MIRES: Intrusion Recovery for Applications based on Backend-as-a-Service

Diogo Vaz, *Student Member, IEEE,* David R. Matos, *Member, IEEE,* Miguel L. Pardal, *Member, IEEE,* and Miguel Correia, *Senior Member, IEEE*

***Abstract*—The Backend-as-a-Service (BaaS) cloud computing model supports many modern popular mobile applications because it simplifies the development and management of services such as data storage, user authentication, and notifications. However, vulnerabilities and other issues may allow malicious actions on the client side to have impact on the backend, i.e., to corrupt the state of the application in the cloud. To deal with these attacks – after they occur and are successful – it is necessary to remove the direct effects of malicious requests and the effects derived from later operations on corrupted data.**

We introduce MIRES, the first intrusion recovery service for mobile applications based on the BaaS model. MIRES uses a two-stage recovery process that restores the integrity of the mobile application and minimizes its unavailability. MIRES provides multi-service recovery for applications that use more than one data store. We implemented MIRES for Android and for the Firebase cloud-based BaaS platform. We did experiments on 4 mobile applications which showed that MIRES can revert hundreds to thousands of operations in seconds, with an associated unavailability of the application also in the range of seconds.

***Index Terms*—Intrusion Recovery, Backend-as-a-Service, Cloud, Mobile Applications.**

I. Introduction

For many years now, mobile applications have played an important role in our lives, as they provide daily-use services like message chats, social networks, online banking, and file storage, just to name a few [1], [2]. Due to the inherent benefits of mobile applications, such as portability, usability, and connectivity, companies are convinced to use mobile applications as client interfaces for their services [3]. Recently, even personal health monitoring has gained adoption with COVID-19 contact tracking applications [4]. A mobile application is a program that runs on a mobile device, typically *smartphones* or *tablets*, but also *smartwatches* or *car dashboards*. Most mobile applications rely on remote services and resources provided by servers, often designated *clouds*, to support their normal operation [5].

Recently, several frameworks and platforms appeared with the objective of supporting the implementation and execution of mobile applications. A highly successful case is the *Backend-as-a-Service* (BaaS) cloud model [6]–[8] that allows 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, the Lyft car sharing platform, The Economist magazine's portal, and AliBaba's messaging service.[1]

Despite our increasing reliance on these applications, they are complex software systems that often contain *vulnerabilities*, e.g., due to improper user input validation and other errors made by developers in designing and/or writing code [9], [10]. These vulnerabilities can be exploited by malicious actors with the intent of corrupting the state of the application stored on the backend, leading to *intrusions*. A recent study found that 2-out-of-3 mobile application vulnerabilities are medium/high risk and that 60% of them are on the client-side [10].

This paper presents the first *intrusion recovery* approach and service for mobile applications based on the BaaS model. In a nutshell, intrusion recovery is the problem of reverting the effects of an intrusion on the state of an application. The classical, simplistic, intrusion recovery solution is to periodically backup the application state – creating *snapshots* – and, when the state is compromised, to replace it with the most recent snapshot. This basic solution leads to data loss, as backups are almost always outdated, e.g., hours or days, depending on their frequency. This data loss is the first problem we aim to solve. Some recovery mechanisms provide a finer-grained approach by considering not only snapshots but also the operations since the last snapshot, which are stored in a *log* [11]. However, the logged operations or data are hard to associate to higher-level operations, i.e., it is hard to understand which parts of the data were tainted by the intrusion even when the malicious high level operations were identified. This is the second problem we aim to solve.

To solve these problems, in this work we present the **M**obile Applications **I**ntrusion **Re**covery **S**ervice (MIRES), an intrusion recovery service for mobile applications based on BaaS. To solve the first problem – loss of operations issued after the most recent snapshot – MIRES logs all operations, and, when an intrusion occurs, it does *rollback* of the state of the system and *selective re-execution* of the transactions, omitting those that were malicious. To solve the second problem – difficulty of associating logged data/operations to higher-level operations – MIRES stores both types of data/operations/events and causality information, allowing the identification of which log entries have to be undone. This intrusion recovery approach is based on recent research that considers different contexts but *not mobile applications or the BaaS model*: email services [12], web applications [13],

INESC-ID, Instituto Superior Técnico, Universidade de Lisboa – Lisboa, Portugal. E-mail:diogo.vaz@tecnico.ulisboa.pt

[1] More examples at https://firebase.google.com

[14], databases [15], [16], operating systems [17], and cloud computing [18], [19].

MIRES has a set of interesting *features*. The recovery is done online, mostly in parallel with the normal functioning of the application, to increase availability, on the contrary of what offline recovery mechanisms do. Moreover, although most applications still use a single backend data store (e.g., a database), many recent applications are starting to use more than one (e.g., a database and a file store), so MIRES supports multi-service recovery. Finally, MIRES also supports a form of client-side recovery in the mobile device to allow users to recover from mistakes.

The *security objectives* [20] of the intrusion recovery approach for mobile applications is to regain *integrity* after an intrusion and to do so with low impact on *availability*; the objective is *not* to achieve *confidentiality* as MIRES operates after the intrusion has happened. Confidentiality protection requires run-time mechanisms that are out of the scope of this paper [21], [22]. Our work also does *not* focus on intrusion detection, that is orthogonal to intrusion recovery; many existing and research mechanisms can be used for this purpose [23]–[25].

The design of MIRES had to deal with a set of *challenges*. First, with mobile applications it is not practical to have a proxy intermediating all client-server communications, unlike what is done in previous works. Second, the BaaS model supports multiple services in the backend, unlike previous works that support a single one (e.g., a database or a file system), so the recovery service has to support multi-service recovery. Third, MIRES is constrained by an environment that is not fully programmable, in the sense that is limited to the services and degrees of freedom provided by the BaaS model, especially in the backend.

We implemented a prototype of MIRES for the Android and the Firebase [26] platform – this implementation is available online[2] – and evaluated it experimentally using 4 mobile applications: a social network, a messaging app, a shopping list app and a contact tracing app. MIRES was able to recover 1000 operations in less than 1 minute, leaving the mobile application unavailable just for less than 15 seconds.

The main contribution of this paper is the first intrusion recovery service for the BaaS model, focused on mobile applications. Moreover, this service (1) provides online recovery to increase the availability of the system, (2) supports multi-service recovery; and (3) includes a client-side recovery mechanism.

II. Background and Related Work

This section provides some background on intrusion recovery and presents related work.

A. From Intrusions to Recovery

The term *intrusion* is often used to designate *unauthorized* activities that affect the *integrity*, *confidentiality* and/or *availability* of a system [20]. However, in this work we use the term in a broader sense to include also *authorized* but erroneous activities that someone later wants to undo. As already explained, the objective is to restore integrity after an intrusion with low impact on availability, but does not include confidentiality.

The process of dealing with an intrusion is divided into three main phases: intrusion detection, vulnerability remediation and 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 resort to *Intrusion Detection Systems* (IDS) [11], [23]–[25] to help in analyzing and detecting suspicious events. Some of these systems need human configuration to deal with precision issues, like false positives, to initiate the recovery process.

The second phase, *vulnerability remediation*, is concerned with fixing the application or its configuration, if needed. This phase consists of *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 [27], [28]. The goal is to prevent similar intrusions from happening again.

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

Selective Re-execution is based on changing the state of the system up to some point back in the past. This includes removing all activity, desirable and undesirable. For example, row versioning and snapshots [11] are two techniques that follow this approach. Then, the system re-executes all the requests not related to the intrusion, a process called *roll forward*, in order to bring the system to the present. MIRES adopts this technique, selective re-execution, to reconstruct the documents affected by the intrusions.

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. For example, compensating transactions [29] is a technique that implements this approach in the context of databases.

B. Related Work

Intrusion recovery has been studied considering different systems or services: email servers [12], [30], databases [15], [16], [31], hypervisor-based virtual machines [32]–[34], files and file systems [17], [35]–[40], web applications [13], [14], [41], and cloud computing service models [18], [19], [42]. This section introduces some relevant intrusion recovery works and briefly discusses their relationship with MIRES.

The first presentation of the broad intrusion recovery approach we follow is due to Brown and Patterson [12]. They present *Undo for Operators*, a tool that allows operators to

[2]https://github.com/diogolvaz/MIRES

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 the recovery mechanism it presents, which eventually was influential in a related mechanism used in Google's GMail. The model for Operator Undo is based on 3 events, introduced in the paper as the *three R's*: *Rewind*, where all the state of the system is 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. MIRES performs recovery in a similar way. First, it reverts the affected records to a previous point in time, then it reconstructs them by re-executing the operations that were logged. MIRES, unlike Operator Undo, does not rollback the entire state of the application, instead it locks the database so that the users can only perform read operations. This allows the application to remain available during recovery.

Another service, Warp [14], assists users and administrators of web applications to recover from intrusions while preserving legitimate user changes. The Warp recovery approach is based on rolling back a part of the database to a point in time prior to the intrusion and then apply compensating operations to correct the state of the database. MIRES presents two similarities with WARP: both require a client-side extension and both offer a recovery mechanism that partially recovers only the affected documents, instead of reconstructing the entire state of the database.

