Sanare: Pluggable Intrusion Recovery for Web Applications 📒

David R. Matos, *Member, IEEE,* Miguel L. Pardal, *Member, IEEE,* Miguel Correia, *Senior Member, IEEE*

Abstract—Web applications are exposed to many threats and, despite the best defensive efforts, are often successfully attacked. Reverting the effects of an attack on the state of such an application requires a profound knowledge about the application, to understand what data did the attack corrupt. Furthermore, it requires knowing what steps are needed to revert the effects without modifying legitimate data created by legitimate users. Existing intrusion recovery systems are capable of reverting the effects of the attack but they require modifications to the source code of the application, which may be unpractical.

We present *Sanare*, a pluggable intrusion recovery system designed for web applications that use different data storage systems to keep their state. Sanare does not require any modification to the source code of the application or the web server. Instead, it uses a new deep learning scheme that we also introduce in the paper, *Matchare*, that learns the matches between the HTTP requests and the database statements, file system operations, and web service requests that the HTTP requests caused. We evaluated Sanare with three open source web applications: WordPress, GitLab and ownCloud. In our experiments, Matchare achieved precision and recall higher than 97.5% with a performance overhead of less than 18% to the application.

Index Terms—Intrusion Recovery, Pattern matching, Learning systems, compensating operations, Recovery, Cloud Computing, Databases

✦

1 INTRODUCTION

W EB applications are exposed to countless threats [1], [2]. Web developers and system administrators adopt intrusion prevention techniques to reduce the probability of attacks being successful, but eventually new vulnerabilities are discovered and exploited by attackers. When an attack succeeds in corrupting the *state* of a web application it is necessary to revert its actions. One way of doing so is to perform regular snapshots of the state that can later be used to revert the application to a point in time prior to the attack. This approach will succeed in removing the effect of the attacker's actions, but at the cost of discarding every legitimate user operation that was executed after the snapshot.

A different approach to revert unintended actions from the application is to use an *intrusion recovery* mechanism that is capable of removing the attackers actions while keeping the remaining ones intact. In the literature there are already intrusion recovery mechanisms designed for web applications [3], [4], [5], [6], [7]. These mechanisms use a combination of regular *snapshots* and operation *logs* to allow system administrators to select unintended actions that should be erased from the state. Notice that in terms of CIA properties these systems are concerned with healing the *integrity* of the application. They are not concerned with *confidentiality* that is an orthogonal concern. They do not prevent intrusions; they remove their effect from the application state.

A drawback of these mechanisms is that they require significant modifications to the source code of the application, databases, or servers running the application. These modifications append extra metadata to the HTTP requests that is passed to the database statements so that it is possible

to trace the effects of the attack in the state, i.e., to identify the database statements that were caused by a specific HTTP request. Although many developers are capable of implementing such adaptations, in many cases this is unfeasible, e.g., because the source code is not available or is too complex. The single intrusion recovery system that aims to solve this problem is Rectify, which does not require any modification to the source code [8]. Rectify assumes a system model in which the web application uses an SQL database to store its entire state. This system model is simplistic since it does not assume that the web application may use the file system or external cloud storage services to save its state. More importantly, Rectify requires the application to use well-formed URIs, which is not the case in many applications.

Current web applications often use different data repositories for different needs. For example, they use a database to store structured and relatively small data records and a cloud storage service to store large data files, such as images and videos. Web applications may also use remote web services to access and modify external data. For example, an e-commerce application may use a web service for the payment service and another web service to emit the invoice. This distribution of the state of the web application makes it unfeasible to use existing intrusion recovery systems that assume a single data storage [3], [4], [5], [6], [7], [8]. Another characteristic of web applications is the vast spectrum of technologies used to implement them. There are several programming languages and many more frameworks. In terms of databases, a developer can now choose between different SQL and NoSQL databases which have their own query languages and characteristics. This complexity in web development makes it difficult to adopt an existing intrusion recovery mechanism that requires modifications to the source code of the application.

• *INESC-ID, Instituto Superior Técnico, Universidade de Lisboa.*

We present *Sanare* ("to heal" in Latin), an intrusion recovery service for web applications that is "pluggable" in the sense that it does not require modifications to the source code of the application. Sanare assumes the web application may keep its state in a set of diverse data repositories, e.g., a database, a file system and a cloud storage service. Sanare allows system administrators to plug agents (self-contained modules) to the different data repositories that will *log* every operation that changes the state of the application. The agents can also log invocations to remote web services, allowing Sanare to notify them that an intrusion occurred.

The main challenge of the work is matching the HTTP requests issued to the application with the operations done in the data repositories. This is even more challenging as we do not assume well-formed URIs. For that purpose, we introduce *Matchare* ("to match" in Italian), a new *deep learning* scheme that does that matching. Matchare is a supervised learning scheme so, during the setup phase, Sanare uses Matchare to learn how the different HTTP requests of the application affect the state, allowing it to adapt to new versions of the application without requiring additional effort from developers. Matchare is based on a Deep Convolutional Neural Network [9], which allows Sanare not to require well-formed URIs, going beyond Rectify that uses simple classifiers (e.g., Naive Bayes). Although Matchare is a supervised learning scheme, the problem it solves is different and more complex than classification. The problem is the association of two timelines of requests, one issued to the application, the other to one of the data repositories. This is not the problem of classifying a request in a class, but the problem of associating pairs of items, one in each timeline. Although we thoroughly searched the literature for solutions for this problem, we did not find any.

Another challenge of Sanare is to provide a modular design that allows developers to install Sanare in a web application that uses several data repositories to keep its state. Sanare solves three problems: (1) it logs every operation that modifies the state of the application regardless of the data repository; (2) it performs recovery in several data repositories, as opposed to current intrusion recovery systems which only recover a single data repository; and (3) it performs recovery without violating consistency of the application data.

Sanare was designed to be deployed as a module, giving system administrators the ability to revert the effects of unintended actions from the application. These unintended actions are HTTP requests that were intentionally caused by an attacker or accidentally by a legitimate user. Sanare allows the system administrator to select these requests from the log of the application or to plug an Intrusion Detection System (IDS) [10], [11], [12], [13] to automate the identification and recovery of the unintended HTTP requests. Recovery is done by executing *compensating operations*[1] in the data repositories that were affected by the HTTP request. These compensating operations erase the effects of the intrusion while keeping the legitimate data intact. This allows recovery to be done *on-the-fly*, meaning that the application does not need to be offline to revert unintended actions, keeping the application available to its users.

Sanare was implemented in Python and packaged in a container, making it ready to be deployed in cloud services. Our implementation is compatible with MySQL [14], an SQL database, MongoDB [15], [16], [17], a NoSQL database, and the most widely used cloud storage services, including Amazon S3, Google Cloud Storage and Azure Blob. The source code of Sanare as well as the container are available for download.[2] [3] Sanare was evaluated in a public cloud with three web applications that have their state distributed between a database and a file system: WordPress [18], GitLab [19] and ownCloud [20]. In the experiments Sanare was able to revert a single intrusion in less than 4 seconds and 100 intrusions in less than 3 minutes, which is very fast compared with an operation that is normally done manually by humans.

The main contributions of the paper are: 1) Sanare, a novel intrusion recovery mechanism that allows the application state to be scattered in several stores, places no restrictions on URI format, and uses deep learning for associating requests to storage operations; 2) Matchare, a new deep learning method that matches HTTP requests with database statements and file system operations, removing the need of modifications to the source code of the application; 3) a prototype of Sanare that can be deployed in a cloud offering to recover from intrusions in web applications; 4) three datasets with samples of HTTP requests and corresponding database statements, file system operations and web service requests from WordPress, ownCloud and GitLab (available in the same GitHub project).

Matchare is used in Sanare but it is made available for other uses, as its advanced capability of matching HTTP requests with data operations is useful for log traceability, for example.

2 WEB APPLICATION ACHITECTURE

Web applications run in a web server with a runtime environment that depends on the language used, e.g., a Java web application requires a server with a Java Virtual Machine (JVM). The web server can be a dedicated server or a virtual machine. Using virtual machines to deploy web applications has some advantages: it is possible to replicate the application by cloning the virtual machine to a different server; it is possible to create snapshots of the state of the virtual machine so that it is possible to recover it to a previous state; developers can replicate the execution environment in their own machines to develop and test their code; it is possible to use orchestration tools, such as Kubernetes [21], to automate the deployment of the application using scripts. When the application is running in a virtualized environment, the term container is often used. A *container* is an execution environment that encapsulates the required software to run the application [21], [22]. The container is a standalone package that can be easily replicated and migrated to other servers. The state of the application can be distributed across databases and file systems.

Figure 1 presents an example architecture of a web application. The web application is replicated in N containers

---

1. The classical term is *compensating transactions* but we use *operations* as they are not necessarily database transactions.

