TÉCNICO LISBOA

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

Diogo Lopes Vaz

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Science Engineering

Supervisor(s): Prof. Miguel Filipe Leitão Pardal
Prof. Miguel Nuno Dias Alves Pupo Correia

Examination Committee

Chairperson: Prof. Francisco António Chaves Saraiva de Melo
Supervisor: Prof. Miguel Filipe Leitão Pardal
Member of the Committee: Prof. João Tiago Medeiros Paulo

November 2020

Acknowledgments

First, I would like to thank my supervisors Prof. Miguel Pardal and Prof. Miguel Correia for the opportunity of doing this dissertation. Thank you for the ideas and suggestions, the knowledge and experience shared during this journey that helped me incredibly during to realize this work. I also want to thank Dr. David Matos as well for directly sharing his research work on the field with me. This literature is the final result of a long path, full of difficulties, persistence and hard work. A path of constant learning and improve, motivated by the feeling of increasing the stock of knowledge in the field of computer science. I would like to thank my family, father, mother and brother for the amazing support, motivation and patience throughout this journey. They were a vital part in this process. I would also like to thank to all my friends for the support during this phase of study, failures, successes, and learning. I share with you all my success. Finally, I would like to specially thank my girlfriend for being my pillar, my support and inspiration on every single day of this journey, through the good and bad moments. Without all of you, this would certainly not be possible. Thank you all.

Resumo

Muitas aplicações móveis populares baseiam-se no modelo de computação na nuvem chamado *Backend-as-a-Service* (BaaS) – Backend-como-um-serviço – para simplificarem o desenvolvimento e gestão de serviços como o armazenamento de dados, a autenticação de utilizadores e notificações. Porém, vulnerabilidades e outros problemas podem originar operações maliciosas na aplicação móvel que consequentemente geram pedidos maliciosos feitos aos servidores, corrompendo o estado da aplicação na nuvem. Para lidar com estes ataques depois de acontecerem e terem successo, é necessário remover os efeitos imediatos criados pelos pedidos maliciosos e efeitos subsequentes derivados de pedidos posteriores. Neste trabalho, apresentamos o MIRES, um serviço de recuperação de intrusões para aplicações móveis baseadas no modelo BaaS. O MIRES usa um processo de recuperação em duas fases que restaura a integridade da aplicação móvel e minimiza a sua indisponibilidade. Para além da funcionalidade principal de recuperação de intrusões, o MIRES também oferece um mecanismo no lado cliente que permite aos utilizadores das aplicações móveis reverterem as suas ações. O MIRES foi implementado em Android e com a plataforma Firebase. Foram feitas experiências com 4 aplicações móveis que mostraram resultados de 1000 operações revertidas em menos de 1 minuto e com as aplicações móveis indisponíveis por menos de 15 segundos.

Palavras-chave: Recuperação de intrusões, Computação móvel, Backend-como-um-serviço, Computação na Nuvem

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 that consequently generate 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 work, 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. Besides the main intrusion recovery feature, MIRES also provides a client-side mechanism that allows the mobile application users to revert their own actions. 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 Computing

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Mobile applications are software programs that run on mobile devices, typically *smartphones* or *tablets* and play an important role in our lives, as they provide daily-use services like message chats, social networks or online banking, just to name a few. Most mobile applications rely on remote services and resources provided by servers, often designated *clouds*, to support their functioning. Recently several frameworks/platforms have appeared to support the development and execution of mobile applications. These frameworks allow to integrate 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) [Car16, FdS14, Lan15], 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 languages[1] and the Lyft car sharing platform[2] are both based on the Firebase BaaS platform[3].

Mobile applications often contain *vulnerabilities*, e.g., due to improper user input validation, or other errors made by developers in designing and/or writing code, leading to an increase of the *attack surface*. These weaknesses can be explored by threat actors, such as malicious users, with the intent to corrupt the state of the application stored in the backend, leading to *intrusions*. For example, a 2019 study revealed that 60% of the mobile applications vulnerabilities were on the client side, where two thirds were medium/high risk [Pos19].

Nowadays, mobile applications are critical assets to companies due to their inherent benefits like portability, usability, and connectivity, that are convincing companies to use mobile applications as client interfaces for their services [LSS04]. Therefore, it is crucial to create *intrusion*

[1] `https://en.duolingo.com/`
[2] `https://www.lyft.com/`
[3] `https://firebase.google.com/`

tolerant mobile applications systems, in order to protect and preserve the *integrity* of the system and, consequently, its correct functioning. Nevertheless, it is important to assure intrusion tolerance without affecting the system performance and availability and therefore conserve a good user experience.

1.1 Topic Overview

This work is about *intrusion recovery*, i.e., about reverting the effects of the intrusion on the state of the application, and to do it with low impact on *availability*. A simple solution on the intrusion recovery field 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* [GMUW08]. 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 [BP03]. Our approach involves generating *compensating transactions* [LAJ00] 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 [AG10, CKS+11], databases [CP05, MC16], operating systems [KWZ+10], email services [BP03], and cloud computing [MPC17, NC15]. 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.

When exploring mobile applications using a cloud-backend as the Backend-as-a-Service, new challenges arise as the use of APIs on the communication between the mobile application and the backend, making impossible to interpose the communication using a proxy as some previous works on intrusion recovery do [BP03, AG10, CKS+11, CKZ13, NC15, MPC17]; the need to maintain the availability of the system, a critical point to provide a good user experience; and provide some kind of an user's recovery mechanism allowing users to revert their actions, since mobile applications are intensive-use application where users' errors are more likely to happen.

1.2 Objectives

The main objectives of this literature are:

1. Develop an intrusion recovery service for an emerging cloud service model, focused on mobile applications;

2. Provide an offline recovery model, that aims to restore the integrity of the systems' state with a focus on maintaining the availability of the mobile application system;

3. A recovery mechanism that allows users to undo their own actions;

1.3 Contributions

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 [ALRL04], 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 [HHJ+11, BHS13]. Our work also does *not* focus on intrusion detection, that is orthogonal to intrusion recovery; other mechanisms could be used for this purpose [ARF+14, GQTZ16, YMHC17].

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

1.4 Thesis Outline

This dissertation is organized in the following way: Chapter 2 introduces the intrusion recovery field, the mobile applications with Android and the cloud computing paradigm, introducing the main cloud computing models and the Backend-as-a-Service cloud computing model with

more detail; Chapter 3 explains the MIRES approach, by presenting its architecture and then describing the execution of MIRES during the normal phase and both administrator and user recovery phases supported by MIRES; Chapter 4 demonstrates the results of the experiments performed on the MIRES service using four different applications; finally Chapter 5 concludes this literature and describes possible improvement points for future work.

Chapter 2

Background & Related Work

In this chapter we present background on the intrusion recovery field, including a set of previous intrusion recovery works, selected based on its importance to this work; the mobile applications, with an in-depth analysis on the Android Operating System, the Android application and their main components; and finally the cloud computing paradigm, introducing the main cloud computing models and the Backend-as-a-Service cloud computing model with more detail.

2.1 Intrusion Recovery

The term *intrusion* is normally used to designate *unauthorized* activities that affect the *integrity*, *confidentiality* and/or *availability* of a system [ALRL04]. However, in this work we use the term in a broader sense to include also *authorized* but erroneous activities from which someone later wants to undo. 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 literature [HHJ+11, BHS13].

The process of dealing with an intrusion is divided into three main phases: *intrusion detection*, *vulnerability fix* and, finally, *intrusion recovery*.

The first phase, *intrusion detection*, consists of monitoring the events in a system or network and analyze them for signs of suspicious activities, being an important procedure to deal with unpredictable attacks. Previous works resorts to *Intrusion Detection Systems* (IDS) [ARF+14, GQTZ16, YMHC17, GTDVMFV09] to help analyzing and detecting suspicious events. However, some of these systems need human configuration to deal with precision issues, like false positives, in order to initiate the recovery process.

The second phase is responsible for *fixing the vulnerability*. This phase consists on *classifying*

and *mitigating* the vulnerability that originated the intrusion, by configuration adjustments or applying security patches – uploading code developed to resolve the specific vulnerability in the software [ZWW+10, TW]. The goal is to prevent similar intrusions from happening again.

The last phase – and the scope of this work – is the *intrusion recovery* phase. This process aims to remove the effects related to the intrusion and return the application to a state where those effects are mitigated, restoring the integrity of the system. To handle intrusion recovery, there are two common approaches that can be used: *rollback* or *compensation*.

Rollback is based on rolling back the state of a system – all activity, desirable and undesirable – to a state believed to be free of damage (e.g., Row Versioning and Snapshots [GMUW08]). Then, the system re-executes all the requests not related to an intrusion, a process called *roll-foward*, in order to bring the system to the present state, annulling the effects of the intrusion.

Compensation is based on undoing malicious intrusions and their direct and indirect effects without necessarily restoring the data state, to appear as if the malicious intrusions had never been executed (e.g., Compensation transactions [KLS90]). Then, if needed, the system re-executes the legitimate requests reverted on the compensation.

Next we will present a set of previous intrusion recovery literatures, selected based on their importance for this work, more precisely, by analysing characteristics as the recovery approach followed, the target system to recover, the type of recovery applied, the existence of an users' recovery mechanism and the use of a proxy in the architecture.

Intrusion recovery has been much investigated considering different systems: databases [CP05, MC16, AJL02], virtual machines [KC03, OCW+08, XJL09], file systems [GPF+05, SFH+99, SGS+00, ZC03, HCR+06, JSDG08, KWZ+10], web applications [AG10, CKS+11, CKZ13] and cloud-computing service models [MPC17, NC15, MPC18].

Undo for Operators [BP03] is a tool that allows operators to recover from their own mistakes, from unanticipated software problems and from intentional or accidental data corruption. As an example, the paper extends an email server with recovery mechanisms. 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. We follow this broad approach in this work.

During the normal execution, a proxy is responsible for intercepting requests coming from users and packaging them into *verbs* —- a verb is an encapsulation of an user interaction with the system, i.e., a record of an event that causes state in the service to be changed or externalized

—- creating a record of user intent. Then, verbs are sent to the undo manager for processing. The undo manager uses the verb interfaces to generate a causally consistent ordering of the verbs it receives, sends the verbs back to the proxy for execution on the service system, and records the sequence of executed verbs in an on disk log. This verb log forms the recorded timeline of the system.