Shuttle [19] is an intrusion recovery service for Platform-as-a-Service (PaaS) applications. PaaS is a popular cloud model that provides an execution environment for web applications. Shuttle was designed to help administrators recover their applications from software flaws and malicious or accidentally corrupted user requests. MIRES and Shuttle present a similarity: both save read accesses to the database, in order to identify dependencies between transactions. However, Shuttle considers a different system model in which the applications are hosted in a PaaS and they hold their state in a SQL database, while MIRES aims to recover a BaaS, which can serve different types of applications.

Rectify and Sanare are two other intrusion recovery services for PaaS applications [18], [42]. They consider that the application is a black box, i.e., that it cannot be modified to implement the intrusion recovery service. Therefore, these services monitor HTTP requests and DB statements and find the relations between them without looking into the application code or requiring modifications to that code. The relations between HTTP requests and DB statements are derived using a supervised machine learning scheme in Rectify and a deep learning scheme in Sanare. The architecture of MIRES is based on Rectify's: in both cases the service is executed on a different container from the application container. However, MIRES is for BaaS-based mobile applications, which is not the case for the other two.

NoSQL Undo [16] is a recovery approach and tool that allows administrators to automatically remove the effect of intrusions, that is, of faulty operations done on data stored in NoSQL databases. To the best of our knowledge, this is the only work that focuses on recovering NoSQL databases. The MIRES reconstruction phase is based on one of the algorithms provided by NoSQL Undo.

TABLE I
COMPARISON OF INTRUSION RECOVERY WORKS.

System	Approach	Type	Target	User Recov.
[classical backups]	Rollback, loosing all updates	Offline	Files	✗
Undo for Operators [12]	Selective re-execution	Offline	Mail server	✗
Bezoar [33]	Selective re-execution	Offline	Virtual Mach.	✗
SHELF [34]	Rollback	Online	Virtual Mach.	✗
Taser [17]	Selective re-execution	Offline	File System	✗
Retro [17]	Selective re-execution	Offline	File System	✗
RockFS [43]	several	Online	Cloud Store	✓
Warp [14]	Selective re-execution	Offline	Web App	✓
Shuttle [19]	Selective re-execution	Semi-online	Web App	✗
Rectify [18]	Selective re-execution	Online	Web App	✗
Sanare [42]	Selective re-execution	Online	Web App	✗
NoSQL Undo [16]	Selective re-exec./Compens.trans.	Online	NoSQL DB	✗
MIRES	*Selective re-execution*	*Semi-online*	*Mobile app*	✓

There are some intrusion recovery mechanisms for recovering files from a file storage service. Examples are Retro, Taser, Back to the Future, RockFS and others [17], [35]–[40], [43]. Most of these works consider file systems [17], [35]–[40]. The exception is RockFS that considers files stored in a cloud store service or a cloud-of-clouds [43]. MIRES supports recovery when data is stored both in databases and file stores (multi-service recovery), but is closer to mechanisms for recovery of web applications backed by databases than to file system recovery mechanisms (Retro, Taser, etc.) or to cloud data storage services like RockFS.

There are a set of older works that are related to ours but have very different concerns and focus. An example is ITDB, a self-healing database that detects attacks and recovers from them [44]. Another example is a large literature on distributed checkpointing, which has the objective of obtaining a consistent view of the state of the (distributed) system to support recovery [45]–[47]. All these works are concerned with the recovery of low-level events (e.g., database operations), not with recovery of high-level events as we are in this work.

This paper extends a preliminary version of MIRES [48] in many ways: with a solution for multi-service recovery, more details about the service and a formalization of the dependencies that lead to recovery, an additional application, and a more extensive experimental evaluation.

Table I summarizes the main literature presented above. MIRES is based on some of the ideas shared in these works. However, *MIRES is the first that considers mobile applications and the BaaS cloud model*. Moreover, it introduces a online recovery process divided in two phases, which improves the availability of the system. MIRES also introduces a *multi-service recovery approach* and provides a new short-term recovery mechanism to mobile applications that allow users to recover their last action that is an enhancement welcome in most applications where end-users can commit mistakes.

III. THE BACKEND-AS-A-SERVICE MODEL

As already explained, Backend-as-a-Service (BaaS) [6]–[8], or the subcase of *Mobile Backend-as-a-Service* (MBaaS) [49], is a cloud service model that has been increasingly adopted to implement mobile apps.

A. The Model

The BaaS model provides a set of ready-to-use application logic services and features that automate and speed up the backend development process of mobile applications. BaaS platforms are also used to build web applications, but our focus is on mobile applications. BaaS aims to provide scalable and optimized backend infrastructures, outsourcing the responsibilities of running and maintaining these infrastructures to the BaaS vendor and leaving only the development of the mobile application to the user of the platform. This simplifies the job of the programmer, but is a challenge for the design of MIRES, as it cannot arbitrarily modify the application, unlike previous intrusion recovery schemes. There are many BaaS platforms today, e.g., Firebase, Back4App, Parse, AWS Amplify, Backendless, Kinvey, and Cloudboost.

The BaaS service model provides a set of common application services to the programmer, 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, e.g., diagnose crashes and performance of the application; and *cloud functions*, as in the serverless computing model, i.e., simple server-side code executed when invoked through HTTP endpoints or when certain events like database changes occur [50], [51].

A simplified architecture of a BaaS platform is represented in Figure 1. Each mobile application (e.g., A and B) running in a mobile device is connected with a virtual environment called a *container* [52]. 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.

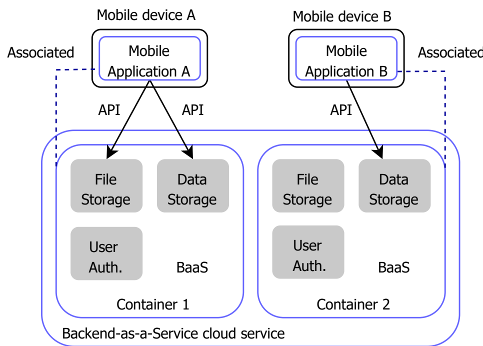


Fig. 1. Architecture of a BaaS service model.

B. Mobile Applications in BaaS

A *mobile application* is a piece of software that is executed on a mobile device, e.g., a smartphone or a tablet, often with a backend in some type of cloud or server(s) [3], [53].

The *state* of a mobile application that follows the BaaS model is divided between the mobile device and the cloud. This division is coordinated by executing remote services such as user management, file or data storage. When a user interacts with a mobile application, e.g., by clicking on a button, a set of operations are executed on the backend, reflecting the users' action.

The recovery mechanism we present in this work interacts mostly with the backend storage service(s), since we assume that these services reflect the state of the application. In the BaaS model, such storage services can be a relational database, a non-relational database [54], or a file store. Some BaaS platforms allow the integration of external databases (e.g., Back4App allows the integration of MongoDB [55]), while others provide their own database (e.g., Firebase provides Cloud Firestore[3]). MIRES does not depend on the specific file store, but the prototype uses a NoSQL database, Firebase's Cloud Firestore.

The term mobile application is ambiguous, so in the rest of the paper we use *mobile application* to mean the part of the application running on the mobile device and *mobile application system* to designate the entire system, i.e., both the parts of the mobile device and the backend.

IV. The MIRES Service

MIRES is focused on regaining the *integrity* of the state of mobile applications by *undoing* the effects of intrusions, i.e., of malicious actions. For a mobile application based on the BaaS model to benefit from MIRES, it has to be configured to use this service. The design of MIRES follows an important design principle: the changes made to the application logic should be minimal. The rational is that this simplifies setting up MIRES with an application.

A. Assumptions

We assume that the system is composed of mobile devices and a cloud, as well as of a mobile application system that runs both in the mobile devices and in the cloud, following essentially the architecture presented in Figure 1.

We use the term *system administrator* to designate the persons that manage the MIRES service and the term *user* to mean the persons who use the mobile application system that MIRES protects. We assume that the system administrator is trusted. We assume that the application has legitimate users that are also trusted. However, malicious users can do actions on the mobile application system that we may want to recover from. These actions can be done locally, by gaining access to the device, or remotely, using attacks such as SQL injection, cross-site scripting (XSS), or cross-site request forgery (CSRF) [9], [56].

Our main objective is to recover the *backend state* of the mobile application system, not the local state of mobile devices, for two reasons: first, this is more relevant and more challenging as the backend is accessed and modified by many different users, while the local state is only accessed in the mobile device; second, many applications already support mechanisms for local recovery, e.g., the restore from backup process supported by WhatsApp and the implicit recovery

[3]https://firebase.google.com/docs/firestore

done by many applications simply by logging out and in again. We assume that the cloud backend state is the authoritative state of the application, i.e., that mobile applications always use data based on the backend state. This is true for arguably all nontrivial mobile applications.

We consider that the state of the mobile application can only be corrupted by transactions originated by user actions. A transaction $t \in \mathcal{T}$ is composed by a set of operations $\mathcal{O}_t \subset \mathcal{O}$, where $\mathcal{T}$ and $\mathcal{O}$ are respectively the sets of all possible operations and transactions. In that sense, an intrusion occurs when a malicious action performed by a user explores a vulnerability on the mobile application, originating a *malicious transaction* t_m. A malicious transaction contains one or more malicious *operations* $\mathcal{O}_m$, and zero or more non-malicious operations $\mathcal{O}_{nm}$. However, when recovering a malicious transaction, all operations that depend on these have to be reverted, both malicious and non-malicious. Similarly to all intrusion recovery systems in the literature, e.g., [12]–[19], 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, but only that they are outside the scope of the solution presented in this paper. They are the subject of a large literature and even the work of organizations such as the Cloud Security Alliance (CSA).