2. Sanare - https://www.github.com/davidmatos/sanare
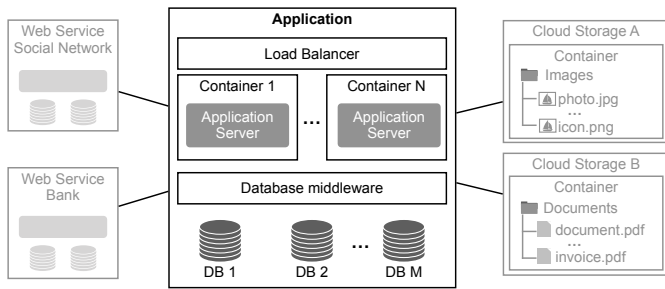3. Matchare - https://www.github.com/davidmatos/matchare

Fig. 1: Example architecture of a web application that uses two external web services and two cloud storage services.

to cope with the traffic. Each replica is running in an application server inside a container. A load balancer redirects the user's requests to the corresponding replica and distributes the load evenly. The database is also replicated in M instances; a database middleware handles the coordination and data replication. In the figure, the application interacts with two web services. One belongs to a social network and another to a banking service. On the right side of the figure are two cloud storage services that are used to store images and documents.

This example shows a set of technical challenges we identified in terms of intrusion recovery: external web services; state of the application distributed in a database and cloud storage services; application deployed in stateless containers that can be replicated to respond to high traffic.

3 SANARE OVERVIEW

This section presents the system model and the architecture of Sanare.

3.1 System Model

Sanare is an intrusion recovery system that recovers web applications. The application that is monitored and recovered by Sanare is referred to as *application*. Sanare has connectivity with the application, i.e., it is able to intercept HTTP requests sent to the application and connect directly to its database and storage. We make the following assumptions:

- the database used by the application can be SQL or NoSQL. For NoSQL databases, we assume there are constructs equivalent to tables, records and columns;
- the file system used by the application provides an interface that is equivalent to POSIX [23] or similar to that of an object storage service (e.g., Amazon S3);
- the application may interact with external web services that are not controlled by the applications' owner;
- the application does not fail while Sanare is recovering the application state;
- it is acceptable for users to observe temporary inconsistencies during the execution of a recovery, i.e., the application is eventually consistent [24].

In the remainder of the paper, we use r to represent an HTTP request, s for a database statement, f for a file system operation, and w for a web service request. We use subscripts to differentiate requests and operations whenever needed.

3.2 Threat Model

We consider the following threat model:

- **t1**: intrusions can only occur at the application level in the form of HTTP requests;
- **t2**: a faulty HTTP request results in the execution of faulty database statements and faulty file system operations that corrupt the state of the application;
- **t3**: a faulty HTTP request may also result in the invocation of faulty web service requests which will corrupt external services that are not controlled by the application's owner;
- **t4**: the attacker cannot tamper with Sanare or the code of the web application nor can he tamper with Sanare logs;
- **t5**: the attacker has no direct access to the database and file system of the web application, only through requests to the application;
- **t6**: every request that reaches the application passes through and is logged by Sanare.

Assumptions **t1** and **t2** indicate that attacks can corrupt the state of the application in both the database and file system and should be recovered by Sanare. With assumption **t3** we acknowledge that intrusions may also affect external web services that are not controlled by the application's owner and cannot be directly recovered by Sanare. Instead these web services are contacted by Sanare so that they can fix or undo the faulty requests. With assumptions **t4** and **t5** we limit the scope of the attacker so that we can focus on the theme of this paper, which is intrusion recovery for web applications. For **t4** there are some solutions that allow to detect corruption of logs [25], [26]. For **t5** it is possible to limit the access of the application for Sanare alone by configuring a firewall such as, iptables [27].

3.3 System Architecture

Sanare runs in a container alongside the application and acts as the entry point of the application to the users. For this to work the system administrator needs to configure the DNS server to ensure that the application's address points to Sanare's IP address, i.e., Sanare acts as a reverse proxy, forwarding the user's requests to the application's container. It also intercepts the application's database statements, file system operations, and web service requests. This way it is able to log all the operations that are caused by the HTTP requests. This process is invisible to the user. From his point of view he is interacting directly with the application.

Figure 2 presents the architecture of an application with Sanare. In the figure there are two containers. The container on the left is running the application. The container on the right is running Sanare, which is composed by the Sanare Manager, the HTTP Agent, the Web Services Agent, the Database Agent, and the File System Agent. The Sanare container uses two databases: one to store the logs and another to store the samples used by the deep learning algorithm. The container running the application is only accessible by Sanare, meaning that users cannot issue requests directly to this container. Besides the containers, Sanare also needs a Cloud Storage service to store backups to ensure that the logs do not grow indefinitely. Next we present each component in more detail.
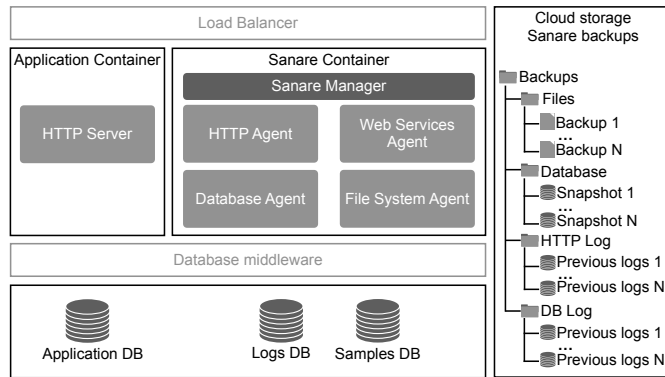
Fig. 2: Architecture of a system with Sanare. Sanare itself is the set of components inside the container on the right.

HTTP Agent: is responsible for intercepting every HTTP request to the application and storing them in the log. HTTP requests do not receive any special treatment since that would reduce the overall performance of the application.

Web Services Agent: intercepts and logs requests to external web services. Like the HTTP Agent, the Web Services Agent does not modify the HTTP requests before logging them.

Database Agent: acts as a reverse proxy and intercepts every statement to the database and logs it in the log iff (if and only if) the execution of the statement returns without error. Erroneous statements are not recorded since they do not have to be recovered.

File System Agent: is responsible for logging storage operations. If the storage is the local file system, the agent starts a watchdog that monitors the file system and issues a notification when there is a change, so that the agent can log every file system operation. If the file system is a cloud storage, the agent acts as a reverse proxy, intercepting the HTTP requests that are destined to the cloud storage. The API to access cloud storage services is accessed by the HTTP protocol.

Logs DB: is a database that stores the logs of the executed operations. It contains the executed database statements, HTTP requests, file system operations, and web service requests. By keeping all the logs in the same database server it is easier to aggregate them, allowing the system administrators to search and analyze the executed operations. It is also a recommended practice [28] to store the logs of a distributed application in the same repository.

Sanare Manager: manages the recovery process, automatic backups and allows the system administrator to configure Sanare. In more detail:

- configuration and setup: it allows the system administrator to configure the ports used by the proxies, the access keys to the cloud storage, the integration with an IDS, and other security parameters;
- automatic backups and garbage collection: it performs automatic backups of the logs to the cloud storage and deletes every log entry. This can be configured to be done periodically (e.g., every 2 weeks) or when a threshold is achieved (e.g., when the log reaches 100GB in size);

- automatic snapshot of the application: from time to time Sanare creates a copy (snapshot) of the application state, which includes both the database and existing files. This snapshot will be used during recovery. The frequency with which Sanare creates snapshots of the application is configured by the administrator, like the backups it can be either by a period of time or by the storage space;
- monitoring: a system administrator is able to analyze the logs using the Sanare Manager. This can be helpful to identify possible intrusions or to tune the automatic backup parameters;
- intrusion recovery: when a system administrator identifies an intrusion, i.e., a malicious HTTP request, he can start the recovery process. This process is orchestrated by the Sanare Manager and it consists in a series of steps that include: assess the damage caused by the faulty HTTP request; calculate and execute the compensating operations; notify the affected external web services about the intrusion; and present a notification to the users to inform them about the recovery so they can be prepared for some inconsistencies. This process is explained in detail in the next section.

3.4 Execution phases

When Sanare is plugged into an application, it runs in four phases:

Learning phase (Section 5): executed before the application is available to the users. In this phase Sanare executes a series of HTTP requests in order to gather examples of the database statements, file system operations and web service requests caused by the HTTP requests. Then these requests are used by the Matchare scheme for training;

Normal phase: the application is available to the users. In this phase Sanare logs every HTTP request, database statement, file system operation, and web service request. Sanare also performs automatic backups of the application and the logs.

Damage assessment phase (Section 6): when the system administrator identifies the malicious HTTP requests that need to be undone, Sanare uses Matchare to identify the database statements, file system operations and web service requests that were caused by the attack;

Damage repair phase (Section 7): Sanare executes the compensating operations to undo the effects of the attack. In this phase the application is still available to the users, but they may observe some inconsistencies or downgraded performance due to the execution of the compensating operations.

4 MATCHARE

The problem solved by Matchare is to match an HTTP request to the database statements, file system operations and web service requests that it caused when it was executed. Data about these statements, operations, and requests is stored in a log. To the best of our knowledge, this problem of matching HTTP requests with operations from different data repositories without requiring modifications to the

source code, has never been solved in the literature. In the case of Sanare, this is needed to identify the effects of the unintended requests from which we want to recover, a hard operation to do manually. However, Matchare may be useful for other purposes beyond intrusion recovery, e.g., for log traceability, debugging or analytics.