During an undo cycle, all system hard state is physically rewound through the load of system-wide snapshot in order to remove any corrupted data — *rewind* — and the operator patch the software flaws of the application — *repair*. Finally, all legitimate requests started after the intrusion are re-sent to the proxy to rebuild the application state — *replay*. During this process, the system only authorizes synchronous read-only requests to be executed, since the objective is to retain the ability for users to at least inspect their mailbox state even if the state is temporarily inconsistent and immutable. All asynchronous requests are delayed —- being asynchronous, they can tolerate the delay —- and synchronous requests that cannot be executed read-only are forbidden. Also, as a consequence of the recovery, external inconsistencies can occur e.g, mails or folders can change, appear or disappear without warning. To compensate this, Undo Operator, for the most part, insert explanatory messages into the mailbox of the user, apologizing for the inconsistencies, explaining what they are and why they were necessary.

Undo for Operators is the first presentation of the broad intrusion recovery approach that we follow. MIRES collects three ideas from this literature: the logic of logging requests and higher-level operations during the normal execution, the read-only permission during the recovery phase and the explanatory messages to the users when a recovery process is executed.

Warp [CKS+11] is a system that assists users and administrators of web applications to recover from intrusions while preserving legitimate user changes. Warp based his recovery approach on Retro [KWZK10], a tool that helps to repair from intrusions on operating systems.

The recovery process of WARP is initiated when the administrator learns that a vulnerability was discovered by the developers. The first step is to determine which runs of the application code may have been affected by a bug. Then, WARP applies the security patch and considers re-executing all potentially affected runs of the application. In order to re-execute the application, WARP records sufficient information of all the inputs, during the normal execution of the application (e.g., HTTP requests). When WARP re-executes the application code with the request of the intrusion, the newly patched application will behave differently and then issue an SQL query to store the results in the database. Since the new SQL query must logically replace the original query, WARP rolls back the database to its state before the attack took place. After the database has been rolled back, and the new query has executed, WARP determines what

other parts of the system were affected by this changed query. To do this, during the original execution WARP records all SQL queries, along with their results. During the repair, WARP re-executes any queries it determines may have been affected by the changed query, as also if a re-executed query produces results different from the original execution, WARP re-executes the corresponding application run as well.

WARP provides a browser extension that records all events for each open page in the browser (such as HTTP requests and user input) and uploads this information to the server. If WARP determines that the browser may have been affected by an attack, it starts a clone of the browser on the server, and re-executes the original input on the repaired page, without involving the user. If any conflict arises, WARP signals the conflict and asks the user (or administrator) to resolve it.

WARP implements online recovery, introducing the notion of *repair generations*, i.e., when repair is initiated, it is created a fork of the current database contents, named *next generation*. All database operations during repair are applied to the next generation. If during repair, users make changes to parts of the current generation that are being repaired, WARP will re-apply the changes of the users to the next generation through re-execution. Changes to parts of the database not under repair are copied verbatim into the next generation. Once repair is near completion, any final requests are re-applied to the next generation and the current generation is set to the next generation.

MIRES presents two similarities with WARP: both require a client-side extension and offer an user recovery mechanism. Also, the objective of the MIRES two-phase recovery process is based on the WARP online recovery idea of improving the availability of the system.

Shuttle [NC15] is a similar intrusion recovery service for Platform-as-a-Service (PaaS) systems, that aims to help administrators to recover their applications from software flaws and malicious or accidentally corrupted user requests. It can be provided by Cloud Service providers (CSPs) as a service integrated in a PaaS system and also works with NoSQL databases. Shuttle assumes a client-server model in which clients communicate with the servers on cloud using HTTP/HTTPS.

Applications supported by Shuttle can operate in one of two phases: *normal execution* and *recovery execution*. During normal execution, Shuttle records the data required to recover the state of the application: it does periodic database snapshots and logs user requests and database accesses.

When an intrusion is identified, the recovery phase is initiated. During this phase, Shuttle removes intrusion effects creating a new branch of the system in which it loads a snapshot,

which contains the application state before the intrusion occurs. It builds a consistent state by re-executing, in the new branch, the legitimate requests logged during normal execution, while new incoming requests are executed in the previous branch. Re-execution can perform either *selective* or *full replay*. Full replay consists of replaying every request done after the snapshot, which can take considerable time, being this approach adequate for intrusions detected reasonably early after they happen, e.g., a few days. Selective replay re-executes only part of the requests, being faster than full-replay and require that tenants provide a set of malicious requests, used to deduce the set of *tainted requests*. A request is said to be tainted if it is one of the attacker's requests or if it reads objects written by tainted request. After the tainted requests are deduced, selective replay gets the requests needed to obtain the values read by them. Next, Shuttle recovery process determines the replay order sorted in start-end order and finally replays the requests. Shuttle provides an API for the application programmer to define how inconsistencies seen by users are dealt with:

- **preRecover() -** invoked before the beginning of the recovery process, allowing to perform a set of actions before the beginning of the recovery process, such as notifying the operations team or taking a new snapshot;
- **handleInconstency** (request, previous response, new response, previous keys, new keys, action) - invoked when there is an inconsistency, taking as input the request that caused the inconsistency as well as the response and keys accessed during the normal execution and during the recovery process. It also takes as argument the action to take, considering three possible actions: ignore the inconsistency, notify the user of the inconsistency and execute another request;
- **posRecover() -** invoked after the end of the recovery process, allowing the tenant to access not only to the statistics of the recovery process but also to an interface to compare the database values before and after the recovery process and the application responses, before exposing the data to the users.

MIRES and Shuttle present a similarity: both save read accesses to the database, in order to identify dependencies between transactions. However, MIRES follows a different recovery approach, by implementing a two-phase recovery algorithm to improve the system availability, instead of interposing the communication between the applications and the backend, i.e., it does not place a proxy between the application and the backend. This is important because it allows to preserve all the functional and security properties provided by the BaaS service API, since a proxy is a single point of failure.

NoSQL Undo [MC16] is a recovery approach and tool that allows administrators to automatically remove the effect of intrusions, as faulty operations on NoSQL databases. This article follows a more recent line of work inspired in systems as Shuttle [NC15] and Phoenix [CP05].

NoSQL Undo is a client-side tool in the sense that it does not need to be installed in the database server, but runs similarly to other clients. The tool only accesses the NoSQL database instance when the database administrator wants to remove the effect of some operations from the database, as malicious intrusions.

NoSQL Undo presents two different methods to recover a database: *Full Recovery*, that works by loading the most recent snapshot of the database and then updating the state to the present by executing the remaining operations, which were previously recorded in a log - a global log that is constructed by the NoSQL database mechanism. The algorithm takes as input a list of incorrect operations that it is supposed to ignore when it is executing the log operations. The other method is the *Focused Recovery*, that instead of rolling back the entire database just to erase the effects of a small set of incorrect operations, the algorithm only executes compensation operations, i.e., operations that corrects the effects of a faulty operations . The algorithm works in the following way: for each faulty operation, the affected record is reconstructed in memory by NoSQL Undo. When the record is updated, NoSQL Undo removes the incorrect record and inserts the correct one in the database.

As far as we know, NoSQL Undo is the only literarure that focuses on recovering NoSQL databases. The MIRES document's reconstruction algorithm is based on the Focused Recovery algorithm provided by NoSQL Undo.

Rectify [MPC17] 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*.

In the learning phase, samples are provided to the system, allowing Rectify to learn that a specific HTTP request will generate a certain kind of database statement. All the information gathered during the learning phase is captured and stored in a knowledge base, where each example is identified by an application route — an URL pattern that is mapped to a resource of the web application. In order to identify the database statements issued by a malicious HTTP request, Rectify needs to solve two classification problems:

- **Signature Matching -** consists of identifying the signature record of the malicious HTTP request. In this step, the malicious HTTP request is parsed in order to extract its relevant

parts (e.g., method, URL or parameters). Using those parsed parts, the classification algorithm is executed to find the corresponding signature record.

- **DB statements matching -** consists in finding in the DB log the actual statements that were created by the malicious HTTP request. Using the signature record from the signature matching, it is possible to find the corresponding database statements issued by the malicious request. First, the algorithm gets all the database statements of the signature record. Then Rectify calculates generic statements taking as an example the DB statements of the signature record and the parameter values from the malicious HTTP request. This generic statements should be as close as possible to the malicious DB statements, allowing a machine learning algorithm to identify them from the DB log.

Finally, Rectify removes the effects of an incorrect statement from the database by calculating a set of database statements — compensation transactions. In a simplistic scenario, in order to undo an insert it is necessary to execute a delete; to undo an update it is necessary to update the record back to its previous value. However, this problem becomes more difficult to solve in relational databases. In this kind of database, it is not recommended to remove a record that is related to other records because of the referential integrity constraints. In order to deal with this problem, an algorithm — two pass repair algorithm [LAJ00] — was used to calculate a graph of dependencies and undo the identified malicious actions.

MIRES architecture is based on Rectify architecture: both are deployed on a different container from the application container. However, Rectify does not need modifications on the applications, while MIRES requires the configuration of the application through a client-side package. Also, 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 Rectify.

Table 2.1 summarizes the main characteristics between the selected works, with the introduction of MIRES service. MIRES collects some of the ideas presented in the previous 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 on offline recovery models. MIRES also provides a new short-term recovery mechanism to mobile applications, supported by technique of parallel recovery processes that allows multiple users to recover their last action at the same time, which is an enhancement welcome in most applications where end-users can commit mistakes.

In the next sections, we will introduce mobile applications, with a special focus on the Android Operating System and applications.

Article	Recovery Approach	Target System	Recovery Type	User Recov.	Proxy Impl.
Undor for Operators [Brown, 2003]	Rollback	Email System	Offline	✗	✓
WARP [Chandra, 2011]	Rollback	Web App.	Online	✓	✓
Shuttle [Nascimento, 2015]	Rollback	Web App.	Online	✗	✓
NoSQL Undo [Matos, 2016]	Rollback and Compens. Trans.	NoSQL database	Online	✗	✗
Rectify [Matos, 2017]	Compens. Trans.	Web App.	Online	✗	✓
MIRES [Vaz,2020]	*Compens. Trans.*	*Mobile Application*	*Offline+*	✓	✗

Table 2.1: Comparison between the selected intrusion recovery works.