When mentioning the storage service(s), we use the terminology of document-oriented NoSQL databases, without lack of generality. We do not assume such a database; we only use the corresponding terminology (alternative options would be using the terminology of SQL, file stores, key-value stores, or NoSQL columnar databases). This means that we refer to data items as *documents* ($\mathcal{D} = \{d_1, d_2, ...\}$), organized in *collections*. These documents may contain *properties*, i.e., keys associated with values.

This also means that the data stores support CRUD operations: *Create*, *Read*, *Update* and *Delete*, but we often summarize them in just two: *writes* that modify the content (i.e., creates, updates, and deletes) and *reads* that do not.

Modern databases support *transactions* that provide the ACID properties[4] allowing applications to perform writes – and in some cases reads – atomically in different documents. We assume that transactions are performed correctly and atomically, as our focus is not on recovering the inconsistent states of applications due to incomplete transactions. In other words, this work is not concerned with fixing broken applications, but with recovering from intrusions in correct applications.

B. Supported forms of Recovery

We consider two recovery scenarios:

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

[4]MongoDB transactions https://www.mongodb.com/transactions; Cloud Firestore transactions https://firebase.google.com/docs/firestore/manage-data/transactions

An interesting scenario happens when the user loses control of his device – and consequently of the application – during an interval of time, e.g., because the device was stolen but later recovered. This scenario is handled mainly 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.

C. Architecture

MIRES is composed of a set of resources running on the frontend (mobile application) and backend (BaaS platform). The BaaS platform provides a set of components and APIs that allow the communication between the mobile application and the BaaS services. These components and APIs are what simplifies the implementation of applications based on BaaS, but also place constraints on how MIRES interplays with the application and does its job. The architecture is represented in Figure 2.

In the Mobile application (top-right of the figure), on the mobile device, the *MIRES package* provides the framework needed to configure the mobile application.

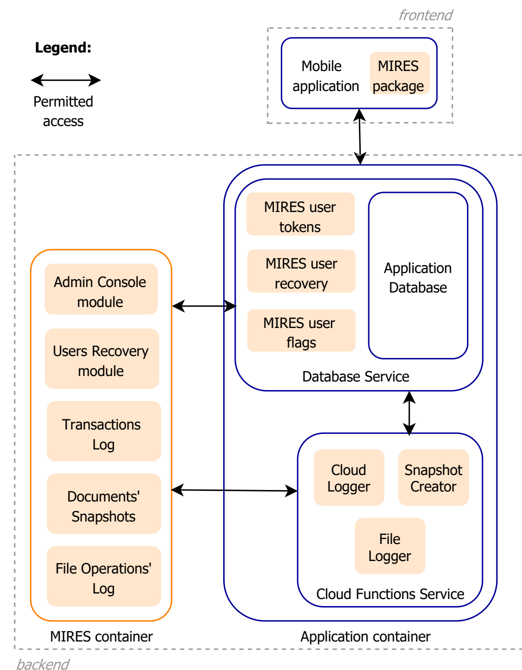


Fig. 2. MIRES architecture. The MIRES components are shown in *orange* and the resources of the mobile application system are shown in blue.

In the Application container (right of the figure), the Application Database takes three new *collections* (in relation to those used by the application itself): the *MIRES user tokens collection* to store the information that allows MIRES to communicate with the application (pairs of user identifier and token generated by the BaaS platform to identify the user's requests); the *MIRES user recovery collection* for storing data necessary for undoing mistakes done by users (see Section VIII); and the *MIRES user flags collection* for storing data

(flags) needed for tracing the mobile application normal execution (Section V). Additionally, three types of *cloud functions* are executed in the Cloud Functions Service (bottom): the *Cloud Loggers* for logging all the requests made to a database and updating the *Transactions Log*; the *Snapshot Creators* for creating snapshots of the database documents, stored on the *Documents' Snapshots*; and the *File Loggers* for logging all the file operations made into a file storage service and updating the *File Operations' Log* (Section VII).

On the MIRES container (left of the figure), there are two modules: the *Admin Console module*, that allows the system administrator to interact with the MIRES service and recover intrusions; and the *Users Recovery module*, responsible for the user recovery mechanism (see Section VIII).

The MIRES architecture provides intrusion recovery without placing a proxy between the mobile device and the backend, on the contrary of what happens in many related systems [12]–[14], [18], [19], [41]. This is important because it avoids creating a performance bottleneck (see, for example, the negative impact of Shuttle's proxy in [19]) and allows preserving the functional and security properties provided by the BaaS service API.

V. Normal Execution

During normal execution, MIRES captures data of each transaction performed in the BaaS that, later, can be used in a recovery process. Figure 3(a) shows the normal function of a mobile application system when using MIRES. This section explains the steps performed to handle a transaction during the normal execution of MIRES.

In this section, for simplicity, we consider that the application is backed by a single data store: a database. In Section VII we generalize for the case of more, and other, data stores.

A. Mobile Application Configuration

Actions done by the user in the mobile application may cause the execution of transactions in the backend database (operation *R1* and *R2* in Figure 3(a)).

MIRES configures each *write* operation to carry additional properties associated with the document that is being written: an *operation ID* representing the operation itself; a *locked* property used in the recovery process (see Sections VI and VIII-B); and an *ignore* property, used by MIRES to perform requests on database documents without activating the *Cloud Loggers*. 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 a message containing a *flag*.

For *read* operations, MIRES configure the application to forbid reads on *locked* or *blocked* documents, i.e., during recovery (details later).

B. Logging Write Operations

For each CRUD operation that modifies the state of the database, MIRES logs the *type of the operation*, the associated *timestamp*, the *document changed*, the *data associated with the operation*, a *transaction ID*, that associates all requests of the same transaction (e.g., *R1* and *R2* in the figure) and the *operation ID* generated by the MIRES package (see Section V-A).

MIRES logs each operation executed by the backend by resorting to two mechanisms: *Flags* and *Cloud Loggers*. Flags is additional information about operations, i.e., information about operations that is not sent automatically by the BaaS platform (Section V-B1). Cloud Loggers are cloud functions that log the requests made by the mobile applications (Section V-B2). We now explain how Flags and Cloud Loggers allow MIRES to log each operation made to the database.

1) Flags: For each write operation made in the database (arrow $\vec{a}$ in Figure 3(a)), the mobile application sends an additional request (arrow $\vec{b}$) with what we call a *Flag*.

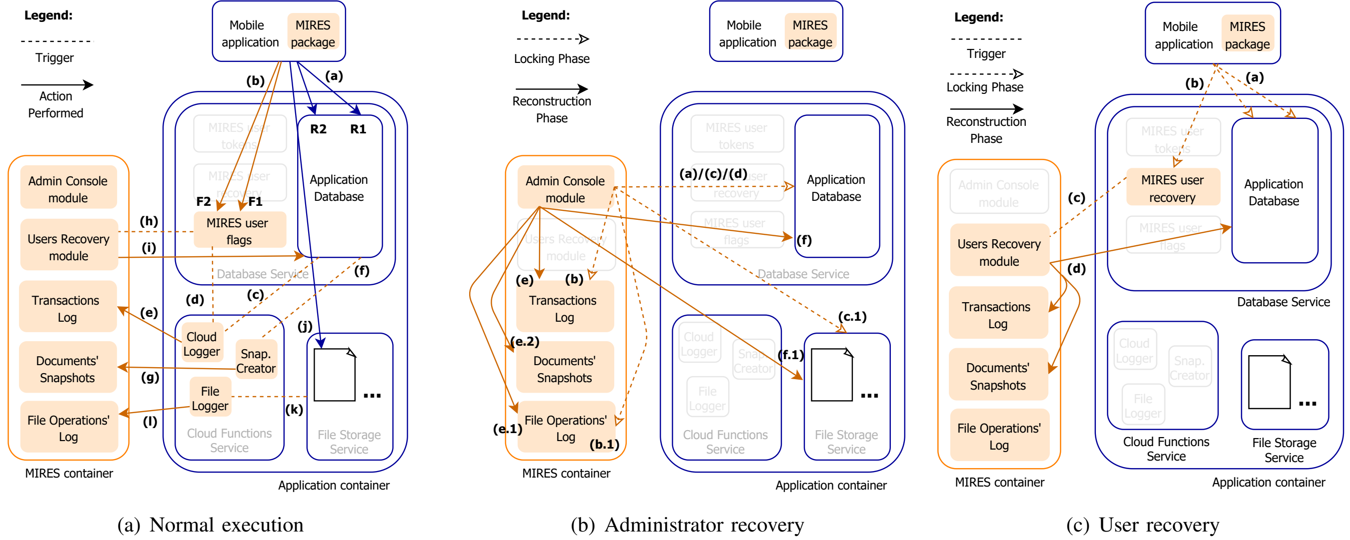
Flags have two roles. The first role is to send to the backend additional information – in relation to the information sent by the BaaS platform – that MIRES needs to log the operation, more precisely, the *transaction ID*, the *timestamp* and the *data structure* of the operation. Notice that it would be possible to change the application requests to take this additional information. However, this would violate the design principle of minimizing the changes to the application logic. Moreover, it might create problems with the specificities of some BaaS platforms (e.g., Firebase supports at most 100 fields in messages, so adding fields for MIRES might be incompatible with some applications). Moreover, an operation that writes data can create or update a document, depending on if the document exists or not; instead, an operation that deletes data is necessarily a delete in the database. Flags are used to disambiguate these cases.