Matchare solves this problem by resorting to a supervised deep learning scheme. Specifically, the scheme can be considered to be a binary classification algorithm that aims to decide if the pair {HTTP request, data operation} is a match, i.e., if the operation was caused by the HTTP request (here data operation can be a database statement, file system operation or a web service request). Matchare verifies every possible combination of HTTP request and operation that occurred in a period of time, meaning that if there are n HTTP requests and m operations, then Matchare needs to verify $n \times m$ pairs.

4.1 Matching scheme

The scheme that finds matches between the pairs {HTTP request, data operation} uses a dataset with all combinations that an HTTP request can have with every database statement, file system operation and web service request. During recovery, the matching scheme tests the malicious HTTP request with every operation that occurred in a time window that starts roughly when the HTTP request was executed and ends after a period of time elapsed (configurable, e.g., a couple of seconds). Sanare does not use the timestamp of the HTTP response since this may ignore some data operations that are executed afterwards. In our experiments we noticed that some operations are still being executed when the HTTP response arrives (for example, some cloud storage operations are executed asynchronously, allowing the HTTP response to be sent to the user before completing the operation). In practice Sanare uses Matchare with three matching models, since the features used for the database statements, file system operations and web service requests are different. The matching scheme, Matchare itself, is the same, but it is trained three times, one for each type of operation.

The matching scheme uses a Deep Convolutional Neural Network (Deep CNN) [9]. In preliminary experiments we tested other classification algorithms such as, Naive Bayes [29], Logistic Regression [30] and Random Forest [31], which provided good results in Rectify with well-formed requests [8]. However, when we tested these algorithms to match the HTTP requests with the operations in the GitLab and ownCloud, we could not get them to correctly match more than 50% of the examples. In our experiments, the Deep CNN algorithm greatly outperformed these algorithms. The main reason why Naive Bayes, Logistic Regression and Random Forest fail to correctly match the requests with the corresponding operations is that both ownCloud and GitLab are more complex applications than those tested in [8], thus requiring more features to train the model. It is possible that with more features these algorithms could achieve the same performance. However, for these algorithms having more features would require more tuning when compared with a Deep CNN [9] making it more difficult to reach the same results.

The Deep CNN uses several hidden layers with each layer having a set of nodes. Our models were composed by a set of 5 layers with each layer having between 4 and 7 nodes. The exact number of nodes depends on the model. The activation function used at a given layer l is given by

$$a^{l+1} = \sigma(W^l a^l + b^l) \tag{1}$$

where W^l denotes the weights used at layer l, a^l the activation for the nodes at layer l, b^l the bias and σ the non-linear activation function $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ (hyperbolic tangent function). At the final layer we use the Rectified Linear Unit (ReLU) function ($f(x) = max(0, x)$), which will result in a non-negative probability of the given pair ({HTTP request, data operation}) being a match. In our experiments the ReLU function achieved a better performance than the $sigmoid$ and $tanh$ functions. To classify the pairs that match we defined a threshold of 70%, meaning that every pair that has a probability greater or equal than 0.70 is considered a match. The threshold value was defined in an empirical manner from the first experiment results.

4.2 Features

The features used by the matching scheme depend on the type of operation. For the database statements we have specific features that are different from those used for file system operations. However, the HTTP request features are the same for the three datasets.

HTTP requests features:

- Verb: GET, POST, PUT or another;
- URI: relative path of the request;
- Bytes sent: size of the request;
- Number of parameters: total number of parameters of the request;
- Parameters (names and values): name and value of every parameter.

Database statements features:

- Type: type of database statement, e.g., *insert*;
- Table: table targeted by the database statement;
- Columns: number of columns in the statement;
- Columns (names and values): names of the columns and corresponding values;
- Hamming distances [4] between every pair of HTTP parameter value and database column value.

File system operations features:

- Operation: type of operation (create, delete, copy, etc.);
- Source path: path of the affected file (for the copy and move operation this corresponds to the origin file);
- Destination path: path where the file will be copied or moved into (if it applies);
- Is directory: a Boolean value indicating if the affected file is a directory or a file;
- Hamming distances between every combination of HTTP parameter value and source and destination paths.

Web service requests features:

---

4. The Hamming distance is a metric for comparing two binary data strings, counting the number of bit positions that are different.

- Verb: GET, POST, PUT or another;
- URL: absolute path of the web service;
- Bytes sent: size of the request;
- Number of parameters: total number of parameters of the request;
- Parameters (names and values): name and value of every parameter;
- Hamming distances between every combination of an HTTP request and web service parameter values.

The Hamming distances capture something that we observed in the logs: that frequently the same string, or a variant, appears in the HTTP request and the database statements or file system operations it causes. This notion is crucial to understand if a request matches an operation. However, it is worth noticing that this may not be true in some cases (for example, when the application performs some computation on the parameter's value before writing it to the database). Nevertheless, we opted for using Hamming distances of these strings as they contributed to a better accuracy of the deep learning models.

As mentioned, the scheme is a form of binary classification, so it assigns a class, called *match*, true or false, to every pair {HTTP request, operation}. Moreover, it is a supervised learning scheme, so it requires a set of pairs labelled with that class for training. It needs a balanced training dataset, so the number of pairs that match and do not match should be the same [32].

Table 1 shows an example of four samples that were generated by Sanare with example HTTP requests. The first features come from the HTTP request, the following from the database statement, and the last column is the label. The label indicates if the database statement was caused by that HTTP request or not. The second sample does not match; it was created for training purposes.

4.3 Pre-processing the features

Before training and using the model to classify the logs, we need to pre-process the features to normalize the numeric features and encode the categorical (text) features.

Normalization refers to the process of scaling numeric variables so that they all fit in a smaller interval. By normalizing the numerical features the model converges to the optimal weights in less steps [33], making the training more efficient. The method we use to normalize is the Z-score normalization in which every numeric value x is converted to x' such that $x' = \frac{x - \bar{x}}{\sigma}$.

Encoding the categorical features involves creating a new variable for each possible value that a categorical feature has. For example, the variable *mode* (from the file system's log) can assume one of three possible values: *read*, *write* and *execute*. During encoding the variable is removed and three new variables are created in the dataset: *read*, *write* and *execute*, then in each sample one of these variables will be 1 and the remaining ones 0.

5 LEARNING PHASE

This section focuses on Sanare/Matchare's learning phase. This phase has to be executed before the application is available for the users and every time it gets updated with

new requests, new operations, or changes in the relations among them. The phase has two steps: 1) simulating HTTP requests from users and 2) training the machine learning model with the results of Step 1.

5.1 Simulating HTTP requests

Sanare simulates the execution of HTTP requests so it can observe the database statements, file system operations and web service requests that are generated by them. The simulation works by executing a list of HTTP requests. This auto-labeling of operations is automatic and it only requires the system administrator to provide a list of HTTP requests. The list should include at least one request for each endpoint of the application, but does not need to include every possible HTTP request (as this would be impossible). For example, the request: `application.com/user/create?name=john` is sufficient to represent the endpoint `/user/create`, it is not necessary to provide other requests with every possible `name` parameter. The list should be extensive enough since the endpoints that are not simulated by Sanare cannot be matched by Matchare and, therefore, cannot be recovered. If a specific HTTP request produces different operations depending on the current state of the application (for example, if the current logged in user belongs to a specific user group), then the system administrator has to provide different HTTP requests for different users to illustrate these situations. This list of HTTP requests is executed in a container that should be in an isolated environment, such as a *test* server, so that the application does not execute external web services during this phase. Instead, during this stage of the learning phase the application should use a mock testing approach [34]. In mock testing, external web services are replaced by dummy components that simulate them so that all features of the application can be tested without issuing requests to real web services. Each HTTP request is executed at a time so that there are no concurrent HTTP requests and it is possible to know exactly which database statements, file system operations and web service requests it executes.

For each simulated HTTP request, Sanare stores in the Samples DB the pairs {HTTP request, data operation} for every operation (in the database, file system and web services) it caused. After the simulation, the Samples DB should have three datasets similar to the example in Table 1: one for database statements, one for file system operations and another for web service requests. Notice that we can use our dataset of only matches to create more pairs that do not match, by combining different HTTP requests from training with different DB statements. This will give us sufficient non-matching statements, which we need to train deep CNN (as explained in Section 4). After simulating every HTTP request Sanare starts training the Matchare models.