2.2 Mobile Applications

A *mobile application* [LSS04, XX13] 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. Mobile applications are usually shared with the consumers through *app stores*, e.g., Google Play[1] or Apple App Store[2], a digital distribution service of mobile applications.

Mobile applications can be divided into three types:

- **Native Application** - mobile applications targeted toward a particular mobile operating system. Generally, the main purpose of this type of application is to take advantage of the features of the target operating system, with good performance and more control over hardware (e.g., the Pokemon GO game);

- **Web Application** - web applications that deliver web pages on web browsers running on mobile devices. They do not need installation and cannot access all resource of the device, such as the camera or geolocation (e.g., Facebook accessed by the browser);

- **Hybrid Application** - a mix of native and web applications, as they need to be installed and also rely on web pages being rendered in a browser. This type of application is generally used when we do not have high-performance requirements but need full access to the resources of the mobile device (e.g., Instagram).

[1] `https://play.google.com/store`
[2] `https://www.apple.com/pt/ios/app-store/`

In the section below, we will focus our work on the Android OS and its architecture due to higher number of devices and users, reflected on a higher value on the market share [MOS].

2.2.1 Android Operating System

The *Android Operating System* [Mei12, GG17] is a Linux-based operating system maintained by the Open Handset Alliance, led by Google. It is a Java-based object-oriented application framework with a good memory and performance efficiency, highly tuned to hardware limitations of mobile devices. The Android OS assigns each deployed application with an unique user and group ID to preserve each application file privacy and implements the principle of least privilege, as applications must declare the permissions they need. When running, applications are sandboxed in separate *Android Runtime* (ART) Virtual Machines – the successor of the *Dalvik* Virtual Machine – which, in turn, runs within its own kernel managed process.

Figure 2.1 shows the overall architecture of the Android operating system. Android OS is organized in 4 layers, composed of 5 components: the *Kernel* that contains the hardware abstraction layer and components for low-level functionalities as memory management and inter-process communication. It also provides drivers for the display, touch input, networking, power management and storage; the *Android Runtime*, a process-based VM optimized for low memory and performance efficiency containing the *Android Runtime* VM, ensuring that multiple instances of the Android Runtime VM – each running an application – can run at the same time. The Android Runtime layer also contains Java-based libraries that are specific to Android development as Android.text, used to render and manipulate text on a device display, or Android.database, used to access data published by content providers; the *Systems Libraries* that stand between the kernel and the application framework layers, containing Android system libraries as libc, SQLite and OpenGL, all exposed through a Java API; the *Application Framework* that provides higher-level services used on mobile applications such as buttons and text boxes, common content providers so that apps may share data between them, a notification manager allowing device owners to be alerted of events and an activity manager for managing the lifecycle of application; and finally, the *Applications* component, the closest layer to the user, where the mobile applications live.

In the next section, we will introduce the Android application, its fundamentals and architecture.

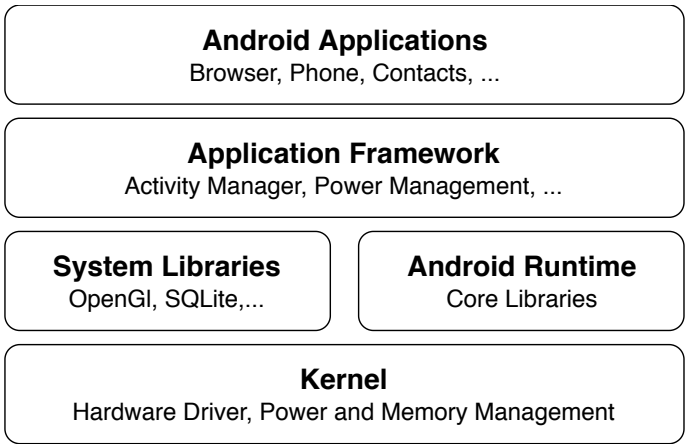

Figure 2.1: Android OS architecture (adapted from [GG17])

2.2.2 Android Application

An *Android application* [Mei12, GG17, MDMN12] is a software application written in a variant of the standard Java language with some differences, especially, in the user interface libraries. Android applications are installed as a single *Android Package* file (extension: .apk) containing the compiled code along with data and resource files.

Android applications are mainly built out of 4 types of components: *Activity* that represents a screen with a visual user interface and handles the user interaction with the mobile device. A typical Android application consists of, at least, one activity; *Service* used for background tasks, as time intensive tasks or inter-application functionalities, which do not require direct user interaction; *Content Provider* that manages store and access to the data of the application, as also provide a way to share data with other mobile applications; and *Broadcast Receiver* that handles and manages *intents* – an abstract description of an operation to perform – from the Android OS or the mobile application.

Besides the components above, an Android application can also contain other components as *Fragments*, that represent a portion of user interface in an Activity, *Views*, UI elements that are drawn on-screen like buttons or lists forms, *Layouts*, representing view hierarchies that control the screen format and appearance of the views, *Intents*, messages wiring components together, *Resources*, containing external elements, constants and drawable pictures and a *Manifest file*, the configuration file of the application, containing elements as the API level, the name of the application or the user permissions.

Figure 2.2 represents a possible example of a music player Android application sample and its architecture. A possible flow of this application could be as follows:

1. Activity A1 represents the first activity of the application – *Main Activity* – that contains two buttons: *login button* (View V1) and a *register button* (View V2). If the user clicks on

the login button, an intent is generated (Intent I1) with a login message and, consequently, a new activity is initiated (Activity A2);

2. Activity A2 represents the music page of the user, containing a *list* (View V3) with all the musics bought by the user and a *play button* (View V4). If the user clicks on the play button, a new intent is generated (Intent I2) in order to initiate the service (Service S1) that will play the album in the background.

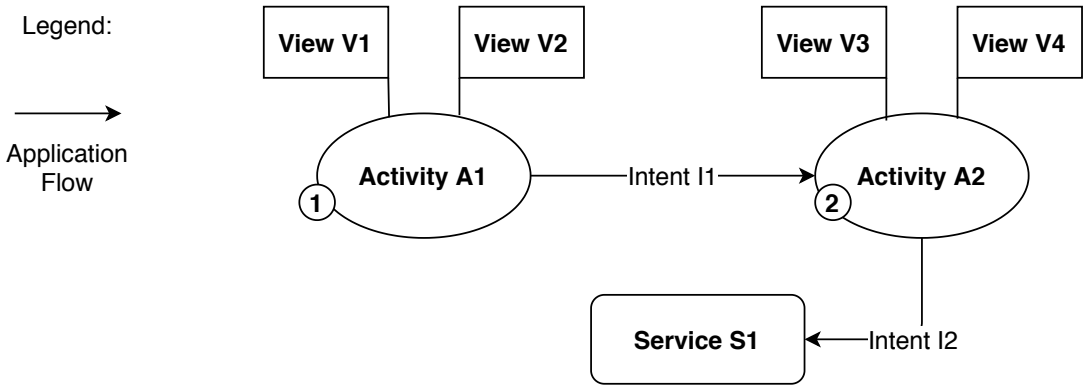

Figure 2.2: Representation of an interaction example between different components on an example Android music application.

Mobile application systems are software architecture composed by two major components: the *frontend*, that is the interface of the application, responsible for interacting directly with the user (e.g., activities and views), and the *backend*, responsible for maintaining the real functioning of the application, with which the user, typically, never interacts (e.g., data storage mechanism). Considering these two components, some mobile applications, generally the most basic ones, are local applications, having both frontend and backend on the application itself (e.g., a calculator application or notes application).

However, today's most mobile applications rely on remote services and resources provided by servers, often designated *clouds*, to support their functioning. Due to this, recently several frameworks/platforms have been appeared to support the development and execution of mobile applications (e.g., a bank or message application). These frameworks allow integrating code running on devices with remote services through APIs and where remote services are executed on the cloud and provide services and features like data and file storage, user authentication and notifications management. To simplify the development of these features, a new cloud service model emerged, named *Backend-as-a-Service*. .

In the next section we will start by introducing the cloud computing paradigm and then focus on the cloud computing model studied, the Backend-as-a-Service model.

2.3 Cloud Computing

Cloud computing [Mar17, MG+11, VRMCL08] is a computing model that aims to provide on-demand network access to a shared pool of elastic and scallable computing resources (e.g., networks, servers, storage, applications or services), motivated by the idea that storage and data processing can be done more efficiently on large computing farms accessible via the Internet.

This new computing paradigm is assured by *Cloud Service Providers* (CSP) (e.g., Google[3], Microsoft[4] or Amazon[5]), vendors who lease their cloud computing resources based on the dynamic use of their customers.

The Cloud Computing model is composed of five main characteristics: *On demand self-service* - computing resources and capabilities are provisioned automatically to the consumers without requiring human interaction; *Broad network access* - resource are available over the network and can be accessed through any platform, e.g., table, mobile phone or computer; *Resource Pooling* - physical and virtual computing resources location independent, i.e. the customer generally has no control or knowledge over their location and pooled into the cloud; *Rapid elasticity* - computing resources can be rapidly and elastically provisioned and released based on the demand of the consumer. To the consumer, the resources available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time; *Measured Service* - Resource usage can be monitored, controlled, reported and optimized by the CSPs through a pay-per-use business model (e.g., Business model similar to electricity or water services).

Since the necessities of each user can differ, the cloud computing environment offers different cloud computing models, each optimized for specific needs and demands.

In the next section we will present the main types of cloud computing service models.

2.3.1 Cloud Computing Services

The cloud computing environment is composed of different computing models, optimized for specific demands, where each cloud computing service model is characterized by a set of features and services whose management may be the responsibility of the user or the cloud computing provider. There are three main cloud computing service models:

- **Infrastructure-as-a-Service** where consumers does not manage or control the underlying cloud infrastructure but have control over operating systems, network, storage, and deployed applications;

[3] `https://cloud.google.com/`
[4] `https://azure.microsoft.com/en-us/`
[5] `https://aws.amazon.com/`

- **Platform-as-a-Service** where consumers have control only over the deployed applications or possibly configuration settings for the application-hosting environment and CSP vendors manage and control the underlying cloud infrastructure including network, servers, operating systems, or storage;

- **Software-as-a-Service** where all the responsibilities are transferred to the CSP vendor, from the underlying cloud infrastructure including network, servers, operating systems, storage, to individual application capabilities, with the possible exception of limited user specific application configuration settings.

