains both phases.

5.1 Locking phase

The recovery starts when an intrusion is detected and the administrator activates the MIRES recovery mechanism. Intrusions can be detected manually or using an intrusion detection system or similar mechanism [3, 12, 38], but, as previously mentioned, this mechanism is orthogonal to recovery and out of scope of the paper. The system administrator starts by using the *Admin Console* to select the transactions to undo, and sends a personalized message to each online mobile application, e.g., to explain to the end-users the reasons behind the recovery process.

MIRES messages are received by the application and shown to the user through notifications.

When the recovery is initiated, MIRES locks the entire database, forbidding writes, allowing only reads. Then, the locking phase begins, where MIRES analyzes the log since the moment the first malicious transaction occurred, in order to identify *dependencies* between later transactions and, consequently, identify and *lock* all the affected documents, i.e., documents where both read and write are forbidden.

5.2 Dependencies

During the locking phase, MIRES analyses the log in order to identify *dependencies* between transactions. This analysis is achieved by simulating the spread of corrupted data in memory and comparing the operations made to the database with the corrupted data, in order to identify posterior infected transactions.

5.2.1 Transitive dependencies. When the data written by a transaction is based on data retrieved by a previous read request to the database, there is a *transitive dependency*. For this reason, when an intrusion occurs, read operations can spread the effects of the intrusion by reading corrupted data on the database and, consequently, generating new corrupted data.

Thereby, based on the information gathered about read operations during normal execution (see Section 4.3), when a write operation is influenced by a read operation, MIRES compares the field-values read with the corrupted data in memory, allowing the service to analyze if the read operation was performed on corrupted data, since a document can contain both legitimate and illegitimate data. When a transactions' operation is influenced by a read operation, that in turn has gathered corrupted data, then the data written by the entire transaction is marked as corrupted, which means that the transaction can be seen as a malicious transaction – as writes of corrupted data – that must be recovered. Then, data written by the malicious transaction is added to the corrupted data simulation.

5.2.2 Structural dependencies. Write operations can also create relations between transactions that we call *structural dependencies*. This type of dependency can occur in two possible scenarios: when a write operation is performed on a document that should not exist or when a document is created that should already exist.

In the first scenario, if a malicious transaction creates a new document, then all following operations to that document must be reverted until the document is finally deleted, since all operations are performed above a malicious structure that should not exist. This scenario also involve *sub-collections*: on documents that contains sub-collections, the deletion of the document during the recovery process must result on the recovery of transactions that interact with the sub-collections.

In the second scenario, when a malicious transaction deletes a document, then a create operation that creates the document again must be reverted, since the document should already exist. However, write operations made after the malicious create operation should be considered as legitimate operations, since they are based on the existence of the document and not on the malicious create operation.

5.3 Reconstruction phase

When the locking phase is terminated, MIRES knows the malicious operations and the documents affected that are locked on the database. Then, MIRES unlocks the database, allowing users to interact again with the backend. With the locking phase finished, MIRES starts to reconstruct the affected documents: with the corrupted documents locked, users can normally interact with unaffected documents the database while MIRES reconstructs the corrupted documents.

5.3.1 Operations model. The reconstruction model adopted was based on the *Focused Recovery* algorithm of NoSQL Undo [26]. The Reconstruction follows an operation model where documents are entirely reconstructed through the replay of operations. However, the NoSQL Undo reconstruction model has a drawback: the time to reconstruct the document increases with the number of *versions* of a document. In MIRES this phase is performed concurrently with user interactions with the backend, so the availability of the system is not fully affected; only the infected documents are temporarily unavailable.