5.2 Training the models

The output of the training phase is a model. A model is defined by an algorithm and a set of parameters (weights and biases) that are adjusted during training. These parameters are used by the deep learning algorithm to calculate a probability of a sample (in this case, a pair {HTTP request, data operation}) being true or not. So, during the training phase, a set of parameters are tuned throughout several iterations,

TABLE 1: Example of samples of HTTP requests with database statements.

| Sample ID	HTTP Features								Database Features								Hamming Distance 1	Hamming Distance 2	Hamming Distance MxN	Label
	Verb	URI	Bytes send	Params	Param 1	Value 1	Param N	Value N	Type	Table	Columns	Column 1	Value 1	Column N	Value N					MATCH
1	GET	search.php	45	1	q	News	-	-	SELECT	-	5	q	News	-	-	0	-	-	1
2	PUT	comment.php	32	2	user	John	comment	This is a…	INSERT	COMMENT	6	user	David	comment	Another…	0	0	5	0
3	POST	post.php	312	2	user	Michael	topic	National	SELECT	-	4	user	Michael	topic	National	0	0	0	1
4	POST	post.php	343	2	user	Michael	subject	Tech post	INSERT	POST	4	Topic	David	Text	Tech post	0	0	0	1
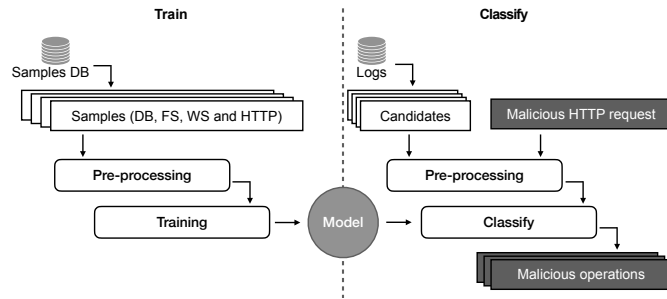
Fig. 3: Training and classification with Matchare.

in each iteration the accuracy of the model improves. The training is done in a sample of examples, called training set, and tested in each iteration in another sample of example, called test set. When training finishes the set of parameters is stored to be later used to classify the pairs {HTTP request, operation}.

There are three models to be trained due to the differences between the types of operations: one model for database statements, another for file system operations, and a third for web services. Besides the different features, the three models are trained the same way, therefore the following description is valid for all the models. Using the samples collected earlier the matching model will gather information that will help it understand how database statements, file system operations and web service requests differ when executing the same HTTP request.

The left-hand side of Figure 3 shows the steps executed during training. First the samples are fetched from the Samples DB, then they are pre-processed (see Section 4.3) and finally the model is trained (as described in this section). The trained model is then stored to be used during recovery to classify the pairs {HTTP request, data operation}.

6 DAMAGE ASSESSMENT PHASE

The damage assessment phase consists of identifying the database statements, file system operations and web service requests caused by the malicious HTTP request. This is done using Matchare and the models trained during the learning phase (Section 6.1). Moreover, Sanare also needs to find the operations that depend on the operations directly caused by the malicious HTTP request (Section 6.2). For example, if a malicious HTTP request results in a database statement that creates a new record in the database then the statements that access that record depend on them and should be undone as well.

6.1 Operations directly caused by HTTP requests

The malicious database statements and file system operations that need to be reverted, as well as the web service requests that need to be reported to the web services owners, are in the Logs DB. To find them, Sanare scans the logs in a

time interval of the intrusion. The time interval is calculated according to the skew, time difference, in the clocks of the HTTP server and the database server hosting the logs using Cristian's algorithm [35]. With the skew of the clocks calculated it is necessary to define a time interval for Sanare to scan the logs. The duration of the time interval cannot be too long, otherwise Sanare wastes time looking into the incorrect log entries, or too short, to avoid missing the log entries caused by the HTTP request. The time interval we established starts in the moment t of the intrusion minus two times the skew and it ends in $t + r + 2 \times skew$ being r the maximum latency observed by any HTTP request from the application.

With the set of candidate operations (all operations that occurred during the time interval), Sanare can combine them with the malicious HTTP request(s) and use the trained models to classify them, i.e., to detect if they match or not. Similarly to training (Section 5.2), this is done in two steps: pre-processing the logs alongside with the malicious HTTP request; and running the trained models to classify. The right side of Figure 3 represents how this process is done.

6.2 Operations indirectly caused by HTTP requests

This section is not concerned with the operations that match the malicious HTTP requests, identified using Matchare, but those that *depend* on them (as seen in the example from the beginning of Section 6). Sanare uses a *dependency graph* to identify the database statements that depend on those that are malicious and spread the execution of the compensating operations to them. Sanare does not calculate a dependency graph for file system's operations since they have access policies very different from a database. For example, most file systems only allow a file to be written by a single process at a time, while in a database several processes may compete to write in the same table. We also exclude web services as they are external to the application, so it is infeasible to grasp dependencies that they may cause.

To obtain the dependency graph, Sanare scans through the HTTP requests and database statements in the log and, for each one of them, verifies a set of rules. Sanare needs to verify the dependency between HTTP requests since they will be used to infer dependencies among the database statements they caused. We use a set of rules to define the dependencies between requests:

HTTP1. For any two HTTP requests r_i and r_j, $r_i \rightarrow r_j$ (i.e., r_i precedes r_j or r_j depends on r_i) if

$$ts\_resp_{r_i} < ts\_req_{r_j} \wedge session_{r_i} = session_{r_j}$$

being $ts\_req_{r_j}$ the timestamp of the request r_j, $ts\_resp_{r_i}$ the timestamp of the response to r_i and $session_{r_i}$ the session of the HTTP request r_i. In other words, r_j depends on r_i if r_i completed before r_j and if both were executed in the same HTTP session.

HTTP2. For any two HTTP requests r_i and r_j, $r_i \to r_j$ if

$$ts\_resp_{r_i} < ts\_req_{r_j} \wedge user_{r_i} = user_{r_j}$$

being $user_{r_i}$ the username of the HTTP request r_i. In other words, r_j depends on r_i if r_i completed before r_j and if both were executed by the same user. This rule is targeted at web applications that handle users with usernames. A drawback of this rule is that if an attacker impersonates a legitimate user he may create dependencies that should not be considered, otherwise recovery will erase legitimate operations. For this reason this rule is optional and can be deactivated by the system administrator.

DB1. For any two database statements s_i and s_j, $s_i \to s_j$ if

$$ts\_res_{s_i} < ts\_req_{s_j} \wedge Rows_{s_i} \bigcap Rows_{s_j} \neq \bot$$

being $ts\_req_{s_j}$ the timestamp of the execution of s_j, $ts\_res_{s_i}$ the timestamp of the result of the execution of s_i and $Rows_{s_i}$ the set of records affected by s_i. This rule considers the case when the database statement s_i completed before s_j was executed and both statements had at least one database record in common.

DB2. For any two database statements s_i and s_j, $s_i \to s_j$ if

$$\forall Rows_{s_i}, \forall Rows_{s_i} \exists Rows_{s_i}(pk) \to Rows_{s_i}(fk)$$

being $Row_{s_i}(pk) \to Row_{s_i}(fk)$ a primary key-foreign key relationship. This rule is used to ensure that, if a record needs to be deleted, then every record that has a foreign key connected to it is deleted first.

With these rules Sanare is able to create tuples representing relations between HTTP requests and database statements. For example the tuple $R = (r_1, r_2, r_3)$ states two dependencies: r_3 depends on r_2 and r_2 depends on r_1. Using these tuples Sanare constructs the dependency graph.

The algorithm to construct the dependency graph works the following way. First, Sanare takes r_f, the malicious HTTP request that was identified by the system administrator and collects every HTTP request that depends on it using rule **HTTP1**. Then, for each HTTP request it collects the database statements that were issued by it. This is done using the algorithm described in Section 6.1. Then, for each subset of database statements Sanare applies rules **DB1** and **DB2** and creates the tuples with the dependencies. Using these tuples Sanare is able to create n dependency graphs, one for each HTTP request. Finally, Sanare connects the dependency graphs the following way: the statement that does not depend on another statement connects with the one from the following dependency graph that has no dependencies. If there is more than one HTTP request identified as malicious, the process is executed once per request.
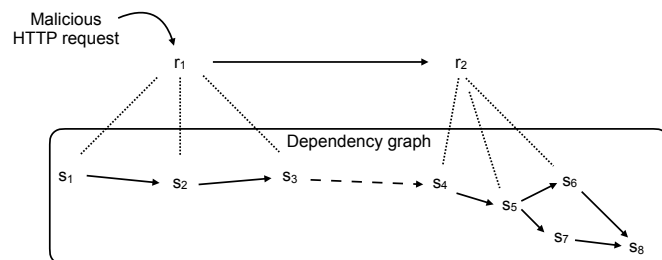

Fig. 4: Example of a dependency graph.

Figure 4 presents an example of a dependency graph. In the example, r_1 is the malicious HTTP request that the system administrator intends to recover from. By using the rule **HTTP1**, Sanare infers that r_2 depends on r_1. Then, it generates two dependency sub-graphs: one with s_1, s_2 and s_3 and another with s_4, s_5, s_6, s_7 and s_8. Finally, Sanare connects both dependency graphs by connecting s_3 with s_4 because: (1) there is no database statement that depends on s3 and (2) s4 is the first operation from r2. Sanare connects these two graphs because since r_2 depends on r_1, then the first operation caused by r_2 depends on the last operation caused by r_1.

7 DAMAGE REPAIR PHASE

To repair the damage caused by the malicious HTTP request(s), Sanare executes compensating operations on the database, on the file system, and contacts the remote services, to undo the effects of the intrusions. In other words, the compensating operations will revert the affected data objects (files and database records) to the previous version prior to the attack. If a legit user did modify a corrupt data object, his changes will be lost since Sanare tagged that data object as corrupt from the moment the attacker modified it. Each transaction undoes a single database statement or file system operation or a remote operation, and they differ depending on the target. A compensating operation that undoes an SQL database statement is an SQL statement while a compensating operation for the file system consists in a file system operation and a compensating operation on a web service is a remote call.