On premises	**Infrastructure-as-a-Service (IaaS)**	**Platform-as-a-Service (PaaS)**	**Backend-as-a-Service (BaaS)**	**Software-as-a-Service (SaaS)**
Application (UI and Logic)	User	User	User	CSP
Data, Users and Auth	User	User	CSP	CSP
Runtime	User	CSP	CSP	CSP
Middleware	User	CSP	CSP	CSP
Operating Systems	User	CSP	CSP	CSP
Virtualization	CSP	CSP	CSP	CSP
Server Management	CSP	CSP	CSP	CSP
Storage	CSP	CSP	CSP	CSP
Networking	CSP	CSP	CSP	CSP

Table 2.2: Separation of responsibilities on each cloud computing service model (white/User cells represent the components managed by the user, while the grey/CSP cells represent the components managed by the cloud computing provider).

Table 2.2 summarizes the management responsibilities on the three original cloud computing models previously presented, where we have also introduced the Backend-as-a-Service. As we can observe, the Backend-as-a-Service cloud computing model stands between the PaaS and the SaaS: BaaS vendors manage the data storage, users and their authentication when compared with the Paas; on the other case, on SaaS, the application, user interface and logic are fully managed by the vendor, when compared with the BaaS.

In the section below, we will explore, with more detail, the Backend-as-a-Service model and analyze some examples of this new cloud service model.

2.3.2 Backend-as-a-Service

Backend-as-a-Service (BaaS) [Car16, FdS14, Lan15], 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. However, in this literature, 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[6], Back4App[7] and Parse.[8]

Typically, a BaaS service model provides a set of common application services including: *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* [MGZ+17] 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 2.3 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.

Despite being a recent cloud computing service model, there are already a great number of BaaS platforms, as the already mentioned Firebase, Parse or Back4app. Table 2.3 gives us an overview of some main features/services of these three BaaS. We choosed to focus our analysis on the client-side, analyzing the features/services provided by the three BaaS to the client, as we think that, when choosing the BaaS to use, it will be the *most complete* and *groundbreaking* BaaS, capable of supporting a more *robust* and *differentiated* mobile application system.

By analysing each BaaS example, we can conclude, from Table 2.3, that all the selected

[6] `https://firebase.google.com/`
[7] `https://www.back4app.com/`
[8] `https://parseplatform.org/`

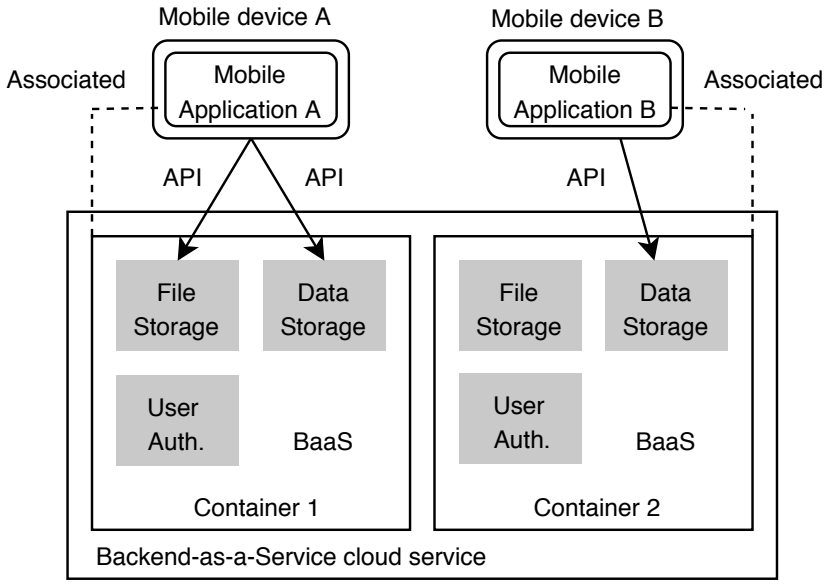

Figure 2.3: Arquitecture of a BaaS service model.

BaaS	**API**	**Datab. and File Storage**	**Push Notific.**	**User Auth.**	**Cloud Func.**	**Hosting Infrastr.**	**Machine Lear. Kit**
Firebase	✓	✓	✓	✓	✓	✓	✓
Back4App	✓	✓	✓	✓	✓	✓	
Parse	✓	✓	✓	✓	✓		

Table 2.3: Comparison between the features/services provided by three chosen BaaS services: Firebase, Back4App and Parse.

BaaS provide the same main features: *API*, *Database and File Storage*, *Push Notifications*, *User Authentication* and *Cloud Functions*. Analyzing now the hosting infrastructure aspect, we can observe that Parse is the only BaaS that does not offer this characteristic. The reason is that Parse is only an open-source framework that does not integrate the infrastructural-side, while Back4App and Firebase are platforms that deal with machine management and all the related problems, as *security settings*, *auto-scaling* or *database optimization*. An example of the Parse framework with the infrastructural-side is the Back4App platform, as Back4App is built on top of Parse framework. We also conclude that, in addition to the main services provided, Firebase also provides a *Machine Learning Kit*, which is a set of machine learning features for mobile use cases, as *recognizing text*, *detecting faces* or *scanning barcodes*.

After analyzing Table 2.3, we decided, to study and explore the Firebase platform, because, from the examples analyzed, Firebase is the most *robust* and *complete* platform that provides more than the default main features. In the next section, we analyze and explore the Firebase platform with more detail.

2.3.3 Firebase

Firebase [MMA17] is a backend development platform, led by Google, that provides a set of features and services that enable developers to create mobile and web applications. The platform is accessed throught the *Firebase Console* – a web page – and is built on three pillars: *Develop*, *Grow* and *Earn*.

The *Develop* pillar is focused on the application development experience and provides a set of features/services as: *Authentication* - provides different processes of authentication; *Realtime database* – a cloud-hosted NoSQL-based database [HHLD11] that provides syncing across all connected devices and support four types of requests: create, read, update and delete; *Cloud storage* – responsable for storing files as photos or videos; *Cloud functions* – allows to write code that runs in response to Firebase events, like database changes or HTTP events; *Cloud firestore* – an improvement of the Realtime Database, focusing on global apps; *Hosting* – service to host global web applications; *Machine learning kit* – a beta service that provides machine leaning features to improve the application (e.g., text recognition); *Test lab* – service that provides a set of mobile devices emulators, hosted by Google on a Test Center, for test porposes; *Crashlytics* – service that provides a stack trace of all applications crashes; *Performance monitoring* - presents a customized and automatic performance tracing of a mobile application, from the point of view of the user; and *App distribution* – a beta service that provides a simple way to send pre-realeased versions of the application to trusted testers;

The *Grow* pillar is related to helping developers to systematically improve and expand their apps, providing features/services as: *Firebase cloud messaging* - service that provides the capability to deliver messages to connected devices, at no cost; *Firebase remote config* - cloud service that provides server-side variables that allow to change the behavior and/or appearance of the application, without changing the application itself (e.g., an e-commerce app that, periodically, wants to provide discounts. This can be achieved by having a variable that contains the value of the discount); *In-App messaging* - a beta service that allows to send targeted and contextual messages to users that are actively using the application; *Predictions* - service that applies machine learning procedures to analytics data to create user segments based on predicted behavior; *A/B testing* - a beta service that simplifies the processes of running, analyzing and scaling product and marketing experiments; *App indexing* - provides the capability to open the application using Google Search; and *App invites and Dynamic links* - service that provides an intelligent and simple way to share the application with other users;

Finally, the *Earn* pillar provides a way to monetize mobile applications with in-app ad-

vertising that can be targeted to the user. This earning process is achieved by AdMob[9], a mobile advertising platform that helps to generate revenue from the mobile application. Firebase also provides a free-to-use analytics service, *Google Analytics*, that provides a number of common analytics (e.g., first open analytics: counts the number of first opens after installing or re-installing the application), as the ability to define custom analytics, that will be gathered without the necessity to write any code.

All the communication between the applications and the backend is achieved by the Firebase API, provided by an SDK for each supported language.

Cloud Firestore

BaaS database services can vary on the database supported, that can be relational or non-relational [JPA+12]. Some services allow the integration of external databases (e.g., Back4App allows the integration of MongoDB[10]), while others provide their own database, such as the *Cloud Firestore*[11] provided by the Firebase platform.

The Firestore database is a NoSQL cloud database to store and sync data for client and server-side development. It is a flexible and scalable database for mobile, web, and server development that keeps your data in sync across client apps through realtime listeners and offers offline support for mobile and web so you can build responsive apps that work regardless of network latency or Internet connectivity. Cloud Firestore also offers seamless integration with other Firebase and Google Cloud Platform products, including Cloud Functions.

NoSQL databases are of different types: key-value [Siv12], columnar [Vor11], or document-oriented, as the already mentioned MongoDB. In this case, the Firestore is a document-oriented model, where the database structure is based on *documents* and *collections*. MIRES does not depend on the specific database or if it is relational or not. However, the prototype uses the Firestore NoSQL database.

Firestore and others 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 also support *transactions* that provide the ACID properties (e.g., the Mongo Transactions[12]). This allows applications to perform writes – and in some cases reads – atomically in different documents. Firebase also supports *Firestore transactions*[13], that are divided in *transactions* and *batched writes*.