5.3.2 Snapshots model. This recovery model is improved using *snapshots* [23], i.e., sets of versions of the documents at certain instants in the past. Snapshots are used by MIRES to mitigate the time to reconstruct the entire document by starting the reconstruction of the document using a document snapshot not corrupted by the intrusion. The creation of snapshots is done during the normal phase based on the operations made per document. This process is supported by the MIRES package, used to configure each write operation – similar to Section 4.1 – by adding a *snapshot* property, that stores the number of operations performed upon the document; and a *timestamp* property, that stores the operation's timestamp. On the backend, the *Snapshot Creator* module listens for database changes (arrow *e* of previous Figure 3) and stores a document snapshot after *N* operations made to the document (arrow *f* of previous Figure 3), e.g., store a version of a document after each 1000 or 10000 operations made. This procedure assures a *non-blocking* model, i.e., the mobile application system is not stopped during this procedure.

6 MIRES USER RECOVERY

Mobile applications are intensive-use applications focused on ensuring a good user experience. However, this intensive-use increases the likelihood of errors and mistakes by the users, e.g., send a wrong message or accidentally delete a post. To help users recover from mistakes, MIRES provides a mechanism that allows users to recover the last action they performed. This mechanism is inspired by the Google Mail undo mechanism that allows users to "unsend" the last mail sent[**].

6.1 Normal Execution

During MIRES normal execution, to activate the users recovery mechanism, each operation suffers an additional configuration, supported by the MIRES package, similar to what is done on Sections 4.1 and 5.3.2. Each write operation is configured to carry extra-data: a *blocked* property, used to generate *blocked* documents; blocked documents are invisible to all the users, i.e., reads are forbidden except for the user that performed the last write on the document; and an *user ID*, representing the user that performed the transaction.

This process works by, when there is a transaction that can be recovered, its operations are saved by the MIRES package. Moreover, write operations *block* the affected document, i.e., by putting the *blocked* property to true.

[**] https://support.google.com/mail/answer/2819488

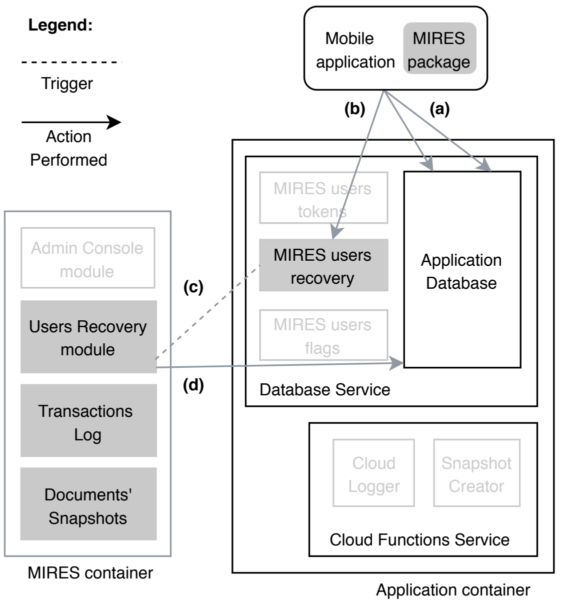

Figure 4: User Recovery mechanism.

After the transaction is performed, a notification with a button and a defined message appears, allowing users to undo their last action. This notification disappears after a time interval T_u (that we set to T_u = 15 seconds in the experiments), or when the mobile application performs another transaction.

On the backend, the Users Recovery module is listening for operation flags (arrow *g* of previous Figure 3). When a flag of a blocked document arrives, the Users Recovery module will unblock the document after a time interval T_u (we set T_u = 30 seconds in the experiments), i.e., changing the *blocked* property to false (arrow *h* of Figure 3).

6.2 Recovery Execution

Figure 4 demonstrates how the user recovery process works. When the user clicks on the undo button, the MIRES package locks the documents directly (arrow *a*) as explained in Section 5.1. After locking the documents, the mobile application sends a recovery request to MIRES users recovery carrying the transaction ID to be recovered and the documents locked (arrow *b*). Then, the User Recovery module gets the recovery request from the user (arrow *c*) and reconstruct the documents affected similar to the reconstruction phase of Section 5.3 (arrow *d*). Both locking the documents and sending the recovery request are made as an atomic model, where the recovery request is only sent if all the documents are locked.