7.1 Compensating operations in the database

The compensating operations for a database vary depending on the database management system. An SQL database, such as PostgreSQL [36] has its own language which is different than a NoSQL database, such as MongoDB [15]. To cope with this heterogeneity, Sanare has an adapter for each of the most used database management systems. It is possible for developers to create more adapters making it compatible with more database management systems. Sanare should be capable of recovering any database as long as the CRUD (*create*, *read*, *update* and *delete*) operations are available. Read operations are ignored by Sanare to calculate the compensating operations since they do not modify the database records. For the remaining operations, the compensating operations are calculated as follows:

- *create*: a record created by an attacker is always corrupt, even if a legitimate user alters it later on. As such it should be deleted. The compensating operation for a *create* operation consists in a *delete* that targets the created records;
- *update*: records that are modified by an attacker using an *update* operation should be reverted back to their previous versions prior to the attack. Sanare does this by calculating an *update* that reverts the affected records back to the most recent version before the attack;
- *delete*: when an attacker deletes a record it should be recreated during recovery. Sanare does this by calculating a *create* statement that inserts the removed record back to the database.

7.2 Compensating operations in the file system

The compensating operations for file system operations should be capable of reverting the affected files and folders back to most recent version prior to the attack. Even if legitimate users modify a file that was corrupted by an attacker, such modifications are ignored. We chose this approach since it is not possible to reconstruct every file ignoring previous modifications.

The compensating operations for the files use a combination of file system backups, that are stored in an external cloud storage, and file system operations, that are stored in the Logs DB. The approach adopted by Sanare to create backups of files is: if a file f is modified, then a new version of f is created in the file system and the previous version is moved to the cloud storage. If a file f is moved from one folder to another, then it is not copied to the cloud storage, instead, the *move* operation is stored in the log. If a file is deleted then it is copied to the cloud storage and deleted in the local file system of the application. When a file is created, a copy is uploaded to the cloud storage. This approach is used in append-only file systems and cloud storage services, such as Microsoft OneDrive, that stores up to 500 versions of each file by default.

The compensating operations for files should take into account the tree structure of the file system. For example, if an attacker creates a new folder and moves several files to that folder, the compensating operations should not just delete the folder and its files. Although the structure of the file system was modified by the attacker, the files were not. For this case Sanare should move the files back to their previous location and then delete the folder created by the attacker. It may happen that legitimate users create files in a directory created by an attacker. By default, Sanare deletes these files as well; even though they were not corrupted by the attacker, it is not possible for Sanare to know where to put such files. However, before executing the compensating operations, Sanare always asks the system administrator if he wishes to copy these files to a new location.

7.3 Compensating operations in remote web services

The compensating operations for remote web services are not calculated by Sanare. These remote web services are controlled by third parties and have different interfaces and functionality. For example, it is typical for a web application to contact a remote web service to issue payments and another to emit receipts. Each of these web services has a different protocol making their compensating operations specific for each one. Furthermore, it is a good practice in remote web services to provide an interface that allows to *undo* previously executed operations (for example, refund a payment from a returned purchased item). One example of such *undo* operations is the PATCH method [37], [38] that was introduced in the HTTP protocol for this situation.

We assume that remote services are prepared to receive repair messages from Sanare that will trigger the required actions to recover their state. Sanare allows the system administrator to manually configure compensating operations for each remote web service. During recovery Sanare will invoke these compensating operations. This also allows the system administrator to make the compensating operations

more sophisticated, for example, by invoking an extra web service that notifies the users of the application that a recovery process was performed.

7.4 Data consistency after recovery

To maintain data consistency after the recovery finished, it is necessary to execute all compensating operations atomically; if only a subset of the compensating operations were executed, then the application would become inconsistent, with part of its state reflecting the effects of the attack and another the effects of the recovery. To avoid this situation, Sanare executes recovery atomically, executing every compensating operation inside a system-wide transaction. The system coordinates the execution of the compensating operations in the following way:

1) for each database record affected by the compensating operation, Sanare saves in memory the current value of the database records. These values are saved in the form of a database statement (rollback transaction) that is capable of reverting the records back to their previous version;
2) for each file affected by the compensating operation, Sanare saves in memory the file;
3) Sanare executes the compensating operations in both the database and the file system;
   a) if at least one compensating operation failed then system uses the rollback transactions to revert the database records to the pre-recovery state. The files affected by the compensating operations are reverted as well. The recovery is considered unsuccessful and the system administrator is alerted to try again;
   b) if every compensating operation is successfully executed then the recovery process is considered completed and the system administrator is alerted.

It is worth mentioning that during the execution of Step 3 the users may observe some inconsistencies as the system is being recovered and some records may be recovered while others may still reflect the effects of the attack. We assume that this behavior is accordingly with our system model that assumes an *eventual consistency* guarantee, which means that users may observe some inconsistencies but, eventually, the application becomes consistent and from that point forward it will be consistent.

7.5 Dealing with external inconsistencies

External inconsistencies happen when users of the application observe effects of the attack and later on, after recovery finishes, these effects of the attack disappear. One example is when an attacker creates messages that are illegally sent to users. The recipients of these messages read and can even reply to them, but once recovery removes those messages, the users may believe the application is malfunctioning. Since it is not possible to make the users forget they read the attacker's messages then the users should be notified that a recovery took place and therefore they may experience some inconsistencies. Sanare handles this by allowing the system administrator to leave a message to users to let them know the application was under maintenance and they may experience some inconsistencies. For example, if

the system administrator recovers the request to the URL www.site.com/posts/a_post, then only the HTTP requests to this location get the notification. This avoids having a notification of a recovery in a completely different part of the application.

8 IMPLEMENTATION

We implemented Sanare as a microservices application [39], [40] where each component of Figure 2 is a microservice. Sanare can be deployed in a single container or distributed, with each component running in its own container.

8.1 Sanare

Each component of Sanare is a microservice that was implemented in Python using the Flask micro web framework [41]. The logs are stored in MongoDB [15], [17] databases.

Sanare Manager: is the coordinator of the system. It starts the remaining microservices and coordinates the recovery process. The Manager can also turn off unused components, for example, if the application does not use web services then the Manager can turn off the Web Service Agent. This microservice has access to a user of the cloud to allow it to start a replica of the application during recovery. This is done using the API of the cloud provider. It is also the only microservice that is accessible by the users of Sanare. Therefore, this is the only microservice that has ports opened for outside traffic.

HTTP Agent: uses NGINX [42], configured as a reverse proxy, to intercept and log every HTTP request that targets the application. Each log entry is then synchronized to a database (HTTP log) with RSYSLOG.[5] The HTTP Agent does not modify the received HTTP request and it logs it as is, this way it interferes as little as possible with the application throughput.

Web Service Agent: is another instance of NGINX that is configured as a proxy that captures every HTTP request that comes from the application. Like the HTTP Agent, the Web Service Agent does not modify the HTTP requests, it just logs and synchronizes them, with the aid of RSYSLOG, to the WS Log.

File System Agent: logs file system operations that occur in the file system where the application is hosted. We implemented this component in two different ways: as a mounted volume using FUSE,[6] and as a listener process that runs in the application container using Watchdog.[7] Each approach has its advantages and disadvantages. With FUSE it is possible to mount an external file system (for example, a cloud storage) and use a dedicated storage recovery system [26], however, this approach requires the application container to be configured to use the mounted volume, which may not be possible in some cloud providers. With Watchdog the setup is simpler because it consists in starting a process in the application container. In both approaches the File System Agent logs file system operations to the Logs DB without modifying them.

5. RSYSLOG – https://www.rsyslog.com
6. Filesystem in Userspace, https://github.com/libfuse/libfuse
7. Watchdog: Python API and shell utilities to monitor file system events – https://pypi.org/project/watchdog/

Database Agent: is a process that runs in the same machine of the application's database and it is responsible for two things: first, when it starts it configures the database of the applications to log every statement in its built-in log; second, it periodically synchronizes the application's database log to the DB Log and garbage collects the old log entries. This approach is different than the one used by the HTTP Agent, which uses a proxy to intercept the requests. We opted for this approach as it performs faster than passing requests through an additional component.

8.2 Deploying Sanare

Sanare was designed to be deployed alongside the application. Given its distributed architecture, it can be deployed in three different ways: embedded, single container, or distributed.

Embedded deployment: Sanare runs in the same container of the application. This approach is adequate for simple applications that do not have high demanding traffic requirements. It is easier to manage, since it does not add the inherent complexity of distributed applications and it requires less resources. It is also cheaper, since it does not require extra machines to run Sanare. The drawback is the performance degradation of having more processes running in the same machine.

Single container deployment: allows the administrator to assign a container to run every component of Sanare. This way the application's container does not get overloaded and the administrator can scale up the machine running Sanare if it is degrading the application's performance. This deployment mode is adequate for medium sized applications that have more demanding traffic requirements.