[9] `https://admob.google.com`
[10] `https://www.mongodb.com/`
[11] `https://firebase.google.com/docs/firestore`
[12] `https://www.mongodb.com/transactions`
[13] `https://firebase.google.com/docs/firestore/manage-data/transactions`

A Firestore transaction is a set of writes and reads performed atomically to one or more different documents, to a maximum of 500 documents. The transaction mechanism, performed by mobile applications, is executed in four phases in the following order:

1. **Read:** First, all the reads are performed on the target documents by the application;
2. **Logic:** Then, based on the data retrieved on the Read phase, some logic can be executed;
3. **Apply:** Finally, the mobile application apply the changes — update, delete or create — to the database documents;
4. **Double Check:** In the last phase, Firebase checks if, during the previous three phases, the documents read on the Read phase have changed. If it is false, then the transaction is executed. If it is true, then the transaction is re-executed (back to the first phase), but only a finite number of times (default number is 25, pre-defined by Firebase), then it aborts. Firebase transactions are built on top of the *optimistic concurrency control*, based on the idea that most of the time, transactions will probably execute. This idea is also based on that NoSQL databases are pre-configured for read-write distributions where reads will happen more frequently than writes, as it is the more typical access pattern in web applications, and so, the probability of another write request to change the same document read by the transaction is lower, increasing the probability of a transaction to be executed.

A Firestore batched write is a set of only write operations performed atomically to one or more different documents, to a maximum of 500 documents. This mechanism is used when we want to perform several writes on different documents without caring about the old values.

In the next section we debate the implementation model followed by the mobile application system using the Backend-as-a-Service, as also the threat models considered in this work.

2.4 Mobile Application System Model

This literature uses 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.

Mobile applications that use a BaaS backend have their *state* distributed between the mobile device and the cloud. In this work, we assume that the state of the application is reflected on a database service, which is the recovery object of MIRES. However, when using the BaaS, applications can interact with other services that, in turn, can also contain part of the system's

state. For a complete recovery, MIRES could be extend to interact and recover the other services. The distribution of the state is coordinated by executing remote services such as user management, file or data storage. We assume that the mobile applications always use data based on the backend state that is considered the authoritative copy of the data.

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.

2.4.1 Threat Model

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 work we assume that transactions are performed correctly and atomically, as 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 document.

2.5 Summary

This chapter presented the main ideas behind this work. The chapter introduced the intrusion recovery field, some definitions and previous works focused on the area; than described mobile applications and the different types, with a depth analysis on the Android Operating System, Android applications and the their main components; and finalized with cloud computing and the main cloud computing models, with a focus on the Backend-as-a-Service cloud computing model.

Chapter 3

MIRES

MIRES (*M*obile Applications *I*ntrusion *Re*covery *S*ervice) is an intrusion recovery service for mobile applications based on the BaaS model. 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.

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 the 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.

3.1 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 3.1 represents the architecture of

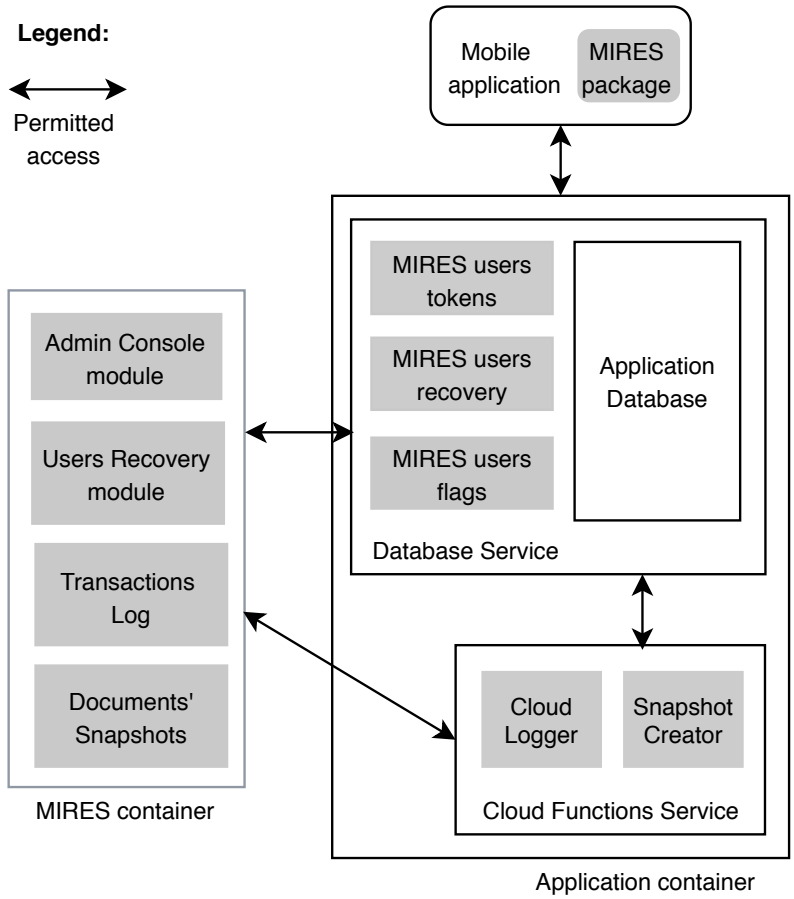

Figure 3.1: MIRES architecture on a mobile application system (MIRES components are shown in grey).

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 communicate with the application; the *MIRES users recovery* for undoing mistakes done by users (see Section 3.4); and *MIRES users flags* used for tracing the mobile application normal execution (see Section 3.2).

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 3.4) 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*.

In the next section we explain the normal execution of the MIRES service.

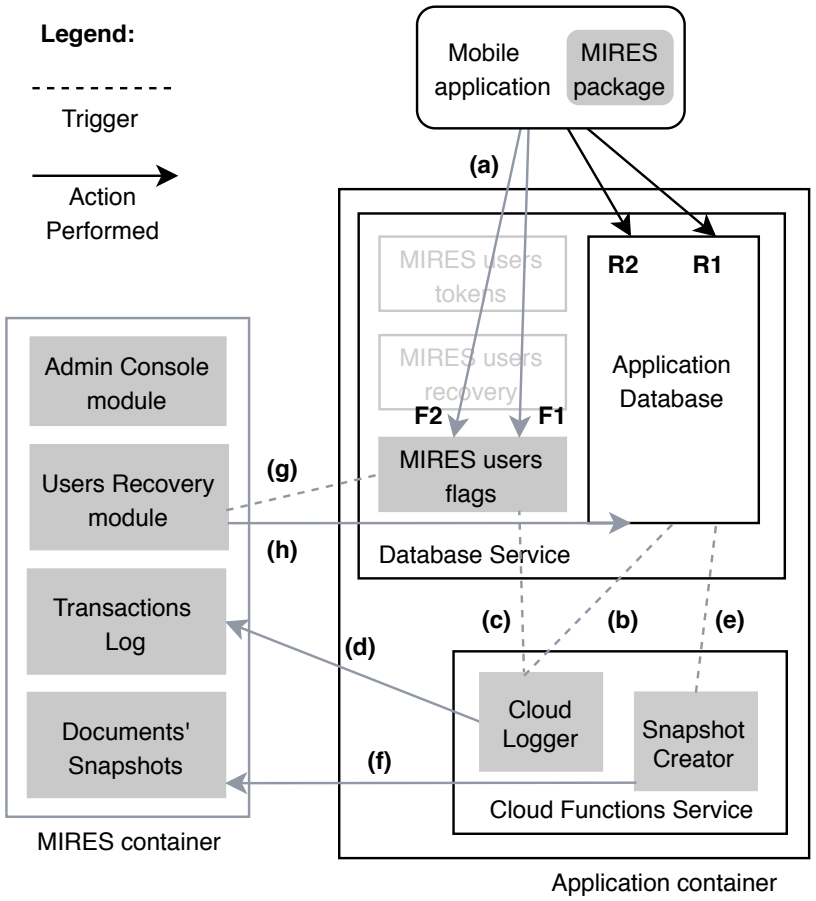

Figure 3.2: Normal execution flow of MIRES.

3.2 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 [BP03, AG10, CKS+11, CKZ13, NC15, MPC17]. This is important because it allows preserving all the functional and security properties provided by the BaaS service API, since a proxy is a single point of failure that can compromise the normal functioning of the application.

During normal execution, MIRES captures specific data of each transaction performed to the BaaS that, later, can be used on the recovery process. Figure 3.2 shows the normal operation of a mobile application system when using MIRES. In the next sections we will explain all the steps performed to a transaction during the normal execution of MIRES.

3.2.1 Mobile application configuration

MIRES Service uses a client-side package that can be introduced on the mobile application and enables the MIRES functioning.

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 3.2). Each write operation is configured to carry extra-data: an *operation ID* representing the operation itself; a *locked* property used in the recovery process (see Sections 3.3 and 3.4.2); 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, i.e., read operations cannot retrieved data from locked and blocked documents.

3.2.2 Logging Process

MIRES logging process is achieved by using two mechanisms: *Flags* and *Cloud Loggers*. Flags carry specific operation information (Section 3.2.2). Cloud Loggers are cloud functions [MGZ+17] that log the requests made by the mobile applications (Section 3.2.2).

For each operation that alters the state of the database – create, update and delete operations – MIRES gathers the *type* of the operation, the *timestamp* associated, the *document changed*, *data associated with the operation*, a *transaction ID*, that associates all requests of the same transaction (in the figure, both *R1* and *R2* carry the same transaction ID) and part of the additional information generated by the MIRES package for each operation (see Section 3.2.1).

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

Flags

Flags are MIRES resources that carry information about the associated operation. 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 3.2). Each flag is responsible for sending additional information needed to log the operation. 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()*, are made on the

mobile application but the logic is only executed on the backend.

In these cases, the log information is completed by the *Cloud Logger* (see Section 3.2.2). 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 are 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.

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 3.3.3; and arrows *g*, *h* relate to user recovery and are explained in Section 3.4.1.

3.2.3 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 3.3.2). 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. To achieve this, MIRES uses the Cloud Loggers to complete the dependency information.

In the next section we explain the administrator recovery mechanism supported by the MIRES service.

3.3 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. In the rest of this section we will explain both phases.

3.3.1 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 [ARF+14, GQTZ16, YMHC17], but, as previously mentioned, this mechanism is orthogonal to recovery and out of scope of this work. 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 any write operations, 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.

3.3.2 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.

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 3.2.3), 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.

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.

3.3.3 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 the unaffected database documents while MIRES reconstructs the corrupted documents.

Operations model

The reconstruction model adopted was based on the *Focused Recovery* algorithm of NoSQL Undo [MC16]. 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.