By making the documents invisible for other users, MIRES can recover the transaction without the need to analyse possible dependencies, allowing the possibility to recover multiple transactions from different users at the same time, without requiring the mobile application system to stop. However, this mechanism must only be used in transactions where the affected document can only be changed by a single user, since the objective is to recover users' actions without affecting the application experience of other users, e.g., on a social network application, posts are only modified by the same user.

Table 1: MIRES lines of code (LoCs).

MIRES package	LoCs	MIRES modules	LoCs
Tokens	47	Cloud Logger (flags)	26
Notifications	52	Cloud Logger (collection)	63
Transaction configuration	174	Snapshot Creator	122
Undo Recovery Mechanism	198	Users Recovery module	558
		Admin Console module	913

7 IMPLEMENTATION

MIRES offers a client-side package (Table 1) that allows to configure: the mobile application by managing MIRES notifications, the locking phase of the user recovery mechanism and the configuration needed for each operation by creating a transaction state, on the beginning of each transaction, that is used by the mobile application code to configure the different operations of a transaction. The package was implemented in Java, which is the most frequent option for Android applications.

MIRES was implemented as a two layer service: a first layer composed by the Admin Console module, that supports the recovery mechanism to the system administrator; a second layer composed by the Users Recovery module that supports the user recovery mechanism. With this, MIRES offers *flexible* and *adaptable* configuration: modules are deployed depending on the functionality that we want to use. However, in both layers, the Cloud Loggers are needed to build the log of transactions. Both modules were implemented using Node.js and JavaScript and can be deployed to isolated containers, which provides an important security aspect. The MIRES implementation is small with a limited number of lines of code, as listed in Table 1.

Cloud Loggers were implemented as JavaScript scripts deployed on the mobile application container using the Cloud Functions service. Cloud Loggers listen for specific pre-defined collections, which assures configuration *flexibility* over the database that we want to protect, e.g., it is only required to deploy a Cloud Logger that listens for the collection containing the data that we want to protect and configures the transactions that interact with the same collection. For storing the snapshots, we used the Firestore database service, as used on the Transactions Log.

The Snapshot Creator followed an implementation process similar to Cloud Loggers: it was developed as a JavaScript script, deployed on the mobile application container using the Cloud Functions service. For storing the snapshots, we used the Firestore database service, as used on the Transactions Log.

The BaaS platform used was *Firebase*. We used the Firestore database service to store the log of transactions. With this service, and the Cloud Loggers, we can assure *automatic scaling* on the creation of the log. Also, since Firestore is a NoSQL database, it offers a flexible storing process with a set of personalized read queries for the recovery process.

The MIRES user flags, user tokens and user recovery were implemented using database collections. By implementing these three collections on the mobile application container, it is possible to reuse the security rules and settings that allow only the authenticated users to interact with the three collections. It is also possible to define specific security rules that allow to isolate the three collections from the rest of the application database. By following this implementation, we used the atomic mechanism provided by the database service, used to implement the flag and the locking phase of the users recovery mechanism.

8 EXPERIMENTAL EVALUATION

Our experimental evaluation aims to answer to the following questions: (1) What is the mobile application performance overhead and the Cloud Loggers performance when logging the operations? (2) How much storage space does MIRES require to store the log and how much space does it take on the database application? (3) How much time does the Admin Console module take to recover the mobile application in different scenarios? (4) What is the performance of the Users Recovery module on unblocking documents and recovering transactions?

To evaluate the MIRES service, we used four open-source Android applications: a *social network application*, Hify,[††] where users can post, comment and like; a *messaging application*[‡‡] for 1 to 1 conversations; a *shopping list application*, ShoppingListApp,[§§] to create lists, by adding, changing and removing products; and an *contact tracking application*, CovSense[¶¶], used to track contacts between their users and manage the COVID-19 spread. Each mobile application was executed on a mobile device with 3GB of memory and an Octa-Core Kirin 710 processor connected to a 47.78 Mb/s download speed and 9.58 Mb/s upload speed network. Both application and MIRES containers were deployed on Google Cloud in the same region to mitigate possible network delays. Each MIRES module was deployed on Google Compute Engine, on a N1 generation machine, with 1vCPU, 3.65 GB of memory and running Debian Linux 10 OS. All results shown next are averages of the results obtained with the 4 applications, except when noticed.