The second role of Flags is to indicate when it is possible to start the recovery process. As Cloud Functions are executed in containers, the first execution of such a function experiences the delay associated to the launch of the container, a phenomenon known as *cold start* [57]. Due to cold starts, on rare occasions Cloud Loggers 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 case, since each flag represents an operation made to the database, the recovery process can only begin when all flags are processed and the flags resource of the MIRES users is empty.

Flags are atomic in relation to the operation they are associated to. This means that each flag is stored in the MIRES user flags collection only if the associate operation is also performed.

2) Cloud Loggers: Cloud Loggers are cloud functions [50], [51] that are activated by two types of events: create/update operations on the Application Database (arrow $\vec{c}$ in Figure 3(a)), and delete operation flags on the MIRES user flags collection (arrow $\vec{d}$).

When a create/update operation is performed, a Cloud Logger is activated. Then, the Cloud Logger accesses the MIRES user flags collection to get the flag associated with the operation. Then, it obtains other information needed to log the operation: the *type* and *data* handled by the operation.



(a) Normal execution (b) Administrator recovery (c) User recovery

Fig. 3. MIRES architecture and interactions. The MIRES components are in *orange* and the components of the mobile application system are in blue.

Obtaining this information may be simple or not. In the case of the Firebase BaaS, the data modified by an operation is not stored explicitly, so it has to be inferred. Specifically, to gather the data written by the operation, the Cloud Logger compares the document before and after the operation. There is a challenge: this solution does not allow obtaining the data written by an update in case it is equal to the content of the document (e.g., updating to 1 a field that already contains 1). For that reason, MIRES sends to the backend a flag with the operations' data structure, i.e., with the properties of the document changed by the operation. Interestingly, this process has an advantage: Cloud Loggers can capture the *direct and indirect effects* of an operation. For example, an update that replaces the entire document by new data produces a direct effect; on the contrary, data overwritten in a document without this being explicitly indicated is an indirect effect. By having access to the versions of the document before and after the operations execution, Cloud Loggers can compare both versions and log the described effects, which allows rebuilding the documents independently from the type of update made.

In delete operations, the logging process is performed differently: when a new flag is added to the MIRES user flags collection, a Cloud Logger instance is activated in order to access the flag and see if it is a delete operation flag. This is the only case, in which a delete operation is directly logged. 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 $\vec{e}$) and deletes the flag from the MIRES user flags collection.

The system administrator can use the *Admin Console* to access the log of operations and recover the state of the application. Arrows $\vec{f}$, $\vec{g}$ relate to snapshots and are explained in Section VI-B2; arrows $\vec{h}$, $\vec{i}$ relate to user recovery and are explained in Section VIII-A; and arrows $\vec{j}$, $\vec{k}$ and $\vec{l}$ relate to multi-service recovery, explained in Section VII.

C. Logging Read Operations

The *state* of the mobile application system is changed by the *write* operations executed on the database, so they must be logged, as explained in the previous section. Instead, *reads* do not modify the state, so they do not have to be logged for that reason. However, some reads do have to be logged for the following reason: the data carried on a create or an update (two types of *writes*) may come directly from the user or may have been taken from the database itself. In the latter case, the data comes from one or more read operations, so there is a dependency between the write and the read(s).

Therefore, MIRES logs the read operations that create *dependencies* between transactions (see Section VI-A). To achieve this, the MIRES package in the client-side is used to make the mobile application send information about the read operation. Thereby, this 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 keeps track of which *version* of the document was accessed; each write operation performed creates a new version of the 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 explicit in the mobile application code. BaaS platforms can provide *native function calls* that remove the necessity of performing a read operation, e.g., Firebase's `incrementValue()` call, where there is a dependency on

the value incremented. In this occasion, the dependency exists, so it is necessary to configure these special cases, e.g, by adding a new property to the normal operation to be analysed by the Cloud Logger in these scenarios. Since Cloud Loggers have access to the version of the document before the operation is executed, these components can log the read dependency, increasing the recovery accuracy of MIRES.

VI. Administrator Recovery

The approach followed by MIRES is to directly remove the *intrusions* and their *effects* by rolling back the state of the application and doing *selective re-execution* of the transactions, i.e., re-executing those that are safe, but not those that are malicious. Figure 3(b) presents the recovery process implemented by MIRES. This process is divided in two phases: (1) the *locking phase*, responsible for identifying the malicious and compromised transactions; (2) and the *reconstruction phase*, responsible for reconstructing the affected documents. This section explains both.

A. Phase 1: Locking Phase

A system administrator initiates the recovery process when he decides to do so, after becoming aware of an intrusion. As previously explained, detection is orthogonal to recovery and out of scope of the paper. We assume that the intrusion is detected in some way[5] and that the set of malicious operations $\mathcal{O}_m = \{o_1, ..., o_n\}$ directly compromised by the actions of the attacker, and malicious transactions $\mathcal{T}_m = \{t_1, ..., t_{n'}\}$ that include these operations, are identified. For example, if the attacker only pressed a button that caused the operation $o_m(d)$ that deleted a document d belonging to a user, then $\mathcal{O}_m = \{o_m\}$ and $\mathcal{T}_m = \{t_m\}$, where $o_m \in t_m$. Needless to say, the recovery process will have to remove the direct effects of these *malicious operations* from the state of the system. However, it will also have to remove their indirect effects, i.e., the effects of operations that were tainted by the former. We call *tainted operations* both the later and the former, i.e., malicious operations are also tainted. Deducing which operations are tainted is an important part of the problem.

The system administrator activates the MIRES recovery mechanism by using the *Admin Console*, where he selects the operations that have to be undone, i.e., the operations in the set $\mathcal{O}_m$. The console also allows sending a personalized message to the online mobile applications, e.g., to explain to the users the reason behind the recovery process. MIRES messages are received by the application and shown to the user through notifications.

When the recovery is initiated, the locking phase begins: MIRES locks the database (arrow $\vec{a}$ in Figure 3(b)), forbidding writes but allowing reads. Then, MIRES analyzes the log since the moment the first malicious transaction occurred (arrow $\vec{b}$), in order to identify *dependencies* between later transactions and, consequently, identify and *lock* all the affected documents (arrow $\vec{c}$), i.e., all documents where both read and write are forbidden. MIRES locks a document by changing the *locked* property of the document to *true*.

[5]Two options would be: (1) to have a host-based IDS that analyzed the logs created by MIRES and alerted the administrator for possible intrusions; or (2) to have a network-based IDS like Snort configured to detect attacks and activate recovery, similarly to what was done in NoSQL Undo [16].

During the locking phase, MIRES analyses the log in order to identify *dependencies* between operations. MIRES implements this process by simulating the propagation of corrupted data in memory and comparing the operations made on the database with the tainted data, to identify operations that were later tainted. From this analysis, two possible dependencies can result: *read-write dependencies* and *structural dependencies*.

We define these dependencies using Lamport's *happened before* relation, which expresses the order in which two operations were executed [58]. This relation, written $\rightarrow$, satisfies three properties: 1) If a and b are events in the same process and a came before b, then $a \rightarrow b$; 2) If a is the event of sending a message and b is the event of receiving that message in another process, then $a \rightarrow b$; 3) If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$ (transitivity). In our case we leverage the first and third properties, as the events that matter to us are local operations in the database.

A dependency is also a relation. We write that operation O_2 *depends on* operation O_1 in the following way: $O_1 \rightsquigarrow O_2$. This relation also satisfies transitivity: if $a \rightsquigarrow b$ and $b \rightsquigarrow c$, then $a \rightsquigarrow c$.

Notice that if $O_1 \rightsquigarrow O_2$ then $O_1 \rightarrow O_2$, but the contrary is not true: $O_3 \rightarrow O_4$ does not imply that $O_3 \rightsquigarrow O_4$ because the two operations may be unrelated. The operations $C(d)$, $R(d)$, $U(d, v)$, and $D(d)$, correspond to the four CRUD operations, and respectively create the document d, read its contents (or part of it), update its content (or part of it) with data v, and delete it.

1) Read-Write Dependencies: When the data written by a transaction is based on data retrieved by a previous read request to the database, there is a *read-write dependency*. For this reason, when an intrusion occurs, read operations can propagate the effects of the intrusion by reading corrupted data on the database and, consequently, generating new corrupted data. Rigorously, for an update U we have (where $\Rightarrow$ means "implies"):

$$R(d) \rightarrow U(d', v) \;\Rightarrow\; R(d) \rightsquigarrow U(d', v) \tag{1}$$

This means that if d is tainted, then d' is also tainted. It also means that MIRES can revert the spread of tainted data based on the data gathered about read operations during normal execution (see Section V-C). When analysing a write operation influenced by a read operation, MIRES compares either the full document or a part of of it (e.g., values of properties in the case of a document database), depending on the implementation and the level of granularity desired.