Distributed deployment: allows each component (microservice) of Sanare to run in its own machine. By distributing every component of Sanare it is possible to scale up the components that are degrading the performance of the application. For example, it is expected that the HTTP Agent, which responds to every user, has far more requests to handle than the Sanare manager, which is only accessed by the system administrator. The drawback of this approach is the complexity of deploying and managing a distributed system. However, it is possible to use a tool like Kubernetes [21], [43] to automate the deployment process. This approach is the most indicated for complex applications that may already be distributed.

9 EXPERIMENTAL EVALUATION

In our experiments we wanted to find the answer to these questions: (a) How does Sanare impact the performance of the application? (b) How much storage is needed to keep the logs and the necessary recovery data? (c) How long does it take to recover an application when the number of malicious HTTP requests varies? How much does it cost to use Sanare in public clouds? What is the accuracy of Sanare in matching operations that cause modifications to the state with the corresponding malicious HTTP requests?

To answer these questions we evaluated Sanare using three different applications: WordPress [18], GitLab [19] and ownCloud [20]. These three applications have very different

purposes and use cases. WordPress is a content management system (CMS) that is used by more than 37% of the top 10 million website.[8] With WordPress we can evaluate Sanare in a web application that is mostly used for blogging and provide content to their users. GitLab is a DevOps lifecycle tool with a Git repository and support for documentation with Wiki and issue tracking. With GitLab we aim to evaluate Sanare with a system that has a complex use case: storing thousands of files in a Git repository and keeping issues and users in a database. OwnCloud is a cloud storage server that offers a similar functionality to DropBox [44], which is to store files in a server and allow users to synchronize them using different client-side applications. This gives us a different use case from GitLab since files can be modified by the users; usually in a Git repository previous versions of the files are kept in a version history.

In our experiments we do not compare Sanare with others intrusion recovery systems since Sanare is the first intrusion recovery that is targeted for web applications that use several repositories. It would be possible to create a test scenario in which the the target application would only have a single repository to allow comparing Sanare with another system, such as [7], [8]. However, such comparison would not evaluate the full potential of Sanare in the sense that it was designed for multi-repository applications. Instead, we focused our experiments in evaluating the accuracy of Sanare in multi-repository applications.

The experiments were conducted in Google Cloud[9] using Google Compute Engine [45], their IaaS offering. This allowed us to distribute Sanare in different containers and use Sanare Manager to coordinate every component of the system using the Google Cloud APIs.[10] The machines had 4vCPU, 15GBs of memory and 20GBs SSDs.

9.1 Performance overhead

Sanare degrades the performance of the application due to the overhead of having every HTTP request, web service, database statement and file system operation being logged. To evaluate the performance penalty of running Sanare we deployed it with the three tested applications: WordPress, GitLab and ownCloud. We opted for the distributed deployment, i.e., each component of Sanare is running in its own container. Then we executed a workload for each application that was composed by a set of different HTTP requests for each application using the Locust open source load testing tool.[11] Locust allows to execute workloads of HTTP requests using distributed machines. The workloads are written and parameterized using scripts, then Locust executes the list of requests in the workload randomly. The workloads that we created for the experiments simulate how users interact with the application. For example, for WordPress the workload has two types of users: readers and writers. The readers only have read access to the content of the site and the writers can write new posts, modify pages, delete comments and upload files. The readers and writers

---

8. Usage statistics of content management systems – https://w3techs.com/technologies/overview/content_management
9. Google Cloud – https://cloud.google.com
10. Google Cloud APIs – https://cloud.google.com/apis
11. Locust – https://locust.io

are distributed 95%/5% for WordPress and 50%/50% for the remaining applications. These distributions are similar to those used in other workload testing tools, namely, in the much used YCSB [46]. For both ownCloud and GitLab we opted to use an update heavy workload since these kinds of applications are used for creating and modifying large quantities of files. All the requests in the experiments were executed randomly.

Figure 5 presents the performance, in terms of requests per second, of the three tested applications with the number of parallel users varying from 1,000 to 10,000. Varying the number of parallel users allows us to assess how the application with Sanare behaves when the traffic load increases. We tested each application with (green line) and without (blue line) Sanare. From 2,000 parallel users some requests start to fail. This happens because the application is not able to keep up with the traffic. The orange and red lines present the failures per second respectively with and without Sanare. As we can see in the figure, there is an average performance penalty of 17%, 15% and 12% for WordPress, GitLab and ownCloud, respectively. The performance degradation increases after 3,000 parallel users, which indicates that the machine hosting this particular application starts failing at around 3,000 parallel users.

9.2 Required storage

Sanare uses logs with the executed operations to perform recovery. These logs take significant space as they grow over time. To evaluate how much storage would be necessary to keep the execution logs, we executed a load of 1,000,000 HTTP requests using Locust. The requests were equally divided between read operations and write operations. This is not a strict concept in the tested applications since it is not possible to draw a line between read operations and write operations as we could do in, for example, a file system. As such, we defined a read operation as one that modifies as little as possible the state of the application. In WordPress this consists in getting a blog post or a page. In GitLab this consists in fetching the code with *git pull* or viewing the code tree of a project. In ownCloud a read operation consists in navigating through the file systems. For the write operations we assume those that modify as much as possible the state of the application. For WordPress this consists in creating a blog post that contains an image file and is shared in a social network. For GitLab it can be the creation of a new repository, pushing new code or removing a repository. For ownCloud it consists in uploading and deleting files. Locust executes these operations randomly with a constant probability.

Table 2 lists the amount of storage required to keep the logs. In each line of the table we have the three tested applications and in each column there is the required storage for each log. The last column sums the total of all the logs. The required storage is in the scale of the GBs of data. The total amount of storage required for 1,000,000 requests is lower than 30GBs. This amount of storage does not vary significantly from other intrusion recovery systems [7], [8], [26].
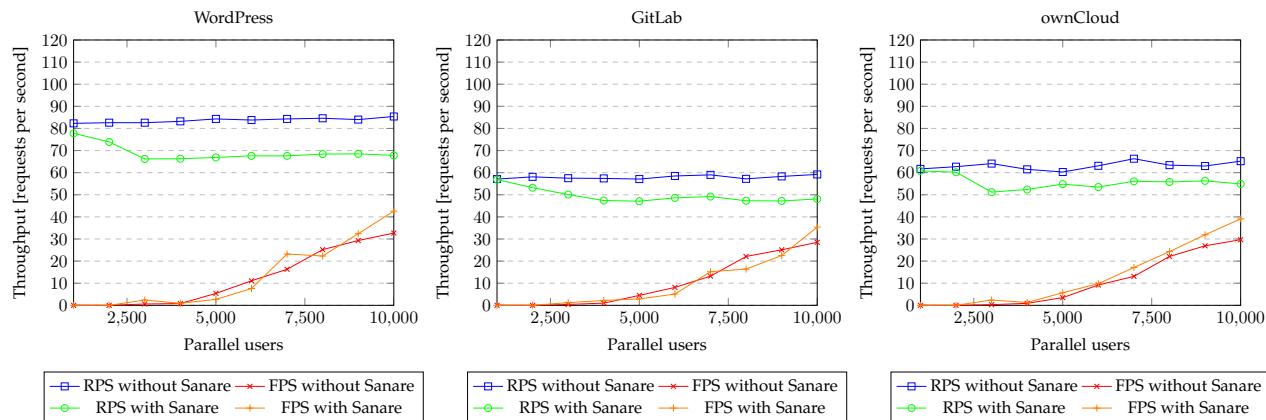
Fig. 5: Throughput in requests per second (RPS) and failures (FPS) of WordPress, GitLab and ownCloud in an out-of-the-box setup and configured with Sanare.

TABLE 2: Size of the logs (in GBs) of the three tested applications after executing 1,000,000 HTTP requests with Locust.

Application	DB	FS	WS	HTTP	Total
WordPress	6.56	1.13	2.32	7.37	**17.38**
GitLab	7.21	4.87	11.32	5.21	**28.61**
ownCloud	5.23	5.13	3.35	6.53	**20.24**

9.3 Time to recover

The time it takes to recover an application depends on the number of intrusions that need to be reverted. For a single intrusion there is only a single HTTP request to analyze and few database statements, file system operations and web services to revert. When this number scales to 100, Sanare needs to repeat the process 100 times. To evaluate how long it takes to revert an intrusion, we tagged a set of HTTP requests as malicious from a log with 1,000,000 requests. These HTTP requests were selected randomly. They are not different from the legit requests besides being tagged as malicious. The goal of this experiment was to measure how long it takes to revert operations. Tagging random operations as malicious serves this purpose. The experiments were repeated the tests 10 times for each number of intrusions. In each test we executed a recovery process with Sanare. We varied the number of malicious operation from 10 to 100 and measure the time it took from the moment recovery started to the moment it finished. Figure 6 shows the average time to recover the three tested applications. The average time to recover grows linearly with the number of intrusions to recover. The values vary depending on the application. WordPress was the fastest application to recover and ownCloud the slowest. This experiment shows that the time to recover depends on two factors: the number of intrusions to recover from and the complexity of the application. A single intrusion takes between 1.8 (WordPress) to 6 seconds (ownCloud).