8.1 Logging Evaluation

8.1.1 Mobile Application Performance. MIRES requires the configuration of the write and read operations made to the database, resulting on an performance overhead on each operation. To test the imposed overhead, we simulated the user's actions by performing a set of 1K CRUD operations. Each block of operations was repeated 5 times and followed a different workflow distribution on each application: 80/20 read/write distribution for the social network application, where users tend to actively read others users posts, comments and likes, 50/50 read/write for the messaging application based on read/reply conversations and a 20/80 read/write for the shopping list application, since lists tend to be intensively updated, by adding, changing and removing items and 0/100 read/write for the CovSense application. since the application does not perform reads in its logic, only writes.

Figure 5 shows the results of the experiment. MIRES imposes an overhead of 23% on the Shopping List App, 18% on the Messaging App, 16% on the CovSense App and 15% on the Social Network App. This difference happens because operations are configured differently on the mobile application: write operations are configured using a *Firebase transaction* with an extra create operation (flag), whereas read operations are configured by adding a *filter* to blocked and locked documents, which leads to a lower cost on read operations.

[††]https://github.com/lvamsavarthan/hify
[‡‡]https://github.com/ResoCoder/firebase-firestore-chat-app
[§§]https://github.com/alexmamo/Firestore-ShoppingListApp
[¶¶]https://github.com/saivittalb/covsense version of 27/09/2020

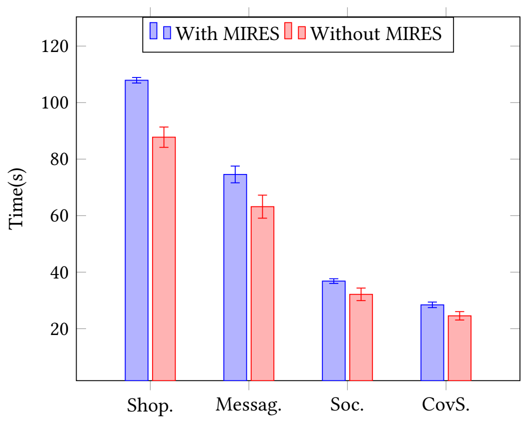

Figure 5: Time to perform 1000 operations on each application, with and without MIRES.

Although the overhead is noteworthy, we consider it acceptable, given the benefit provided by the service.

8.1.2 Cost of logging operations. Cloud Logger scripts were deployed on the mobile application container to listen for database changes and flags. It was deployed on the same region of the application container, to minimize the activation time and assure all the necessary triggers. The script was deployed on a Node.js 10 execution environment with 1 GB of memory dedicated. From the previous workflow made to the database, we observed that each Cloud Logger took an average of 0.47±(0.06) seconds and 0.09±(0.01) GB of memory to execute.

8.2 Space Overhead

8.2.1 Database overhead. The additional data is saved on each database document, which imposes a storage overhead. The size of each document is increased by a minimum of 69 bytes and a maximum of 173 bytes – 69 bytes for the Administrator Recovery, 57 bytes for the Users Recovery mechanism and 47 bytes for the snapshots creation flow (each document has a maximum capacity of 1 MB, which means and occupation between 0.006% and 0.018% of the maximum size allowed). The data is stored on a minimum of 3 field-values and a maximum of 7 field-values – 3 for the Administrator Recovery, 2 for the Users Recovery mechanism and 2 for the snapshots creation flow (each document can only contain 100 fields, which means a minimum occupation of 3% and a maximum occupation 7%).