We have similar dependencies for the other writes, i.e., for creates and deletes:

$$R(d) \rightarrow C(d') \;\Rightarrow\; R(d) \rightsquigarrow C(d') \tag{2}$$

$$R(d) \rightarrow D(d') \;\Rightarrow\; R(d) \rightsquigarrow D(d') \tag{3}$$

2) Structural Dependencies: Besides read operations, write operations can also create relations between transactions, which we designate *structural dependencies*. They can occur in two scenarios: when a write operation is performed on a document that should not exist; or when a document that should already exist is (re)created.

The first scenario happens when a malicious transaction creates a new document, then all subsequent operations on that document depend on the first. During recovery, these operations have all to be undone until the document is finally deleted. Rigorously:

$$C(d) \rightarrow U(d,v) \;\Rightarrow\; C(d) \rightsquigarrow U(d,v) \tag{4}$$

The second scenario happens when a malicious transaction deletes a document, then a create operation creates the document again. In the recovery, the second operation has to be undone, since the document should already exist. Rigorously, the dependency is:

$$D(d) \rightarrow C(d) \;\Rightarrow\; D(d) \rightsquigarrow C(d) \tag{5}$$

B. Phase 2: Reconstruction Phase

After executing the locking phase, MIRES has to undo the effects of the tainted operations. Given the set of malicious operations $\mathcal{O}_m$ previously defined, the set of tainted operations is given by:

$$\mathcal{O}_t = \{o_t : o_t \in \mathcal{O}_m \vee \exists o_m \in \mathcal{O}_m : o_m \rightsquigarrow o_t\}$$

Given the set of tainted operations, the set of *tainted transactions* is given by:

$$\mathcal{T}_t = \{t_t : \exists o_t \in \mathcal{O}_t : o_t \in t_t\}$$

After executing the locking phase, MIRES knows all the tainted operations $\mathcal{O}_t$ and tainted transactions $\mathcal{T}_t$, and has the documents involved locked in the database. Then, MIRES unlocks the database (arrow $\vec{d}$ in Figure 3(b)), allowing users to interact again with the backend. With the locking phase finished, MIRES starts to reconstruct the affected documents (arrows $\vec{e}$ and $\vec{f}$); the corrupted documents are locked, so users can normally interact with unaffected documents within the database while MIRES reconstructs the corrupted documents.

1) Operations Model: The reconstruction of documents in the database is inspired by the *Focused Recovery* algorithm of [16]. This algorithm reconstructs the affected documents by replaying all write operations that affected them, except the malicious operations.

Rigorously, the process consists in re-executing, in the order defined by the *happened before* relation, all operations in the following set:

$$\mathcal{O}_r : \{o : \forall t \in \mathcal{T}_t, o \in t \wedge o \notin \mathcal{O}_m \wedge o \notin \mathcal{R}\}$$

The set $\mathcal{R}$ above represents all read operations executed in the system. It is an artefact to simplify the definition and never actually exists in the system before, during, or after recovery.

However, this reconstruction model presents a drawback: the time to reconstruct the document grows with the number of *versions* of a document and, consequently, the number of operations to re-execute. In MIRES, this phase is performed concurrently with user interactions with the backend, so the system continues to operate; only the infected documents are temporarily unavailable.

2) Snapshots Model: The operations model of the previous section can be improved using *snapshots* [59] (arrow $\vec{e.2}$), 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 free of corruption. 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 V-A – 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 $\vec{f}$ of Figure 3(a)) and stores a document snapshot after *N* operations made to the document (arrow $\vec{g}$ of Figure 3(a)), e.g., store a version of a document after each 1000 or 10000 operations made. This is a *non-blocking* process, i.e., the mobile application system is not stopped during the creation of a snapshot.

VII. Multi-Service Recovery

The state of a web or a mobile application is generally stored in a single storage service, usually a database. As a consequence, work on web application recovery so far has focused on recovering state stored in a single database [13], [14], [16], [18], [19] (there is no previous work on the recovery of mobile applications). However, the complexity of mobile (and web) applications is increasing and they are starting to store their state in more that one data store. An example is the action of posting a picture in a social network application, where the image itself is stored on the file storage service and additional data about the post, e.g., the username of the poster, the timestamp and the picture URL is stored in the database service.

For recovering these actions involving several backend data stores – i.e., for *multi-service recovery* –, it is necessary to recover both the database and the file storage services in a synchronized way. We do not make assumptions about the existence of support for transactions involving both database and file store operations, i.e., about the atomic execution of operations in more than one data storage service. This is the case in most BaaS platforms and an important challenge we have to face in the solution.

Next we explain what changes in relation to single-service recovery (presented before) in normal execution (Section VII-A) and in administrator recovery (Section VII-A). We assume there is a database and mostly explain how to add a file store. We present multi-service recovery as if there was one database and one file store without lack of generality, i.e., without limiting the number of databases and file storage services.

A. Normal Execution

In normal execution (Figure 3(a)), in terms of the file store, mobile applications can create, read, update, and delete files. We group the write operations in two types: *upload* operations that create and update files, and *delete* operations that remove them.

When the mobile application performs an upload file operation (arrow $\vec{j}$ of Figure 3(a)), MIRES configures the *file metadata* by adding: the *transaction ID* of the transaction, allowing the File Logger to log the file operation associated with a specific transaction; an *ignore* property, when MIRES wants to recover the file without activating the logging process; and a *locked* property, to lock the file. This metadata, specifically the transaction ID, is the main mechanism to circumvent the lack of atomic transactions involving both database and file operations. However, there are cases in which this mechanism fails. For example, in Firebase, *delete* operations take as a single parameter a reference to the file, not allowing to send any additional information to the backend. Moreover, a solution based on flags, i.e., on sending a flag with additional information about the operation, would not work as it is not possible to achieve atomicity, i.e., the file flag could be created without the file operation being executed, or vice versa. For these reasons, in the case of Firebase, it is not possible to correlate database operations with file delete operations, only with file upload operations.

Then, the File Logger captures the file change (arrow $\vec{k}$) and stores on the *File Operations' Log* (arrow $\vec{l}$) the information needed to recover the file: transaction ID, file path, timestamp, operation type (delete or upload), and *file generation ID* (version of the file).

As discussed in Section V-B1, this logging process has a problem: cold starts may lead File Loggers to take some time to activate and log the file operations. This can lead to MIRES initiating the recovery process without the log being stable. Flags cannot solve this problem due to lack of atomicity, as refered above, but there are workarounds to mitigate this scenarios, e.g, waiting for a short period of time before the recovery is initiated.[6]

B. Administrator Recovery

When the administrator identifies an intrusion, the recovery process is initiated as explained in Section VI. However, in this case, both the database service and the file storage service are recovered. Figure 3(b) represents the process of recovering multi-service transactions.

1) Locking Phase: When the locking phase begins, the database service(s) is(are) locked, but not the file storage service(s). The reason for this difference is that we opted not to track *read-write dependencies* involving files, on the contrary to what we do with database data. The rationale for this decision is that, in mobile apps, databases often store data used in the logic of the application, whereas file stores are used to keep data that is stored, retrieved, and presented, but not used to take decisions. This means that data read from the database often affects data subsequently written in the database, whereas the same does not happen with files. Therefore, there is usually no dependency and there is no point in tracking it.

Not locking the file system is beneficial in terms of application availability. However, it also raises two issues: broken transactions, since applications can write on the file storage service when the database service is locked; and normal file operations, since users can continue to interact normally with the file storage service through the application. Nevertheless, we assume that mobile applications can deal with broken transactions and, since we do not analyse dependencies on file operations, users can interact normally with the file storage service until the moment that files are locked specifically for being recovered (only the files, not the whole file store).

After the locking takes place, MIRES analyses the database log to search for the malicious transactions $\mathcal{T}_m$ (cf. Section VI-A) and the affected database documents. After locking the affected documents, MIRES analyses the File Operations Log in order to know if that malicious transaction interacted with the file storage (arrow $\vec{b.1}$ in Figure 3(b)). If so, MIRES locks the infected file (but not the whole file storage service), similarly to what it does with database documents (arrow $\vec{c.1}$). Similarly to the process of database documents, files need to be locked, forbidding any upload action on the file, for MIRES being able to analyse the operations performed and rebuild the file, during the normal execution of users.

2) Reconstruction Phase: With the Locking Phase finished, additionally to the malicious transactions and the infected documents known by MIRES, the service also knows the infected files. Both database documents and files are locked on their corresponding storage service. Finally, the Reconstruction Phase is initiated, in which MIRES reconstructs both locked database documents (see Section VI-B) and locked files. To reconstruct the files, MIRES analyses the File Operations Log to find the correct version of the file (arrow $\vec{e.1}$ in Figure 3(b)) and uses the *generation ID* to retrieve that version from the cloud service (arrow $\vec{f.1}$).

VIII. User Recovery

The main goal of MIRES is to support the recovery from intrusions that have an impact on the main state of mobile application systems: the data stored in the backend data storage services. This was what was presented in the previous sections. However, MIRES also provides a client-side mechanism to allow users to recover from their last action, the topic of this section.