9.4 Monetary cost

Sanare was designed to be deployed alongside the application it is protecting, but it may require extra machines in the cloud to run its components. It is possible to deploy Sanare
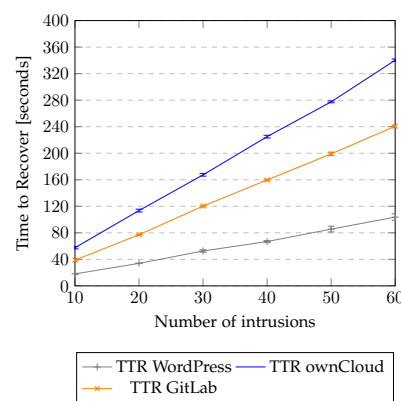

Fig. 6: Average Time To Recover (TTR) the three tested applications with the number of intrusions varying from 10 to 60 with a confidence interval of 95%.

in the ways mentioned earlier (embedded, single container, or distributed containers). To evaluate these monetary costs, we calculated how much it would cost to run Sanare in the three deployments. These values were calculated assuming the application is running in a virtual machine with 4 vCPUs and 16GBs RAM and Sanare modules run in similar machines.

TABLE 3: Additional monetary cost of running Sanare monthly and yearly in the three deployment modes.

Deployment mode	Monthly cost	Yearly cost
Embedded	$0	$0
Single container	$126	$1,512
Distributed	$378	$4,536

Table 3 lists the monetary cost of running Sanare in the three different deployment modes: embedded, Sanare is running in the same container of the application therefore there is no monetary cost associated; single container, every module of Sanare is running in a single container apart from the applications; and distributed, each of the components of Sanare that can be distributed (HTTP agent, Web Service agent and Sanare manager) is running in its own dedicated container.

There is another cost associated with Sanare that is the cost of keeping the logs of the application. Sanare gives the option of garbage collecting old logs to save space, however it is not possible to recover from operations that were discarded. Keeping old logs allows the administrator to recover from intrusions that were detected after a considerable amount of time. Table 4 lists the costs of storing logs in a cloud storage. Unlike the cost of running Sanare, it is not possible to predict the cost the logs will take over time since that depends on the traffic the application receives and, as we see in Table 2, the required storage for the logs is different for each application. So, we assume that the application receives an average of 1,000,000 requests per month and that results in 22GBs (roughly the average of the three applications tested in Table 2). This means that the logs grow 22GB each month until they are garbage collected, in other words, by the end of the year the cost is not the cost of storing 22GBs times 12 months, but instead the cost of storing 22GBs for one month plus 44GBs for another month and so on. Table 4 lists the cost of keeping the logs for an application that receives 1,000,000 HTTP requests per month. In the table there are two types of costs: cloud storage and cold storage. Cloud storage is a standard option that is used to keep data that needs to be constantly available and cold storage is an option to keep data that is rarely used. We believe that cold storage is the best option to keep older logs since they are only consulted when an intrusion happens.

TABLE 4: Monetary Sanare monthly and yearly cost of keeping 1,000,000 HTTP requests from the log in a cloud and cold storage

Storage type	First month	Yearly cost
Cloud storage	$3.71	$289.38
Cold storage	$0.26	$20.28

9.5 Results of the matching model

We used three metrics to evaluate the models that match the HTTP requests with the corresponding operations. These metrics take as input the total number of True Positives and True Negatives, i.e., the samples that the model correctly classified as a match and non-match, and False Positive and False Negatives, i.e., the samples that model failed to classify as match and non-match. The metrics we used are *Precision*, Recall and F1-Score which is calculated as follows:

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives} \tag{2}$$

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives} \tag{3}$$

$$F1Score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{4}$$

The results of these metrics are presented in Table 5. For the experiments we used a dataset composed by 1,000,000 examples of HTTP requests with corresponding operations. The dataset is composed by requests from the three tested applications. Matchare reaches close to 100% in all the metrics, but not 100%. Therefore, some operations can be identified as malicious, even if they are not (false positives),

while others may be identified as legit, while they should be identified as malicious (false negatives). Due to this, Sanare may be configured to not perform recovery right after it finishes running Matchare. Instead, it may present the system administrator a list with the suggested operations Matchare found that should be reverted. Then the system administrator may choose if all operations should be reverted or only a subset of them. It is possible to configure Sanare to allow Matchare to suggest other operations that may be malicious, i.e., that have a high probability of being caused by the malicious HTTP request. By adjusting this probability threshold Matchare is able to identify all positives with the expense of suggesting too many operations to the system administrator.

9.6 Discussion of the results

The results of our experiments suggest how Sanare will perform in a real world scenario. The performance overhead is less than 17%. The storage to keep the required logs after 1,000,000 operations were executed varies from 17GBs to 20GBs depending on the application. The time to recover is linearly proportional to the number of HTTP requests that need to be reverted. In our experiments, it took around a minute to revert 10 HTTP requests, and between 2 and 5 minutes to revert 60 requests. Matchare was able to achieve a precision of around 97% and a similar recall. This means that for 100 HTTP requests that Sanare analyzes, it is able to correctly find every operations that causes changes to the state of the application for 97 of those requests. The remaining 3 requests contain operations that were not caused by the malicious HTTP requests or is missing some operations that were caused by them that should also be reverted.

The overall performance overhead is enough to be noticeable in some applications, however, the 17% downgrade in the throughput can be easily reduced by improving the computing power of the machines running Sanare. This is something that can be done without major effort when the application and Sanare are running in a cloud service. The storage required for the logs is significant and, because of that, we calculated how much it would cost to keep the logs in two different storage options. The yearly cost to store the logs can be reduced from around 300$ to 20$ if the logs are moved to a cold storage service.

10 RELATED WORK

The problem of intrusion recovery has been studied in other works for different types of applications. In [47] the authors present a generic mechanism to recover from intrusions that uses a combination of logs and checkpoints to keep track of the user operations. The proposed approach, called the three R's of recovery consists in performing recovery in three steps: Rewind the system to before the attack; Repair, by eliminating malicious operations from the log and Replay, by re-executing the remaining operations. An implementation of this approach is presented by the same author in [48], with an extensive experimental evaluation in [49]. In this work the three R's approach was implemented in an e-mail system, allowing system administrators to undo unintended actions from the e-mail server while keeping the effects from

TABLE 5: Evaluation of the three models using the Precision, Recall and F1-Score metrics.

Model	Precision %	Recall %	F1-Score %	True Positives	True Negatives	False Positives	False Negatives
Database	97.72	97.95	97.83	481,235	497,463	11,245	10,057
File System	97.29	98.38	97.83	486,905	491,493	13,574	8,028
Web Services	97.91	97.51	97.71	479,921	497,583	10,249	12,247

legitimate operations intact. Sanare takes some inspiration from this work, more specifically the system architecture was based from [48].

There is some work in the literature regarding intrusion recovery for web applications [3], [4], [5], [7]. These intrusion recovery mechanisms require modifications to the source code of the application server running it. These requirements are not in line with our system model since we assume that such modification cannot be done. There is another intrusion recovery system for web application that does not require modifications to the source code called Rectify [8] which uses machine learning techniques to match HTTP requests with database operations (and only this kind of operations). This assumption that it is not possible to modify the source code of the application is also adopted in our system and the mechanisms to correlate HTTP requests with database operations takes some inspiration from this work. There are other similarities from these works with ours; more specifically these systems assume a system architecture in which the state of the application is contained in a database. Our assumption of a typical web application does consider a database to store part of its state but we also assume that the web application may also use a local and remote file system to keep its state. In Sanare we also assume that the web application may interact with external web services which can be a challenge to recover from intrusions. This characteristic is also present in another recovery system called Aire [6] which deals with intrusion recovery from web applications that are distributed and interact through web services. In Aire there is a mechanism to propagate the recovery operations to the external web services which is similar to our approach that allows the system administrator to invoke special web services' operations during recovery.

In order to recover the database of the web application we use an algorithm that assesses the damage in the database and obtains a dependency graph to revert the effects of the attack in every affected record. This algorithm was inspired in [3], which explores the problem of intrusion recovery in databases. We extended the algorithm in [3] to take into account not only the dependencies at database level, but also at application level (HTTP requests), to draw an extended dependency graph. Another work that explores the dependencies among operations for recovery purposes is Retro [50]. Retro is an intrusion recovery that creates action history graphs that describe in detail the system execution of operations triggered by users. This graph is then used to selectively re-execute the correct operations in order to recover the system. Although we do not adopt the same recovery strategy (selective re-execution), we took some inspiration from the algorithm to obtain the dependency graph. More specifically, in Retro the authors take into account the processes and how they interact with objects (files), while we take into account the user session (from the HTTP protocol) and how they interact with the database

records.