MIRES also creates three collections on the application database: user flags, user recovery and user tokens. Both user flags and user recovery are implemented as support structures where data is not persistent over time. Only the user tokens collection saves the users' tokens needed on the recovery process for communication purposes. Each user token is saved on a different document occupying 255 bytes each.

Concluding, the maximum additional data size imposed by MIRES on the application database is given by the expression (in bytes):

$$S_{db} = 173 \times documents + 255 \times users$$

where *documents* is the number of database documents and *users* the number of users. For example, an application with 1M users and 1M documents has 0.41 GB of additional data.

Table 2: Log size.

Mobile Application	Log Size (GB)
Social Network App	0.11
Messaging App	0.25
Shopping List App	0.41
CovSense App	0.42

8.2.2 Log records. MIRES stores specific data in each operation made to the database (see Section 4). Each log record size is given by the following expression (in bytes):

$$S_{log} = 215 + doc + (53 + data)$$

where *doc* represents the path to the document affected and *data* the data sent on the operation; the 53 bytes are only added if the operation wrote any data.

8.2.3 Dependencies. When a write operation is influenced by a read operation, there is an additional information logged related with the read operation (see Section 4.3). The dependencies size of an operation is given by the following expression (in bytes and where *D* defines the number of documents read and *F* the number of field-values read):

$$S_{dep} = 91 + \sum_{d=1}^{D}(doc + 1 + \sum_{f=1}^{F}(field + 1))$$

where the *doc* property represents the path to the document read and the *field* property represents each field-value read.

Table 2 shows the log size needed to store 1M operations following the exact workflow performed on Section 8.1 on each application.

8.3 Admin Recovery Performance

The *Time to Recover* (TTR) is defined as the total time since the system administrator starts the recovery process until the moment that all the effects of the intrusion are removed. In MIRES, the TTR is the sum of the *Locking phase* and *Reconstruction phase* times.

To test the recovery performance, we defined three real scenarios. In *scenario 1* we have created an user in each application and performed a different number of actions resulting in 1 to 1000 operations to recover. In *scenario 2* we used the same user and the applications' actions to perform 1 to 10K operations upon the same document, in order to create 1 to 10K different versions of the document. In *scenario 3* we tested a particular case where the effects of malicious intrusions are not persisted in the database, however malicious information is sent to users. Each recovery scenario was executed 5 times. In the rest of this section, we analyse the MIRES recovery performance on both scenarios.

8.3.1 Scenario 1. Figure 6 shows the results of undoing the actions of the user. Both Locking and Reconstruction phases increase linearly with the increase of the log size, the dependencies and the documents to recover (in this test, there was an average of 45 documents per each 100 operations on each application). Recovering a single operation takes less than 1 second, while recovering 1K operations takes 55 seconds maximum. However, in this latter case, the mobile application system is unavailable for only 15 seconds.

The Locking phase is composed by the load and analysis of the log to identify the malicious transactions and the corrupted documents.

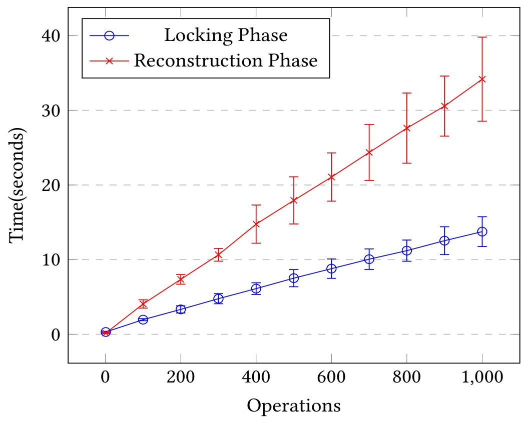

Figure 6: Time to undo a different number of operations.