A. Normal Execution

In normal execution, to provide the user recovery mechanism, each operation requires an additional configuration, supported by the MIRES package and similar to those mentioned previously (Sections V-A and VI-B2). Each write operation is configured to carry extra data: a *blocked* property, used to generate *blocked* documents (blocked documents are invisible to the users, i.e., reads are forbidden except for the user that

[6]https://cloud.google.com/functions/docs/bestpractices/tips

TABLE II
MIRES IMPLEMENTATION LINES OF CODE (LOCS).

Client-side	LoCs	Server-side	LoCs
Tokens	47	Cloud Logger (flags)	26
Notifications	52	Cloud Logger (collection)	63
Transaction configuration	181	File Logger	51
Undo recovery mechanism	198	Snapshot Creator	122
		Users Recovery module	558
		Admin Console module	1119

performed the last write on the document); and a *user ID*, representing the user that performed the transaction.

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., lead MIRES to set the *blocked* property in those documents to true. This is not permanent, but for the interval of time T_u during which user recovery is possible ($T_u = 30$ sec. in the experiments).

When a transaction that can be recovered finalizes, MIRES generates a notification with a button that allows undoing the last operation. This option becomes inactive after the time interval T_u, or when the mobile application performs another transaction. While the option is active, the user can use it to activate the recovery.

On the backend, the Users Recovery module is listening for operation flags (arrow $\vec{h}$ of Figure 3(a)). When a flag of a blocked document arrives, the Users Recovery module will unblock the document after a time interval T_u, i.e., it will change the *blocked* property back to false (arrow $\vec{i}$ in Figure 3(a)).

B. Recovery Execution

The users recovery flow is represented in Figure 3(c). When the user clicks on the undo button, the MIRES package *locks* the documents (arrow $\vec{a}$). More, it sends a recovery request to the Users Recovery module with the transaction ID to be recovered and the documents locked (arrow $\vec{b}$). Both the lock of the documents and the sending of the recovery request are made atomically, whereas the recovery request is only sent if all the documents are locked. Afterwards, the User Recovery module reads the recovery request from the user (arrow $\vec{c}$) and reconstructs the documents affected in a way similar to the reconstruction phase of Section VI-B (arrow $\vec{d}$).

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

IX. MIRES IMPLEMENTATION

The MIRES client-side package allows the configuration of the mobile application to manage: the MIRES notifications, the locking phase of the user recovery mechanism and the configuration performed on each operation, where in the beginning of each transaction, MIRES creates a transaction state used by the mobile application code to configure each operation of the transaction. This client-side resource was implemented in Java, which is the default programming language for Android applications. The number of lines of code of the implementation are shown in Table II.

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

Cloud Loggers, Snapshot Creators and File Loggers were implemented as JavaScript code deployed on the mobile application container using the Cloud Functions service. Cloud Function scripts listen for specific pre-defined collections, which assures configuration *flexibility* over the database that we want to protect. For example, it is only required to deploy a Cloud Logger, a Snapshot Create or a File Logger that listens for a collection containing the data that we want to protect and configure the actions of the mobile application that interact with that same collection.

The BaaS platform used was Firebase. We used the Firestore database service to store the log of transactions, the log of file operations and the snapshots. With this service, and the Cloud Functions mechanism, we can assure *automatic scaling* on the creation of logs and snapshots. Moreover, 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 on the mobile application container: this allows reusing the security rules and settings that allow only the authenticated users to interact with the three collections. Moreover, it is possible to define specific security rules, allowing 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, and also on the locking phase of the user recovery mechanism, allowing us to mitigate possible synchronization problems.

A. Example Applications

To evaluate the MIRES service, we used four open-source Android applications:

1) a *social network application*, Hify [7], where users can post images or text, comment and like other posts;
2) a *messaging application* [8] for 1 to 1 conversations;

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

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

TABLE III
APPLICATIONS AND THEIR CONFIGURED ACTIONS USED TO TEST MIRES.

Applications	Configured Actions
Social Network	Create Account Post, Comment and Like
Messaging	Create Account and Chats Create, Change and Delete messages
Shopping Lists	Create Accounts Add, Change and Delete lists and products
Contact Tracking	Create account Store user's contacts

3) a *shopping list application*, ShoppingListApp [9], to create shopping lists, by adding, changing and removing products;
4) a *contact tracking application*, CovSense [10], used to track contacts between their users and manage the COVID-19 spread.

The social network and messaging applications were chosen based on the high number of dependencies of transactions of different users created on the backend, and because they represent the logic of 4 out of 5 of the most downloaded apps of 2019 [60]. The shopping list was chosen due to its different logic, since it is a personal application, not social. The CovSense application was chosen due to the actual pandemic context that humankind is facing. All applications use Firebase as BaaS and the Firestore database.

Table III provides information about each application and the actions used to test our recovery service. We now 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: photo-sharing (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 and also help others with answers for their questions; 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 user accounts and their main actions on the application, more specifically, posts, comments and likes.

Chat is a 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 inserting the username – and start a conversation, with the possibility to send images or text messages. We have focused our experiments on recovering data related to user accounts, chats, and messages.

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

CovSense is an application for tracking the spread of COVID-19. The application uses a combination of Wi-Fi, Bluetooth, BluetoothLE and ultrasonic modem to communicate a unique-in-time pairing code between devices. This code can be used to trace contacts between users. Application actions are: create account, update the health status – between "Healthy" and "Diagnosed with COVID-19" – and store contacts with other users, that is done automatically by the application. After a 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 user accounts and the contacts stored between different users.

X. EXPERIMENTAL EVALUATION

This section presents the experimental evaluation of MIRES with the four example applications described above. With this evaluation we wanted to answer the following questions: %beginitemize What is the performance penalty and cost of MIRES in run-time, as it requires logging all operations? – Section X-A; What is the storage space overhead of running MIRES? – Section X-B; What is the performance of the administrator recovery? – Section X-C; What is the performance of the multi-service recovery process? – Section X-D; What is the performance of the user recovery? – Section X-E;

We used 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 to test each application. Both application and MIRES containers were deployed on Google Cloud in the same region (`europe-west2`) to mitigate 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 5 executions of the results obtained with the 4 applications, except when noticed.

A. Logging Performance and Cost

This section evaluates the performance overhead of logging the operations done in normal execution mode.

1) Mobile Application Performance: MIRES configures both write and read operations made to the database, resulting in an additional cost to execute each operation. To test the imposed cost, we used the actions of the application and performed a set of 1K CRUD operations per application. Each block of operations followed a different workflow distribution for each application: 80/20 read/write distribution for the social network application, where users tend to actively read posts from other users, 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

[9]https://github.com/alexmamo/Firestore-ShoppingListApp version of 07/07/2020
[10]https://github.com/saivittalb/covsense version of 27/09/2020

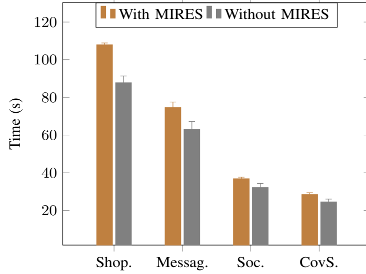


Fig. 4. Time to perform 1000 operations on each application, with and without MIRES.

TABLE IV
INTRUSION RECOVERY SERVICE OVERHEADS (VALUES TAKEN FROM THE CORRESPONDING PAPERS).

System - Application	Overhead
MIRES - Shopping List App	23.00%
MIRES - Messaging App	18.00%
MIRES - CovSense App	16.00%
MIRES - Social Network App	15.00%
Rectify - Wordpress	17.19%
Rectify - LimeSurvey	18.72%
Rectify - MediaWiki	14.49%
RockFS - Async	14.19%
RockFS - Sync	15.88%
Shuttle - Workload A	13.00%
Shuttle - Workload B	16.00%
Warp - Reading	24.00%
Warp - Editing	26.84%
Retro - Web server (1 core)	55.99%
Retro - Web server (2 cores)	24.89%
Retro - HotCRP (1 core)	25.98%
Retro - HotCRP (2 cores)	1.96%
SHELF - Dbench throughput	7.60%
SHELF - Apache ab transfer rate	7.50%

application since most of the application main logic is based on writes.

Figure 4 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 is due to the different configurations made 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. Although the overhead is noteworthy, MIRES presents a similar performance degradation when compared to other intrusion recovery works (see Table IV) which we consider acceptable, given the recovery 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).

2) Mobile Application Performance with File Metadata Configuration: To test the performance of multi-service transactions, we used the Hify application, through image posts[11]. Image posts transactions are composed of 2 operations: a

TABLE V
TIME TO PERFORM A DATABASE-FILE TRANSACTION, WITH AND WITHOUT MIRES.

Action	Time (s)
Post action without file configuration	0.92 +- 0.02
Post action with file configuration	0.96 +- 0.03

TABLE VI
COST OF THE CLOUD FUNCTION EXECUTIONS USED BY MIRES

Cloud Function	Time (s)	Memory (GB)
Cloud Logger	0.47±(0.06)	0.09±(0.01)
File Logger	0.13±(0.01)	0.08±(0.02)
Snapshot Creator	0.10±(0.01)	0.08±(0.01)

create database operation and an *upload* file operation. We executed a flow of 1000 transactions. Table V represents the results of the experiment. We can see that MIRES imposes an overhead of 4.3% when configuring the file metadata. We believe that this is an acceptable overhead given the benefits that it provides.