Snapshots model

This recovery model is improved using *snapshots* [LD97], 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 3.2.1 – 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 3.2) and stores a document snapshot after N operations made to the document (arrow f of previous Figure 3.2), 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.

In the next section we explain the user recovery mechanism supported by the MIRES service.

3.4 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[1].

3.4.1 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 3.2.1 and 3.3.3. 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.

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 3.2). 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 3.2).

3.4.2 Recovery Execution

Figure 3.3 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 3.3.1. After locking the documents, the mobile application sends a recovery request to MIRES users recovery carrying the transaction ID to be recovered, the documents locked and a timestamp associated with the recovery request (arrow b). Then, the User Recovery module gets the

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

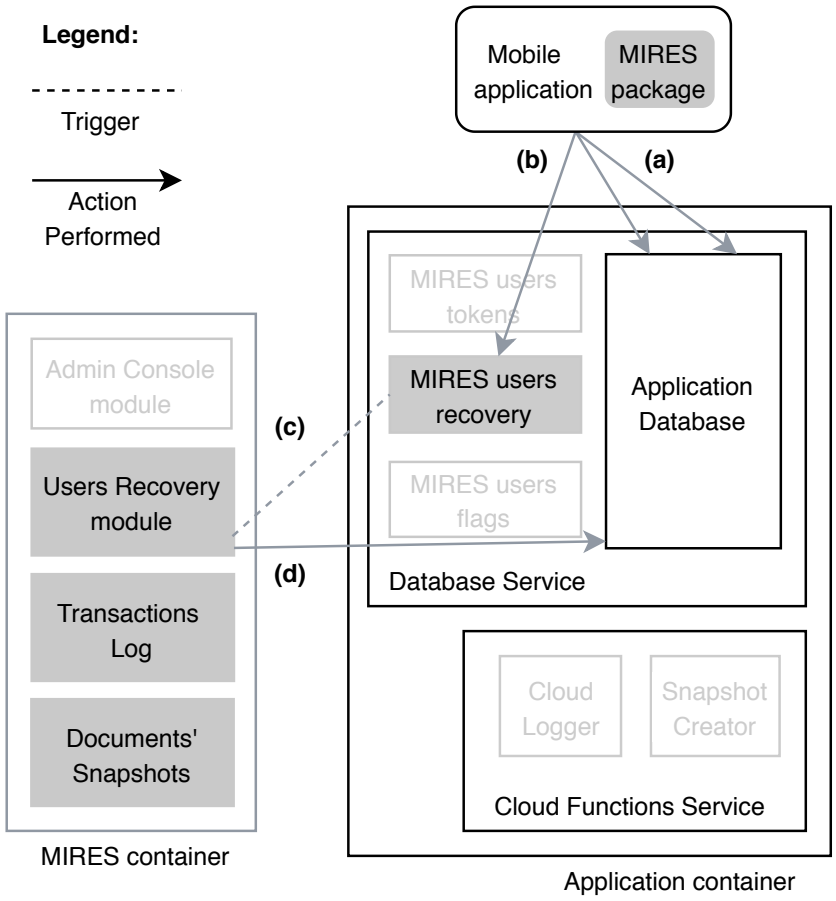

Figure 3.3: User Recovery mechanism.

recovery request from the user (arrow c) and reconstruct the documents affected similar to the reconstruction phase of Section 3.3.3 (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. This allows the recovery of multiple transactions from different users at the same time, without requiring the mobile application system to stop. MIRES only needs to certified that contains in the log all operations performed to the documents affected by the transaction that the user wants to recover. 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.

In the next section we explain how we have implemented the MIRES prototype.

Table 3.1: MIRES package lines of code (LoCs).

MIRES package	LoCs
Tokens	47
Notifications	52
Transaction configuration	174
Undo Recovery Mechanism	198

Table 3.2: MIRES modules lines of code (LoCs).

MIRES modules	LoCs
Cloud Logger (flags)	26
Cloud Logger (collection)	63
Snapshot Creator	122
Users Recovery module	558
Admin Console module	913

3.5 Implementation

MIRES offers a client-side package (Table 3.1) that allows to configure: 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 notifications created by MIRES service on the mobile application and the locking phase of the user recovery mechanism.

The package was implemented in Java, which is the most frequent option for Android applications.

MIRES was implemented as a three layer service (Table 3.2): 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; and a third layer composed by the Snapshot Creator that supports the creation of snapshots. With this, MIRES offers *flexible* and *adaptable* configuration: modules are deployed depending on the functionality that we want to use. However the Cloud Loggers are needed to build the log of transactions in order to use any type of recovery. Both Admin Console module and Users Recovery module were implemented using Node.js and JavaScript and can be deployed to isolated containers, which provides an important security aspect.

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., on a social network application, the system administrator could want to protect only the posts performed by the users. In this case it is only required to deploy a Cloud Logger that listens for the collection that stores the posts and configures the transactions that interact with the same

collection.

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 applied an atomic model – using Firebase transactions – on the operations and their flags, as also on the locking phase of the users recovery mechanism, allowing to mitigate possible synchronization problems.

3.6 Summary

This chapter presented the MIRES approach. The chapter started with the description of the architecture that supports the service. Then focused on all execution phases supported by the service: the normal execution and both administrator and user recovery executions, as also the mechanisms and the ideas behind. The chapter finalized with the description of how the MIRES prototype was implemented.

Chapter 4

Evaluation

Our 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?

In the next sections, we present the results that respond to the questions defined.

4.1 Experimental Evaluation

To evaluate the MIRES service, we used four open-source Android applications: a *social network application*, Hify,[1] where users can post images or text, comment and like other posts; a *messaging application*[2] for 1 to 1 conversations; a *shopping list application*, ShoppingListApp,[3] to create shopping lists, by adding, changing and removing products; and a *contact tracking application*, CovSense[4], used to track contacts between their users and manage the COVID-19 spread. The social network and messaging applications were chosen based on the intensive users'

[1] Google Play: `https://play.google.com/store/apps/details?id=com.amsavarthan.social.hify`; Source code: `https://github.com/lvamsavarthan/hify` version of 06/07/2020

[2] `https://github.com/ResoCoder/firebase-firestore-chat-app` version of 19/08/2020

[3] `https://github.com/alexmamo/Firestore-ShoppingListApp` version of 07/07/2020

[4] `https://github.com/saivittalb/covsense` version of 27/09/2020

dependencies created on the backend, and because they represent the logic of 4 out of 5 of the most downloaded apps of 2019 [Sen20]. The shopping list was choosen due to its different logic, since it is a more individual application. The CovSense application was choosen due to the actual pandemic context that humankind is facing.

In the next section we provide additional information about the four Android applications used in the experiments.

4.1.1 Mobile Applications

We used four Android open-source applications to test the MIRES service. In this section we provide supplementary information about each application.

Hify is an open-source social network application where users can share updates and photos, engage with friends and other users worldwide and stay connect to the world. The application presents features such as: sharing photos (up to 7 in a single post) and updates; get notifications when friends like and comment on your posts; ask questions in the Hify Forum as also help others with their questions; and connect with friends and family and meet new people. In terms of recovery, the data in the database that MIRES will recover is related with the users accounts and their main actions on the application, more specifically, posts, comments and likes.

The Chat application is a very simple messaging application where users can start 1 to 1 conversations with another user. After creating an account, users can search for a specific user – by introducing his username – and start a conversation, with the possibility to send images or text messages. We have focused our experiments in recovering data related with the users accounts, chats and the messages.

The Shopping Lists application is a basic Android application for shopping lists management. Lists and products are created through the definition of a name to each one. Each user can create, update or delete a shopping lists and then add, update or delete products in each list. Besides each list is created by a single user, list can be shared with other users. MIRES will focus its recovery on the data about users accounts, the lists managed by the user and the products in each list.

The CovSense application is a tracking application for the COVID-19 virus. The application uses a combination of Wi-Fi, Bluetooth, BluetoothLE and ultrasonic modem to communicate an unique-in-time pairing code between devices, allowing to store the contacs between users. Application actions are: create account, update health status – between “Healthy” and “Diagnosed with COVID-19 ” – and the store contacts with other users, that is done automatically by the application. After an user changes his health status to “Diagnosed with COVID-19”,

all users that contacted with him are notified about a possible infection. In this application, MIRES will apply its recovery approach on the users accounts and the contacts stored between different users.

All applications use Firebase as BaaS and the Firestore database. 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 (`europe-west2`) 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 in the next sections are averages of the results obtained with the 4 applications, except when noticed.

4.1.2 Logging Evaluation

MIRES Performance Overhead

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 interaction by performing a set of actions on each application – by reading, creating, updating and delete posts, comments and likes on the social network application; reading, creating, updating and delete chats and messages on the messaging application; reading, creating, updating and delete lists and products on the shopping list application; storing contacts with other users and changing the health status on the CovSense application – that resulted on 1K CRUD operations executed per application. 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, 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 most of the application main logic is based on writes.

Figure 4.1 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. When compared

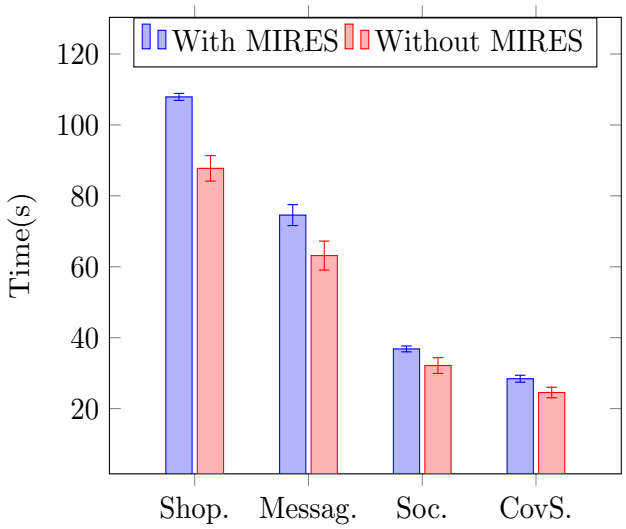

Figure 4.1: Time to perform 1000 operations on each application, with and without MIRES.

with the other applications, the CovSense application presents a lower test overhead. This is justified by the fact that most of the operations done – storing contacts with other users – were made concurrently in order to simulate the real life context where users can make contact with multiple persons at the same time. Although the overhead is noteworthy, we consider it acceptable, given the benefit provided by the service. MIRES adds a create operation for each write operation made on the database (flag process), which contributed to increasing the cost of running the service, since Firebase is charged per cluster of operations (each cluster of 100K operations costs 0.18$ at the time of the evaluation).