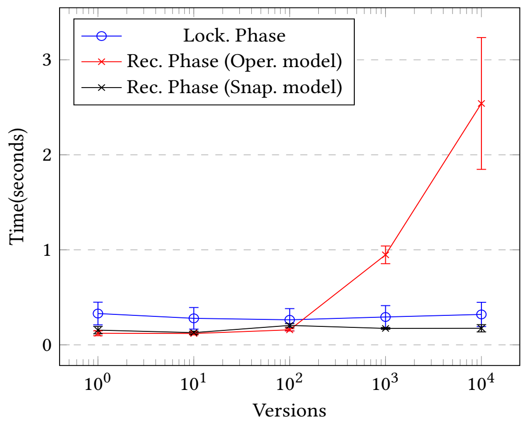

Figure 7: Time to reconstruct a document with different versions.

The Reconstruction phase is composed by the reconstruction of the locked documents. We can see that this phase takes more time than the Locking phase. This happens because, on the Locking phase, MIRES loads and analyses the log on a single process, while on the Reconstruction phase, for each document locked, MIRES needs to load the operations that affect the document to reconstruct – for legitimate operations – or update the log – for malicious operations.

8.3.2 Scenario 2. Figure 7 shows the results of reconstructing a document with different versions. By following an operation model, the reconstruction of a document takes longer with the increase of the number of versions, since MIRES needs to replay all the operations needed to reconstruct the document. However, by using snapshots of 1K versions, we could reconstruct a document with 10K versions in less then 0.5 seconds, instead of the almost 3 seconds needed when using the operations model.

We focused our depth analysis only on the last workflow tested, i.e., the 10k versions. We observed that each Snapshot Creator took an average of 0.10±(0.01) seconds and 0.08±(0.01) GB to execute. All tested applications – Social Network, Shopping Lists and CovSense applications – stored 10 snapshots of the document – each with 1000 versions – imposing an additional storage of 0.01 MB on all applications.

The Locking phase remained practically the same, since it was always the same unique transaction to analyse.

8.3.3 Scenario 3. Scenario 3 was focused on a different type of recovery, where the effects are not persisted in the database, however, some type of malicious information is generated and shared with the users. To test this particular scenario, we used only the CovSense application to simulate a malicious health status change that generated a malicious flow of notifications sent to some users. MIRES was constructed to allow to send recovery notifications to each user. We tested the notifications mechanism by sending 1, 10 and 100 notifications. We performed each test 5 times, concluding that sending a 1 notification costs 0.06±(0.03) seconds, sending 10 notifications costs 0.81±(0.04) seconds and 100 notifications costs 5.80±(0.58) seconds.

8.4 User Recovery Performance

8.4.1 Normal Execution. During normal execution, each time that an invisible document appears, the Users Recovery Module unblocks the document after 30 seconds. To test the unblocking time, we executed three different flows: we have performed 1, 10 and 100 operations concurrently, each generated a blocked document. This test flow was conducted in each application, except the CovSense application. Each flow was repeated 5 times. With this experiment, we concluded that, after the 30 seconds, and with the increase of documents to unblock, the average time to unblock each document is 0.08±(0.04) seconds.

8.4.2 Recovery Execution. The user recovery process is initiated with the direct lock of the documents by the mobile application. We have tested the locking phase by locking 1 and 10 documents. This test flow was conducted in each application, except the CovSense application. Each test was repeated 5 times. We observed that locking a single document costs 0.27±(0.01) seconds, while locking 10 documents costs 1.02±(0.01) seconds. However, since this phase is performed by the mobile application, the time to lock the documents can be volatile, depending on the network speed and on the mobile device.

The Users Reconstruction phase follows the same model as the Administrator Reconstruction phase (see Section 8.3).

9 RELATED WORK

Intrusion recovery has been much investigated considering different systems: databases [2, 10, 26], virtual machines [20, 30, 37], file systems [14, 16, 17, 19, 32, 34, 39], web applications [1, 8, 9] and cloud-computing service models [27–29].