3) Cost of Logging Database Operations: Cloud Logger scripts were deployed on the mobile application container to listen for database changes and flags. The script was deployed on the same region of the application container, to minimize the activation time and assure all necessary triggers. The script was deployed on a Node.js execution environment with 1 GB of dedicated memory. Table VI shows the execution results of the Cloud Loggers, obtained from the workflow made to the database on Section X-A1. 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, CPU/second and the Internet traffic[12]).

4) Cost of Logging File Operations: File Logger scripts were also deployed following the Cloud Loggers deployment model (see Section X-A3). Table VI shows the results of the execution of File Loggers, obtained from the workflow made on Section X-A2

B. Space Overhead

This section evaluates the overhead caused by MIRES in terms of data storage. Notice that MIRES necessarily uses storage space to keep all the data it needs to undo operations, so this overhead is intrinsic to this kind of intrusion recovery approach.

1) Database Overhead: MIRES generate new additional data that is stored on each database document, which imposes a storage overhead. Firebase provides full information about the storage structure of the database[13]. When using MIRES, the size of each document is increased by a minimum of 69 bytes and a maximum of 173 bytes – 69 bytes for the Administrator Recovery, 57 bytes for the Users Recovery mechanism and 47 bytes for the snapshots creation flow (each document has a maximum capacity of 1 MB, which means and occupation between 0.006% and 0.018% of the maximum size allowed). The data is stored on a minimum of 3 field values

[11]We used only the Hify application since it is the only application that provides database-file transactions.

[12]https://firebase.google.com/pricing

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

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%).

Additionally, MIRES creates the three collections mentioned before: the MIRES user flags/recovery/tokens collections. However, only the MIRES user tokens collection stores data persistently as it is needed for recovery, whereas the other data does not persist. For this reason, we only assess the space usage of the MIRES user tokens collection: 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 user tokens collection by reusing the information already stored.

In conclusion, 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 instance, 1M users and 1M documents on a mobile application system increase the storage in 0.41 GB.

2) Database Log Records: Each operation made to the database is logged with specific data (see Section V) given by the following expression (in bytes): $S_{log} = 215 + doc + (53 + data)$ where *doc* represents the path to the document affected and *data* the data sent on the operation; the 53 bytes are only added if the operation wrote any data, i.e., create and update operations. 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.

3) Dependencies: When a write operation is influenced by a read operation, there is additional information logged related with the read operation (see Section V-C). The dependency 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. To exemplify, if a read operation is performed upon the *id*, *username*, *name* and *image* fields of the users' information document, this will lead to 19 bytes (the sum of the field values read). If the path of the document read is `"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

4) File Log Records: MIRES stores the log of the file operation performed. Each log record size is give by the following expressions (in bytes): $S_{dep} = 164 + file + bucket$ for upload operations, and $S_{dep} = 100 + file + bucket$ for delete operations. The *file* property is the file path of the file and the *bucket* property is the file storage name where the file is stored.

TABLE VII
LOG SIZE.

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

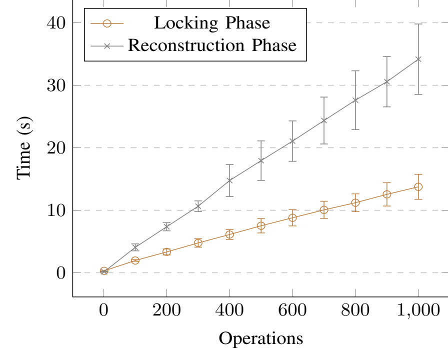


Fig. 5. Time to revert a different number of operations.

Table VII shows the log size needed to store 1M database operations following the exact workflow performed on Section X-A on each application, as also shows the capacity needed to store 1M file operations, following the exact model of Section X-D, where the file size property is 74 bytes and the bucket size property is 23 bytes. Firestore offers a free quota of 1GB. After that, each 1 GB costs $0.18.

C. Administrator Recovery Performance

The total time needed to recover the application, from the moment when the systems starts the recovery until all effects of the intrusion are removed, is defined as the *Time to Recover* (TTR). In MIRES, the TTR is the sum of the *Locking phase* and *Reconstruction phase* times.

To test the recovery performance, we defined three real scenarios. In *Scenario 1* we have created a user in each application and performed a different number of actions resulting in 1 to 1000 operations to recover. In *Scenario 2* we used the same user and the application actions to perform 1 to 10K operations upon the same document, in order to create 1 to 10K different versions of the document. In *Scenario 3* we tested a particular recovery case were intrusions are not propagated to the database service, but instead malicious information is broadcast directly to user applications, through notifications. The rest of the section analyses the MIRES recovery performance in these scenarios.

1) Scenario 1 – Undoing Database Operations: Figure 5 shows the results of undoing the actions of the user. Both Locking and Reconstruction phases increase linearly with the increase of the log size, the dependencies, and the documents to recover (in this test, there was an average of 45 documents for 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.

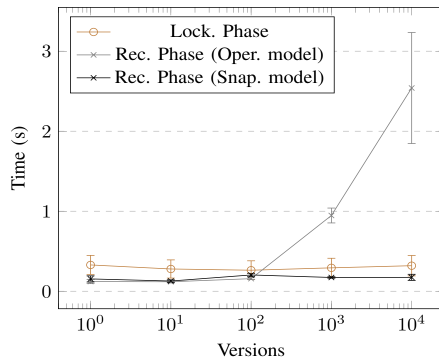


Fig. 6. Time to reconstruct a document with different versions.

The Locking phase is composed of 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. Thereby, this 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 of the reconstruction of the locked documents. We can see that this phase performs worst than the Locking phase: on the Locking phase, MIRES loads and analyses the log on a single process, while on the Reconstruction phase, MIRES needs to load, for each document locked, the operations that affect the document to reconstruct – for legitimate operations – or update the log – for malicious operations.

2) Scenario 2 – Reconstructing a Database Document: Figure 6 shows the results of rebuilding a document with different versions. By following an 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 required to reconstruct the document. However, by using snapshots of 1K versions, we could reconstruct a document with 10K versions in less then 0.5 seconds, instead of the almost 3 seconds required when using the operations model.

We made our in-depth analysis only on the last workflow tested, i.e., the 10k versions. Table VI shows the results observed on the execution of Snapshot Creators. 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.

As expected, the Locking phase times remained practically the same, since it was always only one transaction to be analysed.

3) Scenario 3 – Sending Recovery Notifications: This scenario 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 the CovSense application to simulate a malicious health status change that generated a malicious flow of notifications sent to some users. MIRES sends recovery notifications to each user. We tested the notifications mechanism by sending 1, 10 and 100 notifications. We performed each test 5 times, concluding that sending 1 notification costs 0.06±(0.03) seconds, sending 10 notifications costs 0.81±(0.04) seconds, and 100 notifications costs 5.80±(0.58) seconds.

D. Multi-Service Recovery Performance

We evaluated MIRES when recovering multi-service transactions (Section VII). We used the Hify application and the image post action – this action is supported by a transaction composed of 2 operations: a *create* database operation and an *upload* file operation. We recovered 1 to 1000 transactions, in intervals of 100 transactions. Figure 7 represents the results of the experiment. Recovering 1 transaction takes less than 1 second, while recovering 1000 transactions takes 132 seconds (2 minutes and 12 seconds).

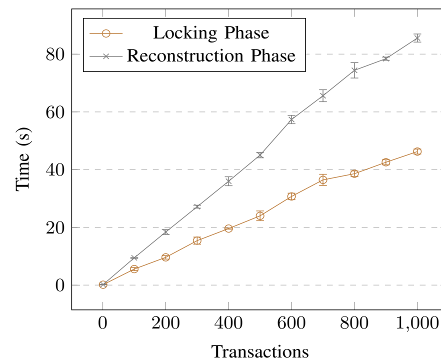


Fig. 7. Time to undo a different number of multi-service transactions.

E. User Recovery Performance

The user recovery mechanism is, similar to the administrator recovery approach, supported by two phases: a locking phase, where the mobile application locks the documents affected by the transaction, and a reconstruction phase, where the locked documents are reconstructed by the MIRES service.

1) Normal Execution: In the course of the normal execution, each time that an invisible database document appears, the Users Recovery Module unblocks the document after 30 seconds. To test the unblocking time, we performed three different flows: we have executed 1, 10 and 100 operations concurrently, each generating a blocked document. This test flow was performed for each application, except the CovSense application. 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.

2) Recovery Execution: The user recovery flow begins with the direct lock of the documents by the mobile application. We have measured the locking phase by locking 1 and 10 documents. This test flow was conducted in each application, except the CovSense application. We observed that locking a single database document costs 0.27±(0.01) seconds, while locking 10 database 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 X-C).