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 0.39±(0.18) seconds and 0.09±(0.01) GB of memory to execute. Firebase offers a free quota of 2M invocations per month. After that, each 1M of invocations costs 0.40$ (however the Cloud Functions service is also priced in GB/second, the CPU/second and the Internet traffic[5]).

[5] `https://firebase.google.com/pricing`

4.1.3 Space Overhead

Database overhead

MIRES configures each operation to send additional data (see Sections 3.2.1, 3.3.3 and 3.4.1). The additional data is saved on each database document, which imposes a storage overhead. Firebase provides full information about the storage structure of the database[6]. 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 an 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. However, the mobile application system can be already storing the users' tokens which allows to mitigate the MIRES users tokens by reusing the information already stored.

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.

Log records

MIRES stores specific data in each operation made to the database (see Section 3.2). Each log record size is given by the following expression (in bytes):

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

[6] `https://firebase.google.com/docs/firestore/storage-size`

Table 4.1: Log size on each application.

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

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. For example, an operation that creates a 20-character name document on a collection named Posts will lead to 26 bytes of *doc* property (*Posts/* + 20-character string). If the operation writes 344 bytes, then the log record of the operation will have 215+26+(53+344) = 638 bytes.

Dependencies

When a write operation is influenced by a read operation, there is an additional information logged related with the read operation (see Section 3.2.3). 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. For instance, a read operation on the *id*, *username*, *name* and *image* fields of the users' information document will lead to 19 bytes of field-values read. Supposing that the read operation is performed upon the document *Users/* + user ID, a 28-character identifier, this will result on a *doc* property of 34 bytes. Thereby, the read operation would increase the log record size by 91+(34+1+(19+4)) = 149 bytes.

Table 4.1 shows the log size needed to store 1M operations following the exact workflow performed on Section 4.1.2 on each application. Firestore offers a free quota of 1GB. After that, each 1 GB costs 0.18$.

Snapshots

The size of each snapshot made by the Snapshot Creator is defined by the following expression (in bytes):

$$S_{snap} = 397 + doc + data$$

where the *doc* property represents the path of the document and the *data* property represents the data of the snapshot to store. For instance, making a snapshot of a document with the path *Posts/*+20-character string (total of 26 bytes) and 344 bytes of data will lead to 397+26+344 = 767 bytes to store.

4.1.4 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 different scenarios. In *scenario 1* we have created an user in each application and performed a different number of actions – creating, updating and deleting posts, comments and likes on the social network application; creating, updating and deleting chats and messages on the messaging application; creating, updating and deleting lists and products on the shopping list application; and storing contacts between other users and changing the health status – resulting on different recovery scenarios, from 1 to 1000 operations. In *scenario 2* we used the same user and the application actions to perform 1 to 10K operations upon the same document – changing the same post on the social network application; changing the same message on the messaging application; changing the same list product from a list on the shopping lists application; and changing the health status – in order to create 1 to 10 000 different versions of the document. In *scenario 3* we exemplified the case where the effects of a malicious intrusion are not persisted in the database: in this case, we used the CovSense application, where the user changed his health status to "Diagnosed with COVID" and a notification was sent to all users that had been in contact with the infected user. In this case, the posterior effects of the intrusion are not persisted in the database, however a malicious information is sent to some users. Each recovery scenario was executed 5 times. In the rest of this section, we analyse the MIRES recovery performance on all scenarios.

Scenario 1

Scenario 1 was focused on recovering a different number of actions. Figure 4.2 shows the results of undoing the actions of the user. Both 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.

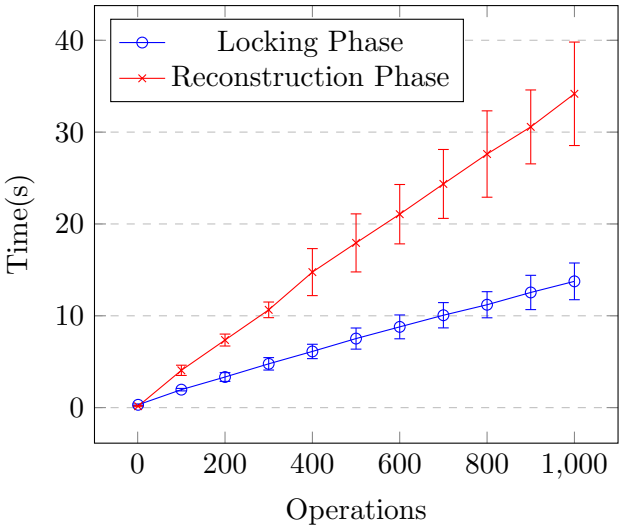

Figure 4.2: Time to undo a different number of operations.

The Locking phase is composed by the load and analysis of the log to identify the malicious transactions and the corrupted documents. This phase presents a drawback: MIRES can lock documents where the state before and after the recovery process is the same, so the documents could be ignored increasing the *locking accuracy*. However, we have not implemented this optimization, since we believe that these scenarios will be rare.

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, where the service identifies posterior effects and, consequently, the infected documents, while on the Reconstruction phase, for each document infected and locked, MIRES needs to load the operations that affect the document to reconstruct – for legitimate operations – or update the log – for malicious operations.

Scenario 2

Scenario 2 is focused on testing the time to recover a document with different versions. Figure 4.3 shows the results of the experiment. We can observe that, when using the operations model, MIRES needs less than 0.5 seconds to reconstruct a document with 1 version and between 2 and 3 seconds to reconstruct a document with 10K versions. By following the operations 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.

Besides the operation model, we also have tested the reconstruction of the document using the snapshots model – however, due to code conflicts between the libraries used on the mes-

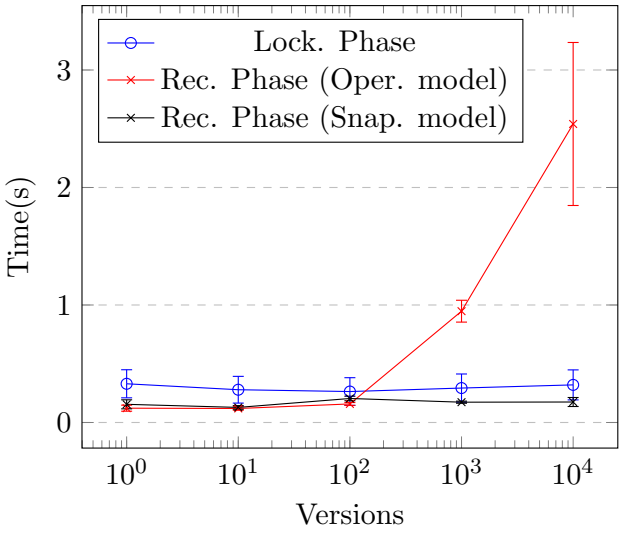

Figure 4.3: Time to reconstruct a document with different versions.

saging application and a newer version of the Firebase Android SDK, the reconstruction using the snapshots model was tested only on the Social network application, the Shopping Lists application and the CovSense application. From the Figure, we can observe that, as previously discussed (see Section 3.3.3), the reconstruction time using the snapshots model is consistent over the time – an average of 0.2 seconds.

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 applications.

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

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. With CovSense we simulated a malicious health status change – from "Healthy" to "Diagnosed with COVID-19" – that, in turn, generated a flow of malicious notifications sent to all users that had been in contact with the malicious infected user. In this case, MIRES allows to send recovery messages, through application notifications, to each user. We tested the notifications mechanism by sending 1, 10 and 100 notifications. We performed each test 5 times.

Table 4.2: Time to send different number of notifications.

Notifications	Time (s)
1	0.06±(0.03)
10	0.81±(0.04)
100	5.80±(0.58)

Table 4.2 represents the time to send a different number of notifications to the mobile applications.

4.1.5 Users Recovery Performance

Similar to the administrator recovery mechanism, the user recovery mechanism is supported by two phases: a locking phase, where the mobile application itself locks the documents affected by the transaction, and a reconstruction phase performed by MIRES Users Recovery module, where the locked documents are reconstructed by the MIRES service.

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 – creating posts on the social network application, sending messages on the messaging application and adding products to lists on the shopping list application – 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.

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 on each mobile application, more precisely, locking posts on the social network application, messages on the messaging chat application and products on the shopping lists application. 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 4.1.4).

4.2 Discussion

With the four applications, we have performed a wide range of different experiments, from performance, to storage and recovery. From the experiments conducted, we could conclude that:

- MIRES imposes an overhead that varies with the type of operation performed: write operations imposes a greater overhead than read operations, e.g., the overhead imposed on the Shopping Lists application with a 20/80 read/write flow is 23%, while on the Social Network application with a 80/20 read write flow is only 15%.
- The use of the Cloud Functions service allow to provide automatic scaling logic with great performance, e.g., the cloud loggers and the snapshot creator scripts deployed on the Cloud Functions service took less than 0.5 seconds and less than 0.1 GB to execute;
- MIRES service requires low storage capacity, e.g, from the tests performed, it requires a maximum of 0.42 GB to store 1 million of requests of the CovSense application;
- As theoretically predicted, the MIRES recovery approach allows to recover the state of the application with a real focus on optimizing the availability of the system during the process. For example, when recovering 1000 operations, MIRES takes around 55 seconds maximum but the mobile application system is unavailable for only 15 seconds maximum;
- Both document reconstruction models work as expected: the operation model time to reconstruct the document increases with the increase of the number of document's versions – less than 0.2 seconds with 1 version and between 2 and 3 seconds with 10 000 versions – while the snapshots model maintains its performance with the variation of the number of document's versions – average of 0.2 seconds.

4.3 Summary

This chapter presented the results of the experiments made with the MIRES service on four different mobile applications systems: a social network application, a messaging chat application,

a shopping lists application and a contact tracing application. The experiments were performed to test the performance of the service, the storage imposed by the service and the time needed to recover the application in different scenarios.