Undo for Operators [5] is both the first presentation of the broad intrusion recovery approach we follow and a tool that allows operators to recover from their own mistakes, from unanticipated software problems and from intentional or accidental data corruption. The model for Operator Undo is based on three concepts referred as the *three R's*: *Rewind*, where all the state of the system is physically rolled back in time to a point before any damage occurred; *Repair*, where the operator alters the rolled-back system to prevent the problem from reoccurring; and *Replay*, where the repaired system is rolled forward to the present by replaying portions of the previously-rewound legitimate requests.

There are some works on web and Platform-as-a-Service (PaaS) applications but we refer only three for brevity. Warp [8] assists users and administrators of web applications to recover from intrusions while preserving legitimate user changes. The Warp recovery approach is based on rolling back a part of the database to a point in time prior to the intrusion and then apply compensating operations to correct the state of the database. Shuttle [29] is a similar intrusion recovery service for PaaS applications, that aims to help administrators to recover their applications from software flaws and malicious or accidentally corrupted user requests.Rectify [27] is a black-box intrusion recovery service for Platform-as-a-Service (PaaS) applications. Rectify considers that the application is a black box, so it observes HTTP requests and DB statements and finds the relations between them without looking into the application code or requiring modifications to that code. Relations between HTTP requests and DB statements are derived using *supervised machine learning*.MIRES is based on some of the ideas of these works. However it is the first that considers mobile applications and BaaS. Moreover, it introduces the idea of dividing the process in two phases, which improves the availability of the system. MIRES also provides a new short-term recovery mechanism to mobile applications that allow users to recover their last action that is an enhancement welcome in most applications where end-users can commit mistakes.

10 CONCLUSION

This paper presents MIRES, an intrusion recovery service for mobile application systems that use BaaS. MIRES performs a two phase recovery process, that aims to recover the state of the mobile application system and minimize the unavailability of the system during the procedure. Besides the intrusion recovery functionality, MIRES also presents an user recovery mechanism allowing application users to undo their last action.

REFERENCES

[1] İ. E. Akkuş and A. Goel. Data recovery for web applications. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 81–90, 2010.

[2] P. Ammann, S. Jajodia, and P. Liu. Recovery from malicious transactions. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1167–1185, 2002.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 259–269, 2014.

[4] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan-Mar 2004.

[5] A. B. Brown and D. A. Patterson. Undo for operators: Building an undoable e-mail store. In *USENIX Annual Technical Conference*, pages 1–14, 2003.

[6] S. Bugiel, S. Heuser, and A.-R. Sadeghi. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *Proceedings of the 22nd USENIX Security Symposium*, pages 131–146, 2013.

[7] B. Carter. Grow your own backend-as-a-service (baas) platform. In *GOCICT 2015 Conference College of Information & Computer Technology*, Nov. 2016.

[8] R. Chandra, T. Kim, M. Shah, N. Narula, and N. Zeldovich. Intrusion recovery for database-backed web applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 101–114, 2011.

[9] R. Chandra, T. Kim, and N. Zeldovich. Asynchronous intrusion recovery for interconnected web services. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pages 213–227, 2013.

[10] T.-C. Chiueh and D. Pilania. Design, implementation, and evaluation of a repairable database management system. In *21st International Conference on Data Engineering (ICDE'05)*, pages 1024–1035, 2005.

[11] J. A. L. Ferreira and A. R. da Silva. Mobile cloud computing. *Open Journal of Mobile Computing and Cloud Computing*, 1(2):59–77, 2014.

[12] K. Gai, M. Qiu, L. Tao, and Y. Zhu. Intrusion detection techniques for mobile cloud computing in heterogeneous 5G. *Security and Communication Networks*, 9(16):3049–3058, 2016.

[13] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Press, USA, 2 edition, 2008.

[14] A. Goel, K. Po, K. Farhadi, Z. Li, and E. De Lara. The taser intrusion recovery system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 163–176, 2005.

[15] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting Android to protect data from imperious applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 639–652, 2011.

[16] F. Hsu, H. Chen, T. Ristenpart, J. Li, and Z. Su. Back to the future: A framework for automatic malware removal and system repair. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 257–268, 2006.