XI. Conclusions

We presented MIRES, the first intrusion recovery service for mobile application systems that use BaaS. The experimental evaluation shows that MIRES is effective and also efficient, as it can revert hundreds to thousands of operations in seconds, with an unavailability period of the application also in the range of seconds. MIRES introduces a set of new ideas, such as the two-phase recovery process, that recovers the state of the mobile application system, minimizes the unavailability of the system during the procedure, and uses the multi-service recovery to revert more complex transactions. Other than the main intrusion recovery functionality, MIRES also presents a user recovery mechanism, allowing application users to undo their last activity. We applied MIRES to recover mobile application systems supporting diverse applications, all of them based on BaaS. However, given its principled approach, it is also possible to use MIRES to recover other types of application, such as web applications.

Acknowledgments This research was supported by national funds through Fundação para a Ciência e Tecnologia (FCT) with reference UIDB/50021/2020 (INESC-ID).

References

[1] F. F.-H. Nah, K. Siau, and H. Sheng, "The value of mobile applications: a utility company study," *Communications of the ACM*, vol. 48, no. 2, pp. 85–90, 2005.

[2] D. Gavalas and D. Economou, "Development platforms for mobile applications: Status and trends," *IEEE Software*, vol. 28, no. 1, pp. 77–86, 2010.

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

[4] C. Troncoso, M. Payer, J. Hubaux, M. Salathé, J. Larus, W. Lueks, T. Stadler, A. Pyrgelis, D. Antonioli, L. Barman *et al.*, "Decentralized privacy-preserving proximity tracing," *IEEE Data Engineering Bulletin*, vol. 43, no. 2, pp. 36–66, 2020.

[5] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

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

[7] J. A. L. Ferreira and A. R. da Silva, "Mobile cloud computing," *Open Journal of Mobile Computing and Cloud Computing*, vol. 1, no. 2, pp. 59–77, 2014.

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

[9] OWASP Mobile Security Project, "OWASP mobile top 10," 2016, https://owasp.org/www-project-mobile-top-10/.

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

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

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

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

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

[15] T.-C. Chiueh and D. Pilania, "Design, implementation, and evaluation of a repairable database management system," in *21st International Conference on Data Engineering*, 2005, pp. 1024–1035.

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

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

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

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

[20] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Jan-Mar 2004.

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

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

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

[24] K. Gai, M. Qiu, L. Tao, and Y. Zhu, "Intrusion detection techniques for mobile cloud computing in heterogeneous 5G," *Security and Communication Networks*, vol. 9, no. 16, pp. 3049–3058, 2016.

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

[26] L. Moroney, Moroney, and Anglin, *Definitive Guide to Firebase*. Springer, 2017.

[27] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou, "Intpatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time," in *European Symposium on Research in Computer Security*. Springer, 2010, pp. 71–86.

[28] S. Thomas and L. Williams, "Using automated fix generation to secure sql statements," in *3rd International Workshop on Software Engineering for Secure Systems (ICSE Workshops 2007)*, 2007, pp. 9–9.

[29] H. F. Korth, E. Levy, and A. Silberschatz, "A formal approach to recovery by compensating transactions," University of Texas at Austin, USA, Tech. Rep., 1990.

[30] A. B. Brown, L. Chung, W. Kakes, C. Ling, and D. A. Patterson, "Experience with evaluating human-assisted recovery processes," in *Proceedings of the 34th IEEE/IFIP International Conference on Dependable Systems and Networks*, 2004, pp. 405–410.

[31] P. Ammann, S. Jajodia, and P. Liu, "Recovery from malicious transactions," *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 5, pp. 1167–1185, 2002.

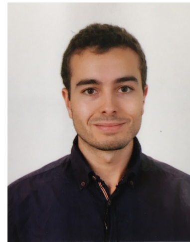
[32] S. T. King and P. M. Chen, "Backtracking intrusions," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 223–236, 2003.

[33] 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*, 2008, pp. 121–128.

[34] 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*, 2009, pp. 484–493.

[35] A. Goel, K. Po, K. Farhadi, Z. Li, and E. De Lara, "The taser intrusion recovery system," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, 2005, pp. 163–176.
[36] 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*, 1999, pp. 110–123.
[37] 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*, 2000.
[38] 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*, 2003, p. 217.
[39] F. Hsu, H. Chen, T. Ristenpart, J. Li, and Z. Su, "Back to the future: A framework for automatic malware removal and system repair," in *Proceedings of the 22nd Annual Computer Security Applications Conference*, 2006, pp. 257–268.
[40] S. Jain, F. Shafique, V. Djeric, and A. Goel, "Application-level isolation and recovery with solitude," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2008, pp. 95–107.
[41] 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*, 2013, pp. 213–227.
[42] D. Matos, M. Pardal, and M. Correia, "Sanare: Pluggable intrusion recovery for web applications," *IEEE Transactions on Dependable and Secure Computing*, 2022, to appear.
[43] 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.
[44] P. Liu, J. Jing, P. Luenam, Y. Wang, L. Li, and S. Ingsriswang, "The design and implementation of a self-healing database system," *Journal of Intelligent Information Systems*, vol. 23, no. 3, pp. 247–269, 2004.
[45] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63–75, 1985.
[46] D. B. Johnson and W. Zwaenepoel, "Recovery in distributed systems using optimistic message logging and checkpointing," *Journal of Algorithms*, vol. 11, no. 3, pp. 462–491, 1990.
[47] N. Neves, M. Castro, and P. Guedes, "A checkpoint protocol for an entry consistent shared memory system," in *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, 1994, pp. 121–129.
[48] D. Vaz, D. R. Matos, M. Pardal, and M. Correia, "MIRES: Recovering mobile applications based on backend-as-a-service from cyber attacks," in *Proceedings of the 17th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, 2020.
[49] I. Costa, J. Araujo, J. Dantas, E. Campos, F. A. Silva, and P. Maciel, "Availability evaluation and sensitivity analysis of a mobile backend-as-a-service platform," *Quality and Reliability Engineering International*, vol. 32, no. 7, pp. 2191–2205, 2016.
[50] M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela, "Serverless execution of scientific workflows: Experiments with Hyperflow, AWS Lambda and Google Cloud Functions," *Future Generation Computer Systems*, no. 110, 2017.
[51] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar *et al.*, "Cloud programming simplified: A Berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.
[52] P. B. Menage, "Adding generic process containers to the linux kernel," in *Linux Symposium*, 2007, p. 45.
[53] S. Xanthopoulos and S. Xinogalos, "A comparative analysis of cross-platform development approaches for mobile applications," in *Proceedings of the 6th Balkan Conference in Informatics*, 2013, pp. 213–220.
[54] N. Jatana, S. Puri, M. Ahuja, I. Kathuria, and D. Gosain, "A survey and comparison of relational and non-relational database," *International Journal of Engineering Research & Technology*, vol. 1, no. 6, pp. 1–5, 2012.
[55] K. Chodorow, *MongoDB: The Definitive Guide*. O'Reilly, 2013.
[56] J. Williams and D. Wichers, "OWASP Top 10 - 2017 rcl - the ten most critical web application security risks," OWASP Foundation, Tech. Rep., 2017.
[57] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold start influencing factors in function as a service," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion*, 2018, pp. 181–188.
[58] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communicatios of the ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.
[59] J.-L. Lin and M. H. Dunham, "A survey of distributed database checkpointing," *Distributed and Parallel Databases*, vol. 5, no. 3, pp. 289–319, 1997.
[60] Sensor Tower, "Q4 2019 store intelligence data digest," 2020.

Diogo Vaz has a BSc (2019) and a MSc (2021) in Computer Science and Engineering by Instituto Superior Técnico (IST), Universidade de Lisboa, Portugal. He is a PhD candidate at IST and a junior researcher at INESC-ID Lisboa. His research interests are in Cybersecurity and Distributed Systems.



David R. Matos has a BSc (2012) and a MSc (2013) in Informatics Engineering from the Faculty of Sciences, University of Lisbon and a PhD (2019) in Computer Sciences and Engineering from Instituto Superior Técnico, University of Lisbon. He is currently a Postdoctoral researcher at FCiências-ID in the LaSIGE laboratory from Faculty of Sciences, University of Lisbon. His research interests are in the area of Distributed Systems and Cybersecurity.

Miguel L. Pardal graduated (2000), mastered (2006), and doctored (2014) in Computer Science and Engineering from Instituto Superior Técnico (IST), University of Lisbon, Portugal. He is an Assistant Professor at IST and a researcher at INESC-ID in the Distributed, Parallel and Secure Systems Group (DPSS), where he is leading the SureThing project (FCT) and completed a participation in the Safe Cloud EU Project (H2020). He is also a Guest Scientist at the Chair of Network Architectures and Services at TU Munich. During his PhD, he was a visiting student at the Auto-ID Labs at MIT. His current research interest is in Cybersecurity applied to the digital frontiers of the Internet of Things and Cloud Computing.

Miguel Correia is a Full Professor at Instituto Superior Técnico (IST), Universidade de Lisboa, senior researcher at INESC-ID, and member of the Distributed Systems Group (GSD). He is coordinator of the Doctoral Program in Information Security at IST. He has been involved in many international and national research projects related to Cybersecurity (QualiChain, SPARTA, SafeCloud, PCAS, TCLOUDS, ReSIST, CRUTIAL, MAFTIA) and has more than 200 publications. His research focuses on Cybersecurity and Dependability (a.k.a. Fault Tolerance) in Distributed Systems, in the context of different applications (Blockchain, Cloud, Mobile).