Sanare recovers corrupted files in the file system using a combination of execution logs and multi-versioned files. There is some work in the literature regarding recovery in file systems. One example is the Elephant File System [51] that keeps every version of each file, allowing users to revert files to previous versions. Another file system that allows users to revert back files is S4 [52], which combines multi-versioned files with operation logs. Allowing system administrators to diagnose the damages caused by the attack and repair from it. RFS [53] is a plugin for existing file systems that can be attached to log operations and recover from intrusions. As opposed to [51] and [52], RFS does not revert to a previous version of the corrupted files; instead it rolls back the system to a point in time prior to the attack and re-executed every legitimate operation, similar to [47]. This approach is also adopted in Taser [54], a file system that uses logs to taint the files affected by an attack, allowing the administrator to revert not only the attacked files but also those that were indirectly contaminated by them. In order to track the tainted files Taser intercepts system calls, allowing it to register the processes that affected the files. This approach is similar to the one used by Sanare, in the sense that it also intercepts file system operations in order to trace the effects of the attack. Another file system that employs some of these techniques is IFS [55]. In this file system some processes only have access to an isolated environment that does not allow them to modify files, instead new copies are created. A similar isolation technique is adopted in Sanare to recreate the attack without corrupting the application.

RockFS [26] is an intrusion recovery service for cloud file systems. This service is capable of recovering single cloud and cloud-of-clouds file systems by using a combination of operation logs and multi-versioned files. RockFS takes advantage of the cloud services by dynamically allocating storage containers and virtual machines so that it can keep logs and perform recovery. This approach of using the cloud services to leverage the recovery mechanism is also adopted in different aspects in Sanare.

11 CONCLUSION

In this paper we presented Sanare, a "pluggable" Intrusion Recovery system for Web Applications. Sanare takes into account the current trends in web development and allows administrators to recover applications that rely on database, file system and external web services. Its modular architecture makes it possible to use Sanare in applications with different complexity levels and without requiring additional software modification to the source code of the applications. The presented results show that it is possible to recover from a single intrusion in a couple of seconds. Although Matchare is a relevant component of Sanare that allows it to find operations caused by an HTTP request it can also be used in different contexts. For example, Matchare can

be used to perform analytical analysis in web applications, by finding which HTTP requests are generating the most complex operations. Matchare can also be used for forensic analysis allowing experts to find how certain operations affected the state of the application.

ACKNOWLEDGMENTS

REFERENCES

[1] J. Williams and D. Wichers, "OWASP Top 10 - 2017 rcl - the ten most critical web application security risks," OWASP Foundation, Tech. Rep., 2017.
[2] CheckPoint, "2020 cyber security report," 2020.
[3] 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.
[4] İ. E. Akkuş and A. Goel, "Data recovery for web applications," in *Proceedings of the 40th IEEE/IFIP International Conference on Dependable Systems and Networks*, 2010, pp. 81–90.
[5] 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.
[6] 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.
[7] D. Nascimento and M. Correia, "Shuttle: Intrusion recovery for PaaS," in *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems*, 2015, pp. 653–663.
[8] 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, pp. 209–221.
[9] H. Lee, R. Grosse, R. Ranganath, and A. Y. Ng, "Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations," in *Proceedings of the 26th annual international conference on machine learning*, 2009, pp. 609–616.
[10] K. L. Ingham and H. Inoue, "Comparing anomaly detection techniques for HTTP," in *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection*, 2007, pp. 42–62.
[11] C. Kruegel, G. Vigna, and W. Robertson, "A multi-model approach to the detection of web-based attacks," *Computer Networks*, vol. 48, no. 5, pp. 717–738, 2005.
[12] G. Nascimento and M. Correia, "Anomaly-based intrusion detection in software as a service," in *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*, 2011.
[13] W. Robertson, G. Vigna, C. Kruegel, and R. Kemmerer, "Using generalization and characterization techniques in the anomaly-based detection of web attacks," in *Proceedings of the 13th Symposium on Network and Distributed System Security*, Feb. 2006.
[14] P. DuBois, *MySQL*. New riders publishing, 1999.
[15] K. Chodorow, *MongoDB: The Definitive Guide*. O'Reilly, 2013.
[16] MongoDB, Inc., "MongoDB," http://www.mongodb.org, Nov. 2018, last checked on Dec 29, 2020. [Online]. Available: http://www.mongodb.org
[17] ——, "MongoDB Manual," https://docs.mongodb.org/manual/, 2016, last checked on Dec 29, 2020. [Online]. Available: https://docs.mongodb.org/manual/
[18] A. Brazell, *WordPress Bible*. John Wiley and Sons, 2011.
[19] J. M. Hethey, *GitLab Repository Management*. Packt Publishing Ltd, 2013.
[20] B. Martini and K.-K. R. Choo, "Cloud storage forensics: ownCloud as a case study," *Digital Investigation*, vol. 10, no. 4, pp. 287–299, 2013.
[21] D. Bernstein, "Containers and cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.
[22] P. B. Menage, "Adding generic process containers to the Linux kernel," in *Linux Symposium*, 2007, p. 45.
[23] B. Gallmeister, *POSIX. 4 Programmers Guide: Programming for the real world*. O'Reilly, 1995.
[24] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, 2009.
[25] D. Ma and G. Tsudik, "A new approach to secure logging," *ACM Transactions on Storage*, vol. 5, no. 1, 2009.
[26] 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.
[27] M. Rash, *Linux Firewalls: Attack Detection and Response with iptables, psad, and fwsnort*. No Starch Press, 2007.
[28] K. Brown and B. Woolf, "Implementation patterns for microservices architectures," in *Proceedings of the 23rd Conference on Pattern Languages of Programs*. The Hillside Group, 2016, pp. 1–35.
[29] I. Rish *et al.*, "An empirical study of the naive bayes classifier," in *IJCAI 2001 workshop on empirical methods in artificial intelligence*, vol. 3, no. 22, 2001, pp. 41–46.
[30] C.-Y. J. Peng, K. L. Lee, and G. M. Ingersoll, "An introduction to logistic regression analysis and reporting," *The journal of educational research*, vol. 96, no. 1, pp. 3–14, 2002.
[31] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.
[32] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 9, pp. 1263–1284, 2009.
[33] T. Jayalakshmi and A. Santhakumaran, "Statistical normalization and back propagation for classification," *International Journal of Computer Theory and Engineering*, vol. 3, no. 1, pp. 1793–8201, 2011.
[34] T. Mackinnon, S. Freeman, and P. Craig, "Endo-testing: unit testing with mock objects," *Extreme programming examined*, pp. 287–301, 2000.
[35] F. Cristian, "Probabilistic clock synchronization," *Distributed computing*, vol. 3, no. 3, pp. 146–158, 1989.
[36] B. Momjian, *PostgreSQL: introduction and concepts*. Addison-Wesley, 2001, vol. 192.
[37] L. Dusseault and J. Snell, "Patch method for http," Internet Engineering Task Force (IETF), Tech. Rep., 2010.
[38] J. Snell and P. Hoffman, "Json merge patch, RFC 7386," Internet Engineering Task Force (IETF), Tech. Rep., 2014.
[39] S. Newman, *Building microservices: designing fine-grained systems*. O'Reilly, 2015.
[40] J. Thönes, "Microservices," *IEEE Software*, vol. 32, no. 1, pp. 116–116, 2015.
[41] M. Grinberg, *Flask web development: developing web applications with Python*. O'Reilly, 2018.
[42] C. Nedelcu, *Nginx HTTP Server: Adopt Nginx for Your Web Applications to Make the Most of Your Infrastructure and Serve Pages Faster Than Ever*. Packt Publishing Ltd, 2010.
[43] J. Shah and D. Dubaria, "Building modern clouds: using docker, kubernetes & google cloud platform," in *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, 2019, pp. 0184–0189.
[44] I. Drago, M. Mellia, M. M. Munafo, A. Sperotto, R. Sadre, and A. Pras, "Inside dropbox: understanding personal cloud storage services," in *Proceedings of the 2012 Internet Measurement Conference*, 2012, pp. 481–494.
[45] S. Krishnan and J. L. U. Gonzalez, "Google compute engine," in *Building your next big thing with Google cloud platform*. Springer, 2015, pp. 53–81.
[46] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, 2010, pp. 143–154.
[47] A. B. Brown and D. A. Patterson, "Rewind, repair, replay: three r's to dependability," in *Proceedings of the 10th ACM SIGOPS European Workshop*, 2002, pp. 70–77.
[48] A. Brown and D. Patterson, "Undo for operators: Building an undoable e-mail store," in *Proceedings of the USENIX Annual Technical Conference*, 2003, pp. 1–14.
[49] A. Brown, L. Chung, W. Kakes, C. Ling, and D. 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.
[50] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Intrusion recovery using selective re-execution," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010, pp. 89–104.

[51] 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.

[52] 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.

[53] N. Zhu and T. Chiueh, "Design, implementation, and evaluation of repairable file service," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2003, p. 217.

[54] A. Goel, K. Po, K. Farhadi, Z. Li, and E. D. Lara, "The Taser intrusion recovery system," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, vol. 39, no. 5, 2005, pp. 163–176.

[55] 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.
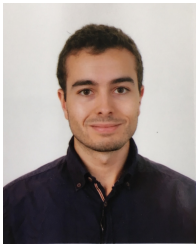
David R. Matos has a BSc (2012) and a MSc (2013) in Informatics Engineering from the Faculty of Sciences, University of Lisbon and a PhD (2019) in Computer Sciences and Engineering from Instituto Superior Técnico, University of Lisbon. He is currently a Postdoctoral researcher at FCiências-ID in the 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).