[17] S. Jain, F. Shafique, V. Djeric, and A. Goel. Application-level isolation and recovery with solitude. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 95–107, 2008.

[18] N. Jatana, S. Puri, M. Ahuja, I. Kathuria, and D. Gosain. A survey and comparison of relational and non-relational database. *International Journal of Engineering Research & Technology*, 1(6):1–5, 2012.

[19] T. Kim, X. Wang, N. Zeldovich, M. F. Kaashoek, et al. Intrusion recovery using selective re-execution. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, pages 89–104, 2010.

[20] S. T. King and P. M. Chen. Backtracking intrusions. *ACM SIGOPS Operating Systems Review*, 37(5):223–236, 2003.

[21] K. Lane. Overview of the backend-as-a-service (baas) space. *API Evangelist*, 2015.

[22] V. Lee, H. Schneider, and R. Schell. *Mobile Applications: Architecture, Design, and Development*. Prentice Hall PTR, USA, 2004.

[23] J.-L. Lin and M. H. Dunham. A survey of distributed database checkpointing. *Distributed and Parallel Databases*, 5(3):289–319, 1997.

[24] P. Liu, P. Ammann, and S. Jajodia. Rewriting histories: Recovering from malicious transactions. In *Security of Data and Transaction Processing*, pages 7–40. Springer, 2000.

[25] M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela. Serverless execution of scientific workflows: Experiments with hyperflow, aws lambda and google cloud functions. *Future Generation Computer Systems*, 2017.

[26] D. Matos and M. Correia. NoSQL undo: Recovering NoSQL databases by undoing operations. In *2016 IEEE 15th International Symposium on Network Computing and Applications*, pages 191–198, 2016.

[27] D. R. Matos, M. L. Pardal, and M. Correia. Rectify: Black-box intrusion recovery in paas clouds. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, page 209–221, 2017.

[28] D. R. Matos, M. L. Pardal, and M. Correia. RockFS: Cloud-backed file system resilience to client-side. In *Proceedings of the 2018 ACM/IFIP/USENIX International Middleware Conference*, 2018.

[29] D. Nascimento and M. Correia. Shuttle: Intrusion recovery for paas. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 653–663, 2015.

[30] D. Oliveira, J. R. Crandall, G. Wassermann, S. Ye, S. F. Wu, Z. Su, and F. T. Chong. Bezoar: Automated virtual machine-based full-system recovery from control-flow hijacking attacks. In *Proceedings of the IEEE Network Operations and Management Symposium*, pages 121–128, 2008.

[31] Positive Technologies. Vulnerabilities and threats in mobile applications, 2019. 6 2019.

[32] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of ACM SIGOPS Symposium on Operating Systems Principles*, pages 110–123, 1999.

[33] S. Sivasubramanian. Amazon dynamodb: a seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 729–730, 2012.

[34] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: protecting data in compromised system. In *Proceedings of the 4th USENIX Symposium on Operating System Design & Implementation*. USENIX Association, 2000.

[35] M. N. Vora. Hadoop-hbase for large-scale data. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, volume 1, pages 601–605, 2011.

[36] S. Xanthopoulos and S. Xinogalos. A comparative analysis of cross-platform development approaches for mobile applications. In *Proceedings of the 6th Balkan Conference in Informatics*, pages 213–220, 2013.

[37] X. Xiong, X. Jia, and P. Liu. Shelf: Preserving business continuity and availability in an intrusion recovery system. In *Proceedings of the Annual Computer Security Applications Conference*, pages 484–493, 2009.

[38] S. D. Yalew, G. Q. Maguire Jr., S. Haridi, and M. Correia. DroidPosture: A trusted posture assessment service for mobile devices. In *Proceedings of the 13th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications*, Oct. 2017.

[39] N. Zhu and T.-c. Chiueh. Design, implementation, and evaluation of repairable file service. In *Proceedings of the International Conference on Dependable Systems and Networks*, page 217, 2003.