Chapter 5

Conclusions

In this chapter we present our achievements and point to future work directions.

5.1 Achievements

With the intensive development of mobile applications, the use of Backend-as-a-Service became a viable way to develop software systems based on mobile or web applications. However, errors during analysis or development can lead to the existence of possible vulnerabilities on mobile applications that, in turn, can be explored to attack the Backend-as-a-Service, affecting the integrity of the system.

In this document we have presented various recovery systems. However none of them can recover the state of mobile applications using Backend-as-a-Service, as none of them are prepared to deal with the specific problems and challenges of a Backend-as-a-Service. To fill this gap, this work 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.

We applied MIRES to recover mobile applications systems based on BaaS. However, it is possible to use the MIRES approach to recover other applications and services, e.g., web applications.

5.2 Future Work

Besides the already present features as the hybrid recovery process and the user recovery mechanism, MIRES could be improved in various points in order to apply a more robust and complex

recovery. Therefore, there are many possible improvements: evolve MIRES to a full online recovery process, provide users a recovery mechanism capable of recovering data written by different users, extend MIRES to recover other services such as the authentication service or the file storage service, mitigate part of the client-side package by using a machine learning technique to log the operations and extend the recovery approach to log and recover *Cloud Functions* work.

MIRES provides a hybrid recovery approach that, nevertheless, needs to stop the system during the recovery. The idea of providing a full online recovery process is to mitigate this stopping time during the process. There are some works on the field that already implement a full recovery process, however based on the existence of a proxy that allows to introduce delays on the new transactions made during the recovery process. In our case, since we do not use any kind of proxy, the idea would be to introduce delays following another approach, e.g., introduce the delay directly on the affected document, that in turn would affect only the users that want to access it.

The user recovery mechanism provides the capacity to users recover from their actions. However, it is only capable to recover data only written by the same user, e.g., a post. MIRES could be improved to allow recover any also data written by different users, e.g., a shared counter. A possible solution could be to verify if, when the user tries to block the documents and send the recovery request, check if the last transaction performed on the documents is the transaction that the user wants to revert and continue with the process only in that case.

The scope of this work was to recover the database service of the BaaS, since this service contains the majority of the system's state. However, other services such as the authentication and file storage services can also contain part of the system's state that could be recovered. To deal with this cases and provide a more complete recovery, MIRES could be extended to interact and recover different services that can be infected, following previous works as [CKZ13].

Rectify uses a novel logging approach by using a supervised machine learning technique, allowing to recover web application using PaaS without needed to change or configure the application itself. MIRES, on the other case, imposes application code modifications through the MIRES Package. Following Rectify approach, MIRES could be improved in order to decrease the amount of modification required, or even mitigate all necessary modifications to the application, using machine learning techniques.

The last improvement point is not directly related with mobile applications, but with another service: the *Cloud Functions* service. When interacting with the mobile application systems, users' actions can trigger or interact with cloud functions scripts that will change the state of the mobile application and so, it would be interesting to recover. A possible simple solution

could be to develop a *Cloud Functions package* – similar to MIRES Android application – that would allow to log the operations following the same model already implemented. A second solution, more complex and interesting, could be running the cloud functions again: since cloud functions scripts are deployed on the cloud and run with changes on the database, MIRES could reconstruct the documents directly in the database, allowing to trigger and run the cloud functions again with the new recovered data.

Bibliography

[AG10] İstemi Ekin Akkuş and Ashvin Goel. Data recovery for web applications. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 81–90, 2010.

[AJL02] P. Ammann, S. Jajodia, and P. Liu. Recovery from malicious transactions. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1167–1185, 2002.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl 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.

[ARF+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick 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.

[BHS13] Sven Bugiel, Stephan Heuser, and Ahmad-Reza 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.

[BP03] Aaron B Brown and David A Patterson. Undo for operators: Building an undoable e-mail store. In *USENIX Annual Technical Conference*, pages 1–14, 2003.

[Car16] Brian Carter. Grow your own backend-as-a-service (baas) platform. In *GOCICT 2015 Conference College of Information & Computer Technology*, November 2016.

[CKS+11] Ramesh Chandra, Taesoo Kim, Meelap Shah, Neha Narula, and Nickolai Zeldovich. Intrusion recovery for database-backed web applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 101–114, 2011.

[CKZ13] 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.

[CP05] Tzi-Cker Chiueh and Dhruv Pilania. Design, implementation, and evaluation of a repairable database management system. In *21st International Conference on Data Engineering (ICDE'05)*, pages 1024–1035, 2005.

[FdS14] Joao André Lopes Ferreira and Alberto Rodrigues da Silva. Mobile cloud computing. *Open Journal of Mobile Computing and Cloud Computing*, 1(2):59–77, 2014.

[GG17] Dawn Griffiths and David Griffiths. *Head first Android development: A brain-friendly guide*. O'Reilly Media, Inc., 2017.

[GMUW08] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, USA, 2 edition, 2008.

[GPF+05] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal De Lara. The taser intrusion recovery system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 163–176, 2005.

[GQTZ16] Keke Gai, Meikang Qiu, Lixin Tao, and Yongxin Zhu. Intrusion detection techniques for mobile cloud computing in heterogeneous 5G. *Security and Communication Networks*, 9(16):3049–3058, 2016.

[GTDVMFV09] Pedro Garcia-Teodoro, Jesus Diaz-Verdejo, Gabriel Maciá-Fernández, and Enrique Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *computers & security*, 28(1-2):18–28, 2009.

[HCR+06] Francis Hsu, Hao Chen, Thomas Ristenpart, Jason Li, and Zhendong Su. Back to the future: A framework for automatic malware removal and system repair. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 257–268. IEEE, 2006.

[HHJ+11] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David 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.

[HHLD11] Jing Han, Ee Haihong, Guan Le, and Jian Du. Survey on nosql database. In *2011 6th international conference on pervasive computing and applications*, pages 363–366. IEEE, 2011.

[JPA+12] Nishtha Jatana, Sahil Puri, Mehak Ahuja, Ishita Kathuria, and Dishant Gosain. A survey and comparison of relational and non-relational database. *International Journal of Engineering Research & Technology*, 1(6):1–5, 2012.

[JSDG08] Shvetank Jain, Fareha Shafique, Vladan Djeric, and Ashvin 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.

[KC03] Samuel T King and Peter M Chen. Backtracking intrusions. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 223–236, 2003.

[KLS90] Henry F Korth, Eliezer Levy, and Abraham Silberschatz. *A Formal Approach to Recovery by Compensating Transactions*. University of Texas at Austin, Department of Computer Sciences, 1990.

[KWZ+10] Taesoo Kim, Xi Wang, Nickolai Zeldovich, M Frans 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.

[KWZK10] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. Intrusion recovery using selective re-execution. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, pages 89–104, 2010.

[LAJ00] Peng Liu, Paul Ammann, and Sushil Jajodia. Rewriting histories: Recovering from malicious transactions. In *Security of Data and Transaction Processing*, pages 7–40. Springer, 2000.

[Lan15] Kin Lane. Overview of the backend-as-a-service (BaaS) space. *API Evangelist*, 2015.

[LD97] Jun-Lin Lin and Margaret H Dunham. A survey of distributed database checkpointing. *Distributed and Parallel Databases*, 5(3):289–319, 1997.

[LSS04] Valentino Lee, Heather Schneider, and Robbie Schell. *Mobile Applications: Architecture, Design, and Development*. Prentice Hall PTR, USA, 2004.

[Mar17] Dan C Marinescu. *Cloud computing: theory and practice*. Morgan Kaufmann, 2017.

[MC16] D. Matos and M. Correia. NoSQL undo: Recovering NoSQL databases by undoing operations. In *IEEE 15th International Symposium on Network Computing and Applications*, pages 191–198, 2016.

[MDMN12] Zigurd R Mednieks, Laird Dornin, G Blake Meike, and Masumi Nakamura. *Programming Android*. O'Reilly Media, Inc., 2012.

[Mei12] Reto Meier. *Professional Android 4 application development*. John Wiley & Sons, 2012.

[MG+11] Peter Mell, Tim Grance, et al. The NIST definition of cloud computing. 2011.

[MGZ+17] Maciej Malawski, Adam Gajek, Adam Zima, Bartosz Balis, and Kamil Figiela. Serverless execution of scientific workflows: Experiments with hyperflow, aws lambda and google cloud functions. *Future Generation Computer Systems*, 2017.

[MMA17] Laurence Moroney, Moroney, and Anglin. *Definitive Guide to Firebase*. Springer, 2017.

[MOS] Mobile operating system market share worldwide. `https://gs.statcounter.com/os-market-share/mobile/worldwide`, last accessed on 25/09/2020.

[MPC17] David R. Matos, Miguel L. Pardal, and Miguel Correia. Rectify: Black-box intrusion recovery in paas clouds. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, page 209–221, 2017.

[MPC18] 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.

[NC15] Dário Nascimento and Miguel Correia. Shuttle: Intrusion recovery for paas. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 653–663, 2015.

[OCW+08] 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.

[Pos19] Positive Technologies. Vulnerabilities and threats in mobile applications, 2019. 6 2019.

[Sen20] Sensor Tower. Q4 2019 store intelligence data digest. 2020.

[SFH+99] 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.

[SGS+00] 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.

[Siv12] Swaminathan 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.

[TW] Stephen Thomas and Laurie Williams. Using automated fix generation to secure sql statements. In *3rd International Workshop on Software Engineering for Secure Systems (ICSE Workshops 2007)*.

[Vor11] Mehul Nalin Vora. Hadoop-hbase for large-scale data. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, pages 601–605, 2011.

[VRMCL08] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, pages 50–55, 2008.

[XJL09] 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.

[XX13] Spyros Xanthopoulos and Stelios 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.

[YMHC17] Sileshi D. Yalew, Gerald Q. Maguire Jr., Seif Haridi, and Miguel 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*, October 2017.

[ZC03] 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.

[ZWW+10] Chao Zhang, Tielei Wang, Tao Wei, Yu Chen, and Wei Zou. Intpatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In *European Symposium on Research in Computer Security*, pages 71–86. Springer, 2010.